MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
15 August 2024

REVISED
6 December 2024

ACCEPTED FOR PUBLICATION
15 January 2025

PUBLISHED
29 January 2025

PAPER

SymbolNet: neural symbolic regression with adaptive dynamic pruning for compression

Ho Fung Tsoi[1,*], **Vladimir Loncar**[2,3], **Sridhara Dasu**[1] and **Philip Harris**[2,4]

[1] University of Wisconsin-Madison, Madison, WI, 53706, United States of America
[2] Massachusetts Institute of Technology, Cambridge, MA, 02139, United States of America
[3] Institute of Physics, Belgrade, Serbia
[4] Institute for Artificial Intelligence and Fundamental Interactions, Cambridge, MA, 02139, United States of America
[*] Author to whom any correspondence should be addressed.

E-mail: ho.fung.tsoi@cern.ch

Keywords: symbolic regression, neural network, dynamic pruning, model compression, low latency, FPGA

Abstract

Compact symbolic expressions have been shown to be more efficient than neural network (NN) models in terms of resource consumption and inference speed when implemented on custom hardware such as field-programmable gate arrays (FPGAs), while maintaining comparable accuracy (Tsoi *et al* 2024 *EPJ Web Conf.* **295** 09036). These capabilities are highly valuable in environments with stringent computational resource constraints, such as high-energy physics experiments at the CERN Large Hadron Collider. However, finding compact expressions for high-dimensional datasets remains challenging due to the inherent limitations of genetic programming (GP), the search algorithm of most symbolic regression (SR) methods. Contrary to GP, the NN approach to SR offers scalability to high-dimensional inputs and leverages gradient methods for faster equation searching. Common ways of constraining expression complexity often involve multistage pruning with fine-tuning, which can result in significant performance loss. In this work, we propose `SymbolNet`, a NN approach to SR specifically designed as a model compression technique, aimed at enabling low-latency inference for high-dimensional inputs on custom hardware such as FPGAs. This framework allows dynamic pruning of model weights, input features, and mathematical operators in a single training process, where both training loss and expression complexity are optimized simultaneously. We introduce a sparsity regularization term for each pruning type, which can adaptively adjust its strength, leading to convergence at a target sparsity ratio. Unlike most existing SR methods that struggle with datasets containing more than $\mathcal{O}(10)$ inputs, we demonstrate the effectiveness of our model on the LHC jet tagging task (16 inputs), MNIST (784 inputs), and SVHN (3072 inputs).

1. Introduction

Symbolic regression (SR) is a supervised learning method that searches for analytic expressions that best fit the data (see [1] for recent efforts). Unlike traditional regression methods, such as linear and polynomial regression, SR can model a much broader range of complex datasets because it does not require a pre-defined functional form, which itself is dynamically evolving in the fit.

By expressing models in symbolic forms, SR facilitates human interpretation of the data, enabling the potential inference of underlying principles governing the observed system, in contrast to the opaque nature of black-box deep learning (DL) models. A historical example is Max Planck's 1900 empirical fitting of a formula to the black-body radiation spectrum [2], known as Planck's law. This symbolically fitted function not only inspired the physical derivation of the law but also played a key role in the revolutionary development of quantum theory. Moreover, due to its compact representation compared to most DL models, SR can also be used as a distillation method for model compression [3]. This can accelerate inference time and reduce computational costs, making it particularly valuable in resource-constrained environments.

However, a significant drawback of SR is its inherent complexity. The search space for equations expands exponentially with the number of building blocks (variables, mathematical operators, and constants), making it a challenging combinatorial problem. In fact, finding the optimal candidate has been shown to be NP-hard [4].

Genetic programming (GP) has traditionally been the primary approach to SR [5–9]. It constructs expressions using a tree representation, where the algebraic relations are reflected in the tree's structure. The tree's lowest nodes consist of constants and variables, while the nodes above represent mathematical operations. GP grows an expression tree in a manner that mimics biological evolution, employing node mutations and subtree crossovers to explore variations in expressions. Candidates are grouped into generations and participate in a tournament selection process, where individuals with the highest fitness scores survive and advance. Although GP has been successful in discovering human-interpretable solutions for many low-dimensional problems, its discrete combinatorial approach and lengthy search times make it unsuitable for large and high-dimensional datasets.

An alternative approach to SR involves using a DL framework [10–18], such as training a neural network (NN) with activation functions that generalize to broader mathematical operations, including unary functions like $(\cdot)^2$ and $\sin(\cdot)$, as well as binary functions like $+$ and $\times$. The NN is trained with enforced sparse connections, ensuring that the final expressions derived from the NN are compact enough to be human-interpretable or efficiently deployable in resource-constrained environments. In addition to benefiting from faster gradient-based optimization, the DL approach can utilize GPUs to accelerate both training and inference, whereas GP-based algorithms are typically limited to CPUs. The key to training a NN that produces effective SR results, balancing model performance and complexity, is controlling sparsity. However, most recent developments in using NNs for SR have relied on less efficient pruning methods to achieve the necessary sparsity, requiring multiple training phases with hard-threshold pruning followed by fine-tuning. These multistage frameworks often result in significant performance compromise, as accuracy and sparsity are optimized in separate training phases. The lack of an integrated sparsity control scheme prevents DL approaches from fully realizing their potential to expand SR's applicability to a broader range of problems.

In this contribution, we introduce `SymbolNet`[5], a DL approach to SR utilizing NN in a novel and SR-dedicated pruning framework, specifically designed as a model compression technique for low-latency inference applications, with the following properties:

- **End-to-end single-phase dynamic pruning.** It requires only a single training phase without the need for fine-tuning. Unlike traditional methods that rely on a pre-specified threshold for 'heavy-hammer-style' pruning, this framework, inspired by dynamic sparse training [20], introduces a trainable threshold associated with each model weight. The pruning of a weight is automatically determined by the dynamic competition between the weight and its threshold. We extend this concept to also prune input features, automating feature selection within the framework without the need for external packages or additional steps. Similarly, we introduce operator pruning, which involves dynamically transforming more complex mathematical operators into simpler arithmetic operations. Overall, a trainable threshold is independently assigned to each model weight, input feature, and mathematical operator, enabling dynamic pruning to occur within a single training phase.
- **Convergence to the desired sparsity ratios.** For each of the four pruning types—model weights, input features, unary operators, binary operators—we introduce a regularization term that adaptively adjusts its strength in relation to the training loss. This allows the model to converge to the desired sparsity ratios, as specified by the user.
- **Scalability of SR to high-dimensional datasets.** Dynamic pruning can enforce strong sparsity while being optimized simultaneously with model performance. Combined with gradient-based optimization, this approach enables the generation of optimal and compact expressions that can effectively fit large and complex datasets.

As far as we are aware, most of the SR literature has primarily tested their methods on datasets with input dimensions below $\mathcal{O}(10)$. These methods have yet to be demonstrated as efficient solutions for high-dimensional problems such as MNIST and beyond. Our framework leverages the compact representation of symbolic expressions and is designed to enable low-latency inference on custom hardware such as field-programmable gate arrays (FPGAs), targeting high-dimensional datasets without ground-truth equations, as commonly encountered in high-energy experiments at the CERN large Hadron Collider

[5] A tensorflow [19] implementation code is available at https://github.com/hftsoi/SymbolNet

(LHC). We validate our framework by learning compact expressions from datasets with input dimensions ranging from $\mathcal{O}(10)$ to $\mathcal{O}(1000)$, demonstrating the effectiveness of the model in solving more practical problems for deployment with constrained computational resources.

The paper is structured as follows. Section 2 discusses some of the previous efforts related to this work. Section 3 details the architecture and training framework of `SymbolNet`. Section 4 describes the datasets and outlines the experiments. Section 5 presents the results, comparing `SymbolNet` with baseline methods in obtaining compact and competitive expressions, and also comparing `SymbolNet` with typical compression methods in the context of FPGA deployment for sub-microsecond latency.

2. Related work

SR has started to gain significant attention over the past decade [1, 21–23]. Approaches to SR have traditionally been based on GP, first formulated in [24], arising from the idea of creating a program that enables a computer to solve problems in a manner similar to natural selection and genetic evolution. `Eureqa` [5] is one of the first GP-based SR libraries, but it was developed as a commercial proprietary tool, limiting its accessibility for scientific research. `PySR` [6] is a recently developed open-source library built upon the classic GP approach, augmented with a novel evolve-simplify-optimize loop, making it suitable for practical SR and the automatic discovery of scientific equations [25–31]. Other examples include `gplearn` [7], `Operon` [8], and `GP − GOMEA` [9]. For scalable SR, feature selection can be employed prior to performing the equation search. For instance, `PySR` integrates an external random forest regression algorithm to handle high-dimensional datasets by selecting a subset of features based on their relevance to a regression task, which are then passed to the main GP algorithm. However, this approach is practically viable only if the number of selected features remains low, such as below ten, as the search algorithm is still being performed with GP. Alternative methods include the fast function extraction [32, 33] and differentiable GP approach [34], which aim to address the scalability challenges of SR but are not NN-based.

DL has been highly successful in addressing complex problems in fields such as computer vision and natural language processing [35], yet, its application in the domain of SR has not been thoroughly investigated. Equation learner (`EQL`) [10–12] is one of the first NN architectures proposed for performing SR. The approach involves constructing a NN using primitive mathematical operations for neuron activation, training it to achieve a sparse structure through pruning [36–38], and then unrolling it to obtain the final expression. To prevent the formation of overly complex equations, a three-stage training scheme is employed. In the first stage, a fully-connected NN is trained without regularization, focusing solely on minimizing regression error to allow the parameters to vary freely and establish a solid starting point. In the second stage, L_1 regularization is imposed to encourage small weights, leading to the emergence of a sparse connection pattern. Finally, all weights below a certain threshold are set to zero and frozen, effectively enforcing a fixed L_0 norm. The remaining weights are fine-tuned without any regularization. Later research demonstrated that using the $L^*_{0.5}$ regularizer [13, 14], which is constructed from a piecewise function and serves as a smooth variant of $L_{0.5}$, can enforce stronger sparsity than L_1. These methods were primarily tested on simple dynamic system problems with input dimensions of $\mathcal{O}(1)$.

Other variants of NN-based approaches to SR include `OccamNet` [15], which uses a NN to define a probability distribution over a function space, optimized using a two-step method that first samples functions and then updates the weights so that the probability mass is more likely to produce better-fitting functions. Sparsity is maximized by introducing temperature-controlled softmax layers that sample sparse paths through the NN. `DSR` [16] employs an autoregressive recurrent NN to generate expressions sequentially, optimizing them based on reinforcement learning. `MathONet` [17] is a Bayesian learning framework that incorporates sparsity as priors and is applied to solve differential equations. `N4SR` [18] is a multistage learning framework that allows integration of domain-specific prior knowledge. Another class of approaches utilizes transformers pre-trained on large-scale synthetic datasets to generate symbolic expressions from data. For those interested, further details can be found in [39–42].

However, these approaches primarily focus on small and low-dimensional datasets and do not explore scalability. GP-based methods are inherently difficult to scale due to their discrete search strategies, while DL-based methods lack an efficient and SR-dedicated approach to constrain model size. Our method, detailed in the following section, attempts to address this gap.

3. SymbolNet architecture

In this section, we describe the model architecture and training framework. We construct a NN composed of symbolic layers, using generic mathematical operators as activation functions. Trainable thresholds are

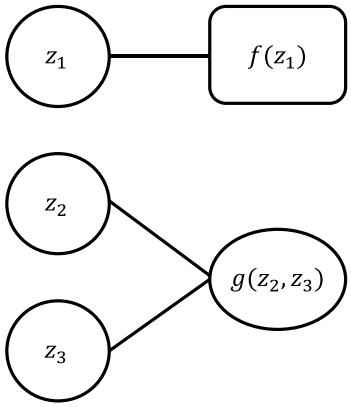

Figure 1. A symbolic layer composed of three linear transformation nodes z, activated by a unary operator f and a binary operator g.

introduced to dynamically prune model weights, input features, and operators. Additionally, a self-adaptive regularization term is introduced for each pruning type to ensure convergence to the target sparsity ratio.

3.1. Neural SR

We adapt the `EQL` architecture introduced in [10] as our starting point for approaching SR using NNs. This basic architecture functions similarly to a multilayer perceptron [43–45], with the key difference being that each hidden layer is generalized to a symbolic layer. A symbolic layer consists of two operations: the standard linear transformation of outputs from the previous layer, followed by a layer of heterogeneous unary ($f(x)$: $\mathbb{R} \to \mathbb{R}$, e.g. x^2, $\sin(x)$, and $\exp(-(x)^2)$) and binary ($g(x, y)$: $\mathbb{R}^2 \to \mathbb{R}$, e.g. $x + y$, xy, and $\sin(x)\cos(y)$) operations as activation functions. Thus, if a symbolic layer contains u unary operators and b binary operators, its input dimension is $u + 2b$ and its output dimension is $u + b$. An example of a symbolic layer is illustrated in figure 1, where $u = 1$ and $b = 1$.

3.2. Dynamic pruning per network component type

The expressiveness of NNs is partly due to the large number of adjustable parameters they contain. Even a shallow NN can be over-parameterized in the context of symbolic representation. Therefore, controlling sparsity is crucial for NN-based SR, while still maintaining reasonable model performance. An example of a sparsely connected NN composed of symbolic layers, which generates a compact expression, is illustrated in figure 2.

Dynamic sparse training, introduced in [20], is an improved alternative to the traditional 'heavy-hammer-style' pruning that applies a fixed threshold for all model weights. Instead, this method defines a threshold vector for each layer, where the thresholds are trainable and can be updated through backpropagation [46]. This makes pruning a dynamic process, with weights and their associated thresholds continuously competing. Pruning occurs more precisely as it takes place at each training step rather than between epochs. Since both the weight and its threshold continue to update even after pruning, a pruned weight can potentially be recovered if the competition reverses. This approach follows a single training schedule, eliminating the need for multistage training with separate fine-tuning, while simultaneously optimizing both model weights and sparsity ratios. This simple yet effective framework is particularly valuable for NN-based SR.

Inspired by this method, we incorporate it into our model to perform a weight-wise pruning. We then generalize the idea to also prune input features and mathematical operators, introducing a regularization strategy to achieve convergence to the desired sparsity ratios.

To implement dynamic pruning, [20] used a step function as a masking function, along with a piecewise polynomial estimator that is nonzero and finite in the range $[-1, 1]$ but zero elsewhere. This approach was used to approximate the derivative in the backward pass, preventing a zero gradient in the thresholds, since the derivative of the original step function is almost zero everywhere. In this work, we employ a similar binary step function $\theta(x) = \mathbf{1}_{x>0}$ (i.e. 1 if $x > 0$, otherwise 0) for masking in the forward pass, along with a smoother estimator in the backward pass using the derivative of the sigmoid function $\frac{\mathrm{d}\theta(x)}{\mathrm{d}x} \approx \frac{\kappa \mathrm{e}^{-\kappa x}}{(1+\mathrm{e}^{-\kappa x})^2}$ with $\kappa = 5$ (a very high κ would effectively turn it into a delta function, while a very low κ would make it flat and less sensitive to the step location), which is nonzero everywhere.

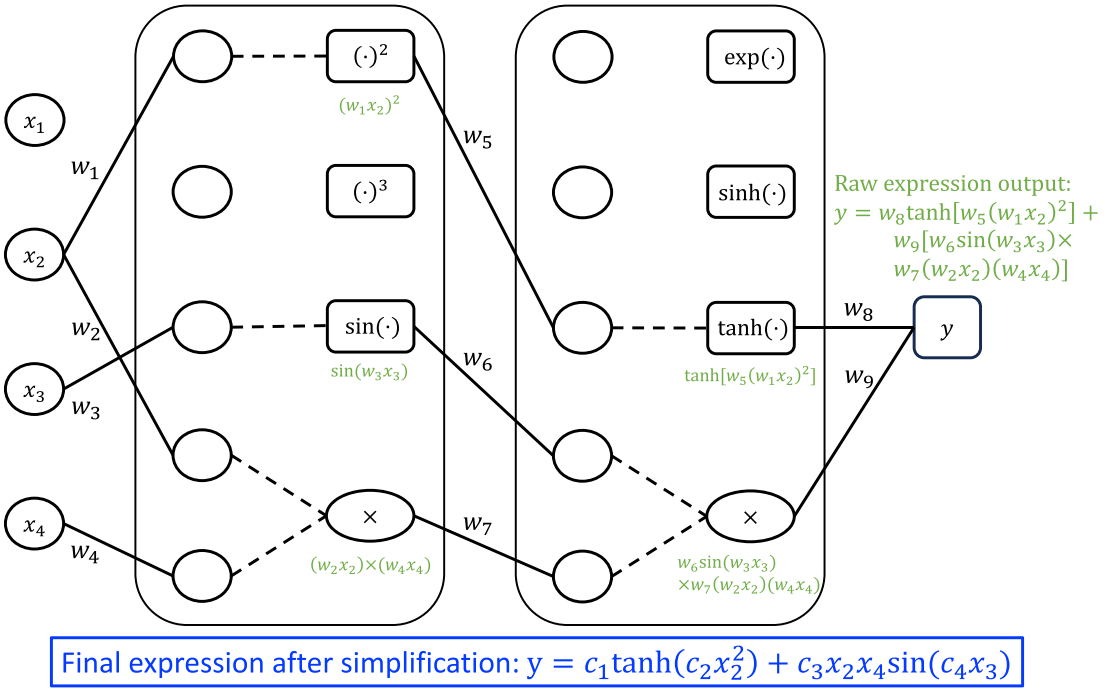

Figure 2. An example NN with four input features (x), two symbolic layers (large rectangles), and one output node (y). Each symbolic layer contains five linear transformations (empty circles), followed by three unary operations (small rectangles) and one binary operation (ovals). The solid lines represent nonzero model weights (w) for the linear transformation, while the dashed lines indicate activation by mathematical operations. The intermediate expression outputs are shown as green text. The final expression from this example model, after simplifying the constants from w to c, is shown in blue text: $y = c_1 \tanh(c_2 x_2^2) + c_3 x_2 x_4 \sin(c_4 x_3)$. This illustrates the basic architecture of `SymbolNet` before incorporating additional components for adaptive dynamic pruning.

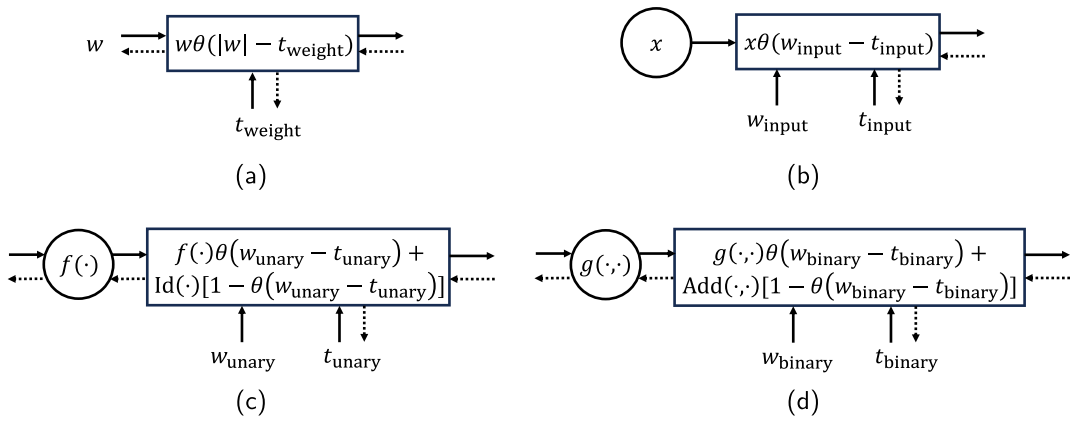

Figure 3. Schematic sketch of the SR-dedicated dynamic pruning mechanism within the `SymbolNet` architecture: (a) model weights, (b) input features, (c) unary operators, and (d) binary operators. Solid arrows represent the forward pass, while dotted arrows represent the backward pass, linking the trainable parameters. The `SymbolNet` architecture is constructed by integrating these elements with the basic network architecture illustrated, for example, in figure 2.

We introduce a pruning mechanism for model weights, input features, unary operators, and binary operators, as illustrated in figure 3 and explained in the following subsections.

3.2.1. Pruning of model weights

For each model weight w, we associate a trainable pruning threshold t_{weight}.

In each layer that performs a linear transformation with input dimension n and output dimension m, there are a weight matrix and a threshold matrix: $\boldsymbol{w}, \boldsymbol{t}_{\text{weight}} \in \mathbb{R}^{n \times m+m}$, where the second m corresponds to the number of bias terms. Each threshold is initialized to zero and is clipped to the range of $[0, \infty)$ during parameter updates, as the magnitude of each w is unbounded.

The weight-wise pruning in each layer is implemented using the step function matrix $\boldsymbol{\Theta}_{\text{weight}} = \theta(|\boldsymbol{w}| - \boldsymbol{t}_{\text{weight}}) \in \mathbb{R}^{n\times m+m}$, applied as follows: $w_{ij} \to (\boldsymbol{w} \odot \boldsymbol{\Theta}_{\text{weight}})_{ij} = w_{ij}\theta(|w_{ij}| - t_{\text{weight},ij})$, as illustrated in figure 3(a).

Concatenating all layers of the architecture, we denote $\boldsymbol{W} = \{\boldsymbol{w}\}$ and $\boldsymbol{T}_{\text{weight}} = \{\boldsymbol{t}_{\text{weight}}\}$, with the total dimension being the total number of weights n_{weight}.

3.2.2. Pruning of input features

Similarly, for each input node x, we associate it with an auxiliary weight w_{input} and a trainable pruning threshold t_{input}.

Since there is already a linear transformation of the inputs when propagating to the next layer, and because only the relative distance between the weight and threshold matters on the pruning side, a trainable auxiliary weight would be redundant. Therefore, we fix all $w_{\text{input}} = 1$ and set them as untrainable. Each threshold is initialized to zero and is clipped to the range of $[0, 1]$ during parameter updates.

For the input vector $\boldsymbol{x} \in \mathbb{R}^{n_{\text{input}}}$, there is an auxiliary weight vector and a threshold vector $\boldsymbol{W}_{\text{input}}, \boldsymbol{T}_{\text{input}} \in \mathbb{R}^{n_{\text{input}}}$. The pruning operation is implemented using the step function vector $\boldsymbol{\Theta}_{\text{input}} = \theta(\boldsymbol{W}_{\text{input}} - \boldsymbol{T}_{\text{input}})$, applied as follows: $x_i \to (\boldsymbol{x} \odot \boldsymbol{\Theta}_{\text{input}})_i = x_i\theta(1 - T_{\text{input},i})$, as illustrated in figure 3(b). In other words, an input node is pruned when its associated threshold value reaches 1.

3.2.3. Pruning of mathematical operators

Instead of pruning mathematical operators to zero directly, we simplify them to basic arithmetic operations, as pruning to zero can be accomplished by pruning the corresponding weights in their linear transformations when propagating to the next layer.

Similar to input pruning, for each unary operator $f(\cdot)$: $\mathbb{R} \to \mathbb{R}$, we associate it with an untrainable auxiliary weight fixed at $w_{\text{unary}} = 1$ and a trainable pruning threshold $t_{\text{unary}} \in [0, 1]$. The pruning operation is applied to each unary operator node as $f(\cdot) \to [f(\cdot)\theta(1 - t_{\text{unary}}) + \text{Id}(\cdot)(1 - \theta(1 - t_{\text{unary}}))]$, where $\text{Id}(\cdot)$ is the identity function, as illustrated in figure 3(c). This means a unary operator will be simplified to an identity operator when necessary, helping to prevent overfitting and reducing overly complex components, such as function nesting (e.g. $\sin(\sin(\cdot))$). We denote all thresholds of this type by a vector $\boldsymbol{T}_{\text{unary}}$, with a dimension of n_{unary}, representing the total number of unary operators in the architecture.

Similarly, for each binary operator $g(\cdot,\cdot)$: $\mathbb{R}^2 \to \mathbb{R}$, we associate it with an untrainable auxiliary weight fixed at $w_{\text{binary}} = 1$ and a trainable pruning threshold $t_{\text{binary}} \in [0, 1]$. The pruning operation is applied to each binary operator node as $g(\cdot,\cdot) \to [g(\cdot,\cdot)\theta(1 - t_{\text{binary}}) + \text{Add}(\cdot,\cdot)(1 - \theta(1 - t_{\text{binary}}))]$, where $\text{Add}(\cdot,\cdot)$ is the addition operator, as illustrated in figure 3(d). This means a binary operator will be simplified to an addition operator when necessary, which is motivated by the fact that addition is typically simpler to compute than other more complex binary operators. For instance, an addition operation requires only one clock cycle to execute on an FPGA, while other binary operators may generally take significantly longer. We denote all thresholds of this type by a vector $\boldsymbol{T}_{\text{binary}}$, with a dimension of n_{binary}, representing the total number of binary operators in the architecture.

3.3. Self-adaptive regularization for sparsity

We introduce a regularization term for each threshold type to encourage large threshold values. For the model weight thresholds, we adapt the approach in [20] and use the form $L_{\text{threshold}}^{\text{weight}} = \frac{1}{n_{\text{weight}}} \sum_{i=1}^{n_{\text{weight}}} \exp(-T_{\text{weight},i})$, where the sum runs over all the weight thresholds in the architecture. Since each threshold $T_{\text{weight},i} \in [0, \infty)$ is unbounded, this sum of the exponentials will not drop to zero immediately, even if some thresholds become large. For other types with bounded thresholds, we use a more efficient form with a single exponential: $L_{\text{threshold}}^{\text{aux}=\{input,unary,binary\}} = \exp(-\frac{1}{n_{\text{aux}}} \sum_{i=1}^{n_{\text{aux}}} T_{\text{aux},i})$.

For each of the regularizers, we introduce a *decay factor*:

$$D(s; \alpha, d) = \exp\left[-\left(\frac{\alpha}{\alpha - \min(s, \alpha)}\right)^d + 1\right]. \tag{1}$$

Here, the sparsity $s \in [0, 1]$ represents the ratio of pruned parameters or operators, evaluated at each training step. The parameter $\alpha \in [0, 1]$ sets the target sparsity ratio, and d is the decay rate that controls how quickly the sparsity driving slows down. When the regularizer is multiplied by this term, it effectively slows down high-threshold driving as the sparsity ratio s grows, and eliminates the regularizer when $s \geqslant \alpha$. The decay factor is illustrated in figure 4 for different decay rates d.

The full regularization term for each threshold type takes the form $L_{\text{sparse}} = L_{\text{error}} D(s; \alpha, d) L_{\text{threshold}}$, where L_{error} represents the training loss (e.g. regression error). The strength of the threshold regularizer $L_{\text{threshold}}$ is adaptively adjusted by the product $L_{\text{error}} D(s; \alpha, d)$, which is initially set by the training loss.

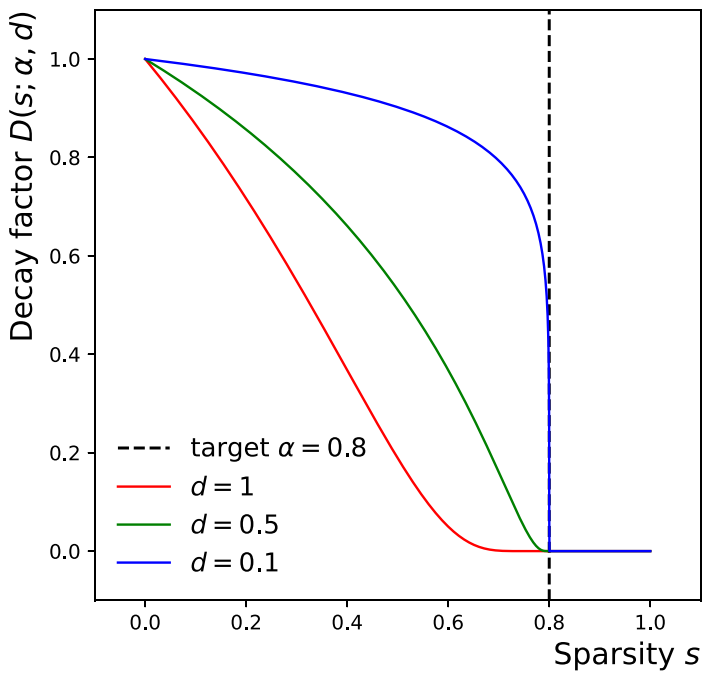

Figure 4. The decay factor, $D(s;\alpha,d)$, is employed to reduce the rate of increase in high-thredhold values as the sparsity ratio (s) approaches its target value (α). High-threshold driving is paused when $s \geqslant \alpha$. The profiles of $D(0 \leqslant s \leqslant 1)$ for a target sparsity ratio of $\alpha = 0.8$ at three different decay rates (d) are shown.

At the start of the training, when the sparsity ratio is initialized at $s = 0$, the regularization is as significant as the training loss itself: $L_{\text{sparse}} = L_{\text{error}}$, since $L_{\text{threshold}} = D = 1$. The presence of $L_{\text{threshold}}$ drives the thresholds to increase through backpropagation, which in turn reduces $L_{\text{threshold}}$.

As the sparsity ratio begins to increase $s > 0$, the decay factor D drops below 1, slowing the growth of the thresholds. This process continues until the target sparsity ratio is reached, where D eventually drops to 0, or until further increasing the sparsity would significantly elevate the training loss, even if the target is not yet reached. Thus, the regularizer L_{sparse} is designed to guide the sparsity ratio toward convergence at the desired target value.

3.4. Training framework

Integrating all together, for a dataset $\{(\boldsymbol{x}^i, \boldsymbol{y}^i)\}_{i=1}^N$ with inputs $\boldsymbol{x}^i \in \mathbb{R}^{n_{\text{input}}}$ and labels $\boldsymbol{y}^i \in \mathbb{R}^{n_{\text{output}}}$, the algorithm aims to solve the following multi-objective optimization problem for a network $\phi : \mathbb{R}^{n_{\text{input}}} \to \mathbb{R}^{n_{\text{output}}}$ with outputs $\hat{\boldsymbol{y}}^i = \phi(\boldsymbol{x}^i)$:

$$\boldsymbol{W}^*, \boldsymbol{T}^*_{\text{weight}}, \boldsymbol{T}^*_{\text{aux}} = \underset{\boldsymbol{W}, \boldsymbol{T}_{\text{weight}}, \boldsymbol{T}_{\text{aux}}}{\arg\min}\ \mathcal{L}\left(\boldsymbol{W}, \boldsymbol{T}_{\text{weight}}, \boldsymbol{T}_{\text{aux}}; \alpha_{\text{weight}}, \alpha_{\text{aux}}, d\right), \tag{2}$$

where

$$\begin{aligned} \mathcal{L} &= L_{\text{error}} + L^{\text{weight}}_{\text{sparse}} + L^{\text{aux}}_{\text{sparse}}, \\ L^{\text{weight}}_{\text{sparse}} &= L_{\text{error}} \times D\left(s_{\text{weight}}; \alpha_{\text{weight}}, d\right) \times \frac{1}{n_{\text{weight}}} \sum_{i=1}^{n_{\text{weight}}} \exp\left(-T_{\text{weight},i}\right), \\ L^{\text{aux}}_{\text{sparse}} &= L_{\text{error}} \times D\left(s_{\text{aux}}; \alpha_{\text{aux}}, d\right) \times \exp\left(-\frac{1}{n_{\text{aux}}} \sum_{i=1}^{n_{\text{aux}}} T_{\text{aux},i}\right), \end{aligned} \tag{3}$$

with aux $= \{\text{input}, \text{unary}, \text{binary}\}$. We use the mean squared error (MSE) as the training loss: $L_{\text{MSE}} = \frac{1}{N n_{\text{output}}} \sum_{i=1}^{N} \sum_{j=1}^{n_{\text{output}}} (y^i_j - \hat{y}^i_j)^2$. For our experiments presented in the next section, we set $d = 0.01$ (a lower value is preferred to avoid too fast a decay rate where most weights are pruned at early epochs before the model learns anything from data). The remaining free parameters are α_{weight}, α_{input}, α_{unary}, and α_{binary}, which respectively determine the target sparsity ratios for different types of pruning.

In this single-phase training framework, the overall sparse structure is divided into sparse substructures for the model weights, input features, unary operators, and binary operators, respectively. In particular, a set

of sparse input features is automatically searched without the need for an external feature selection process. Each of these sparse substructures is dynamically determined by the competition between the corresponding weights and thresholds. Furthermore, the sparse structure dynamically competes with the regression performance, allowing both to be optimized simultaneously. This approach enables direct specification of target sparsity ratios for each component (weights, features, and operators), rather than requiring the indirect tuning of multiple regularization parameters whose relationships to the resulting sparsity levels are more difficult to predict. The final symbolic expressions are derived by unrolling the trained network.

4. Experimental setup

4.1. Expression complexity

To quantify the size of a symbolic model, we use a metric called expression complexity [6]. This metric is computed by counting all possible steps involved in traversing the expression tree, which corresponds to the total number of nodes in the tree. For example, the expression $y = c_1 \tanh(c_2 x_2^2) + c_3 x_2 x_4 \sin(c_4 x_3)$, generated in figure 2, has a complexity of 17, as illustrated in figure 5. The `Sympy` library [47] provides the *preorder_traversal* method, which can be used to calculate the number of steps required to traverse a given expression.

We assume that all types of tree node (mathematical operator, input variable, and constant) have the same complexity of 1 during the counting process. However, this assumption may not hold universally. For example, in the context of FPGAs, computing $\tan(\cdot)$ might require significantly more clock cycles than computing $\sin(\cdot)$. Conversely, if mathematical functions are approximated using lookup tables (LUTs) that each requires only one clock cycle to compute, the assumption that all unary operators are equally weighted becomes valid [3]. The definition of complexity for each node type depends on the specific implementation and resource allocation strategy, which we do not explore in depth. Instead, we adopt the most straightforward assumption for our experiments.

4.2. Baseline for comparison

From DL-based methods, we use the `EQL` architecture [10–12], trained within a three-stage pruning framework [13], as the baseline for comparison purposes. In this baseline framework, model weights are regularized using the smoothed $L_{0.5}$ term, referred to as $L_{0.5}^*$, which is controlled by a free parameter λ:

$$\begin{aligned} \mathcal{L} &= L_{\text{MSE}} + \lambda L_{0.5}^* \\ L_{0.5}^*(w) &= \begin{cases} |w|^{0.5} & , |w| \geqslant a \\ \left(-\frac{w^4}{8a^3} + \frac{3w^2}{4a} + \frac{3a}{8}\right)^{0.5} & , |w| < a \end{cases} \end{aligned} \tag{4}$$

where we set $a = 0.01$. In the first phase, $L_{0.5}^*$ is turned off ($\lambda = 0$) to allow the model to fully learn the regression and establish a solid starting point for the model weights. In the second phase, $L_{0.5}^*$ is activated ($\lambda > 0$) to reduce the magnitudes of the weights, promoting the emergence of a sparse model structure. After this, weights with small magnitudes are set to zero and subsequently frozen. In the final phase, $L_{0.5}^*$ is turned off again ($\lambda = 0$) during the fine-tuning of the sparse model. In the experiments that follow, we configure the baseline with network sizes and operator choices similar to those used for `SymbolNet`. We vary λ from 10^{-4} to 10^{-1} and adjust the hard pruning threshold from 10^{-4} to 10^{-1} to conduct a complexity scan by running multiple trials.

Additionally, we use the GP-based `PySR` as another baseline for comparison. In general, the efficiency and performance of GP-based SR algorithms tend to degrade as the number of input features increases, typically beyond ten. To address this, `PySR` incorporates an external feature selector based on a gradient-boosting tree algorithm to identify important features before feeding into the main search loop, particularly for datasets with high input dimensionality [6]. For the LHC dataset, which has 16 input features, we configures `PySR` to select only half of features before performing equation searches. However, for MNIST (784 features) and SVHN (3072 features), which feature counts exceeding the LHC dataset by more than an order of magnitude, reducing the number of features to fewer than 10 makes the models less accurate, while exceeding 10 makes GP-based searches computationally impractical, especially since the outputs are multidimensional as well. Therefore, we restrict our comparison with `PySR` to the LHC dataset only.

4.3. Datasets and experiments

We test our framework on datasets with input dimensions ranging from $\mathcal{O}(10)$ to $\mathcal{O}(1000)$: the LHC jet tagging dataset with 16 inputs [48], MNIST with 784 inputs [49], and SVHN with 3072 inputs [50]. Datasets are split into train, validation, and test sets with ratios of 0.6/0.2/0.2. Models are trained for 200 epochs with

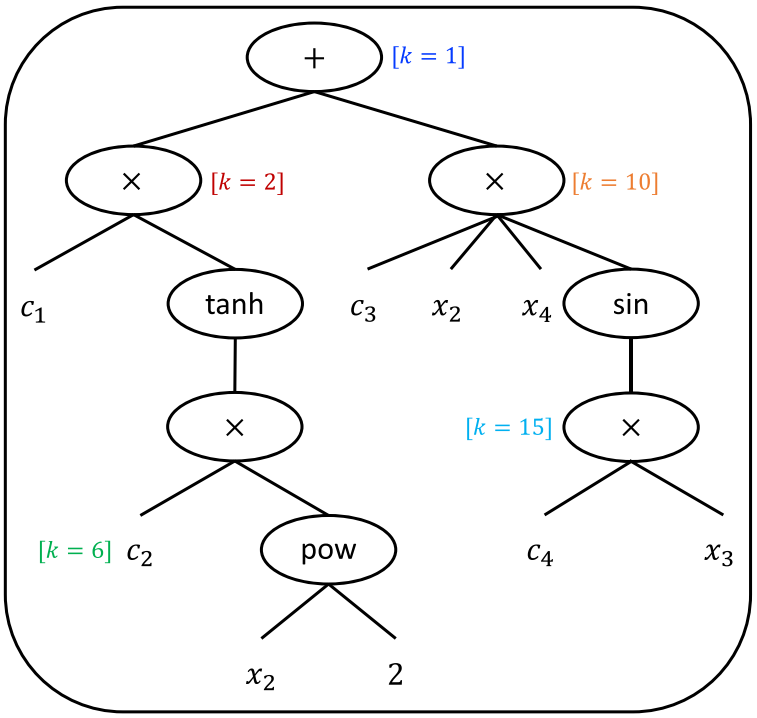

Figure 5. Counting the complexity of an expression in its tree representation involves tree traversal. Using the example expression $y = c_1 \tanh(c_2 x_2^2) + c_3 x_2 x_4 \sin(c_4 x_3)$ from figure 2, the sub-expressions at each step of the traversal (k) are listed. The expression complexity of this example is 17, assuming all mathematical operators, input features, and constants are equally weighted. Note that the number of possible traversal steps corresponds to the number of nodes in the tree.

a batch size of 1024 using the `Adam` optimizer [51] and a learning rate of 0.0015. As described in section 3.2, trainable thresholds for all pruning types are initialized to zero and are clipped to the range [0,1] when competing with the corresponding fixed threshold target of 1, except for thresholds associated with model weight pruning, which are clipped to $[0,\infty]$ since the model weights are unbounded. Model weights and biases are initialized with random normal distributions. Typically, one to two symbolic hidden layers with $\mathcal{O}(1-10)$ unary and binary operators are sufficient for most cases. There is flexibility in choosing differentiable operators for activations, with elementary functions like trigonometric and exponential functions being adequate due to the extensive function space the network can represent.

4.3.1. LHC jet tagging

We chose a dataset from the field of high-energy physics due to the increasing demands for efficient machine learning solutions in resource-constrained environments, such as the LHC experiments [52, 53].

In collider experiments, a jet refers to a cone-shaped object containing a flow of particles, which can be traced back to the decay of an unstable particle. The process of determining the original particle from the characteristics of the jet is known as jet tagging. The LHC jet tagging dataset consists of simulated jets produced from proton-proton collisions at the LHC and is designed to benchmark a five-class classification task: identifying a jet as originating from a light quark, gluon, W boson, Z boson, or top quark. This classification is based on 16 physics-motivated input features constructed from detector observables, including $\left(\sum z \log z,\ C_1^{\beta=0,1,2},\ C_2^{\beta=1,2},\ D_2^{\beta=1,2},\ D_2^{(\alpha,\beta)=(1,1),(1,2)},\ M_2^{\beta=1,2},\ N_2^{\beta=1,2},\ m_{\text{mMDT}},\ \text{Multiplicity}\right)$. The dataset is publicly available at [48], and further descriptions can be found in [54–56].

The inputs are standardized and their distributions are shown in figure 6(a). The labels are one-hot encoded. We train `SymbolNet` with five output nodes, each generating an expression corresponding to one of the five jet substructure classes. We consider models with one or two hidden symbolic layers, using unary operators including $\sin(\cdot)$, $\cos(\cdot)$, $\exp(\cdot)$, $\exp(-(\cdot)^2)$, $\sinh(\cdot)$, $\cosh(\cdot)$, and $\tanh(\cdot)$, and binary operators including $+$ and $\times$. To explore a range of expression complexities, we conduct 40 trials for a grid search of the hyperparameters, varying the number of operators within the ranges $u \in [2,30]$ for unary operators and $b \in [2,30]$ for binary operators. We also vary the target sparsity ratios within the ranges $\alpha_{\text{weight}} \in [0.6, 0.99]$, $\alpha_{\text{input}} \in [0.4, 0.9]$, $\alpha_{\text{unary}} \in [0.2, 0.5]$, and $\alpha_{\text{binary}} \in [0.2, 0.5]$.

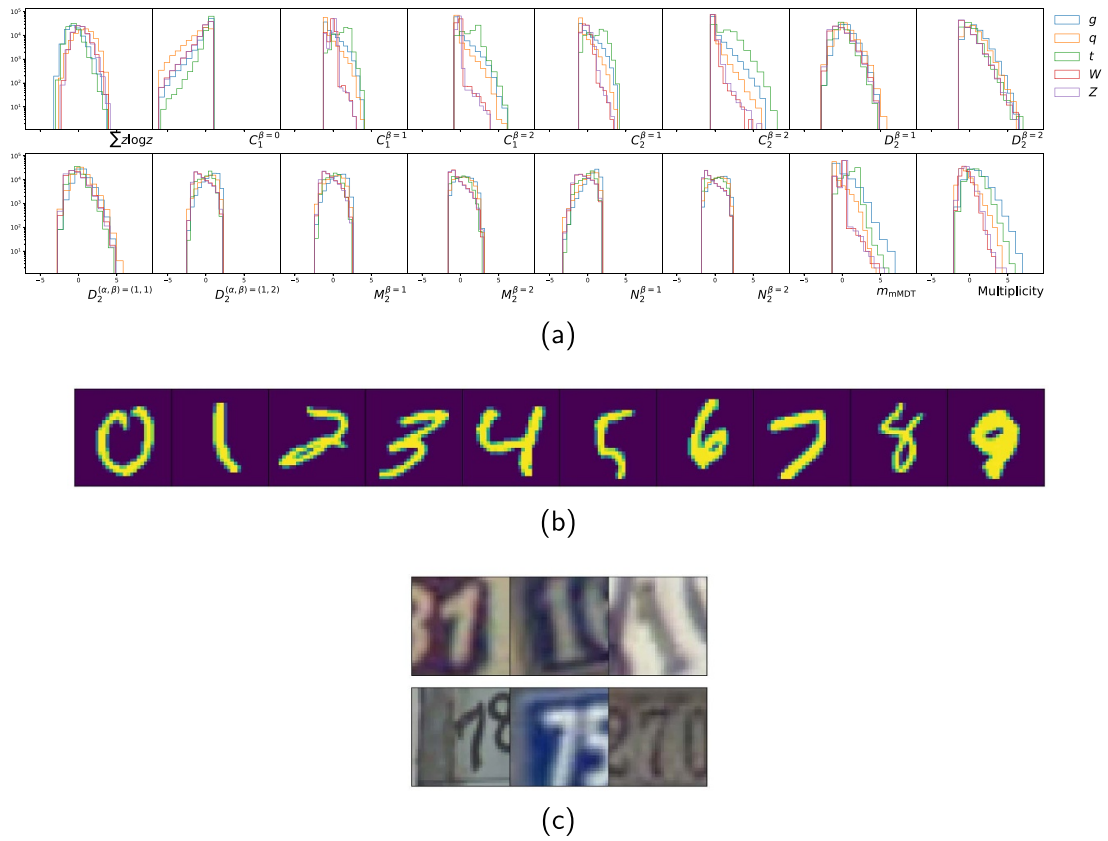

Figure 6. (a): Distributions of the 16 standardized input features in the LHC jet tagging dataset. From top left to top right: $\sum z \log z$, $C_1^{\beta=0}$, $C_1^{\beta=1}$, $C_1^{\beta=2}$, $C_2^{\beta=1}$, $C_2^{\beta=2}$, $D_2^{\beta=1}$, and $D_2^{\beta=2}$. From bottom left to bottom right: $D_2^{(\alpha,\beta)=(1,1)}$, $D_2^{(\alpha,\beta)=(1,2)}$, $M_2^{\beta=1}$, $M_2^{\beta=2}$, $N_2^{\beta=1}$, $N_2^{\beta=2}$, m_{mMDT}, and Multiplicity. The five jet classes are plotted separately: gluon (blue), light quark (orange), top quark (green), W boson (red), and Z boson (purple). (b): MNIST input images with one example per class. (c): SVHN input images with three examples from the digit '1' class (top) and the digit '7' class (bottom), respectively.

4.3.2. MNIST and binary SVHN

We aim to demonstrate that `SymbolNet` can handle datasets with high input dimensions, which most existing SR methods cannot process efficiently. To illustrate this, we examine the MNIST and SVHN datasets. Our objective is not to create an exhaustive classifier with state-of-the-art accuracy, but rather to show `SymbolNet`'s capability to generate simple expressions that can fit high-dimensional data with reasonable accuracy.

The MNIST dataset consists of grayscale images of handwritten digits ranging from '0' to '9', each with an input dimension of $28 \times 28 \times 1$. The task is to classify the correct digit in a given image. Figure 6(b) shows an example input image for each of the ten classes. The inputs are flattened into a 1D array, denoted $x_0, \ldots, x_{783}$, and scaled to the range of $[0, 1]$. We train `SymbolNet` with ten output nodes, each generating an expression corresponding to one of the ten digit classes. We consider models with one or two hidden symbolic layers, using unary operators including $\sin(\cdot)$, $\cos(\cdot)$, $\exp(-(\cdot)^2)$, and $\tanh(\cdot)$, and binary operators including $+$ and $\times$. To explore a range of expression complexities, we conduct 40 trials, for a grid search of hyperparameters, by varying the number of operators within the ranges $u \in [2, 20]$ for unary operators and $b \in [2, 20]$ for binary operators. We also vary the target sparsity ratios within the ranges $\alpha_{\text{weight}} \in [0.7, 0.999]$, $\alpha_{\text{input}} \in [0.6, 0.99]$, $\alpha_{\text{unary}} \in [0.2, 0.5]$, and $\alpha_{\text{binary}} \in [0.2, 0.5]$.

Similar to MNIST but more challenging, the SVHN dataset consists of digit images with an input dimension of $32 \times 32 \times 3$ in RGB format, where the digits are taken from noisy real-world scenes. These images often include various distractors alongside the digit of interest, making the 10-class classification particularly challenging without additional architectural modifications, such as convolutional layers or other techniques to handle the complexity. Therefore, for simplicity, we focus on binary classification between the digits '1' and '7'. Figure 6(c) shows some example input images for each of these two classes. The inputs are flattened into a 1D array, denoted $x_0, \ldots, x_{3071}$,and scaled to the range of $[0, 1]$. In this binary setting, we label

the digit '1' as 0 and the digit '7' as 1, so `SymbolNet` has one output node and generates one expression per model. We consider a single hidden symbolic layer, with unary operators including $\sin(\cdot)$, $\cos(\cdot)$, $\exp(-(\cdot)^2)$, and $\tanh(\cdot)$, and binary operators including $+$ and $\times$. To explore a range of expression complexities, we conduct 40 trials, for a grid search of hyperparameters, by varying the number of operators within the ranges $u \in [2, 20]$ for unary operatos and $b \in [2, 10]$ for binary operators. We also vary the target sparsity ratios within the ranges $\alpha_{\text{weight}} \in [0.8, 0.999]$, $\alpha_{\text{input}} \in [0.8, 0.999]$, $\alpha_{\text{unary}} \in [0.2, 0.5]$, and $\alpha_{\text{binary}} \in [0.2, 0.5]$.

4.4. FPGA resource utilization and latency

It has been demonstrated in [3] that symbolic models can potentially reduce FPGA resource consumption by orders of magnitude and achieve significantly lower latency compared to quantized yet unpruned NNs when applied to the LHC jet tagging dataset. In this study, we perform a similar comparison, but this time between symbolic models and NNs that are strongly quantized [38, 57, 58] and pruned [36–38] as a typical compression strategy.

For the LHC jet tagging dataset, the baseline architecture is adopted from [54], which is a fully-connected NN, or DNN, with three hidden layers, consisting of 64, 32, and 32 neurons, respectively. For the MNIST and SVHN datasets, the baseline architecture is adopted from [59], which is a convolutional NN [60, 61], or CNN, consisting of three convolutional blocks with 16, 16, and 24 filters of size 3×3, respectively, followed by a DNN with two hidden layers consisting of 42 and 64 neurons, respectively. These baseline architectures were selected with consideration that the models are small enough to fit within the resource budget of a single FPGA board.

The baseline NNs are further compressed through quantization and pruning. The models are trained using quantization-aware techniques with the `QKeras` library[6] [57], and pruning is performed using the `Tensorflow` pruning API [62]. We quantize model weights and activation functions in all hidden layers to a fixed total bit width of 6, with no integer bit (denoted as $\langle 6, 0 \rangle$). The model weights are pruned to achieve a sparsity ratio of approximately 90%.

Both the symbolic models and the NNs are converted to FPGA firmware using the `hls4ml` library[7] [3, 54, 63]. Synthesis is performed with Vivado HLS (2020.1) [64], targeting a Xilinx Virtex UltraScale+ VU9P FPGA (part no.: xcvu9p-flga2577-2-e), with the clock frequency set to 200 MHz (or clock period of 5 ns). We compare FPGA resource utilization and inference latency between symbolic models learned by `SymbolNet` and the compressed NNs.

5. Results

5.1. LHC jet tagging

To demonstrate the effectiveness of our adaptive dynamic pruning framework, we conducted a trial using `SymbolNet` with two symbolic layers, each containing $u = v = 20$ operators, and set the target sparsity ratios for model weights, inputs, unary operators, and binary operators to $\alpha_{\text{weight}} = 0.9$, $\alpha_{\text{input}} = 0.75$, $\alpha_{\text{unary}} = 0.6$, and $\alpha_{\text{binary}} = 0.4$, respectively. The training curves for this trial are shown in figure 7.

As seen in figure 7(a), the training loss (L_{MSE}) and the other sparsity terms (L_{sparse}) steadily decrease until around epoch 50. At this point, the trainable thresholds for the input, unary operator, and binary operator all approach 1 on average, as shown in figure 7(b). Consequently, many nodes begin to be pruned as their threshold values reach 1. This pruning is reflected in figure 7(c), where the sparsity ratios increase sharply, with the ratios for unary and binary operators reaching their target values, causing their losses to drop steeply toward zero.

However, the sparsity ratios for both model weights and inputs remain below their target values, so their losses do not yet reach zero. A small kink is observed in L_{MSE} (or accuracy) around the same epoch, caused by the steep increase in sparsity ratios. Despite this, the training process adjusts dynamically, and the training loss continues to decrease as the sparsity ratios steadily increase. The sparsity ratio for model weights reaches its target value around epoch 100, resulting in a steep drop in its loss at that point. By the end of the training, the input sparsity ratio reaches around 70%, slightly below its target of 75%, so its loss does not fully vanish. Throughout the training, the sparsity ratios converge toward their target values, with regression and sparsity being optimized simultaneously in this dynamic process.

Figure 8(a) illustrates the trade-off between accuracy and model complexity, demonstrating that `SymbolNet` outperforms `EQL` in the multistage pruning framework and is comparable to the computationally intensive GP-based algorithm `PySR`.

[6] https://github.com/google/qkeras
[7] https://github.com/fastmachinelearning/hls4ml

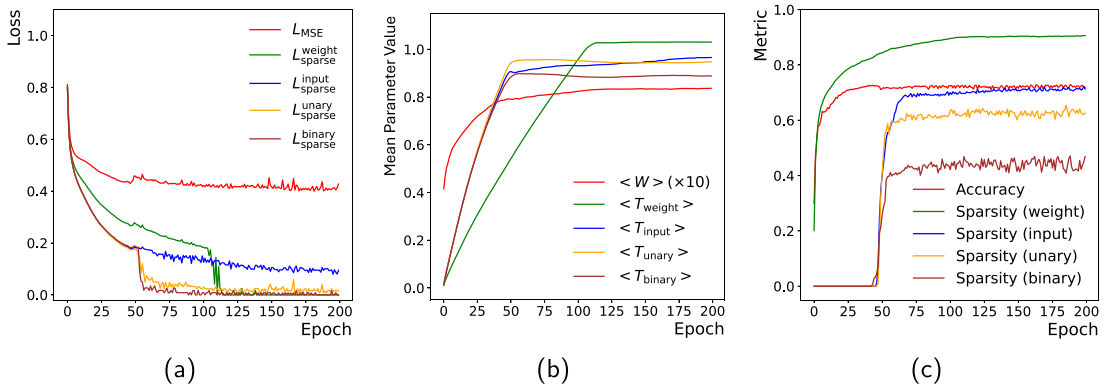

Figure 7. We demonstrate the training performance of `SymbolNet` on the LHC jet tagging dataset. This `SymbolNet` model consists of two symbolic layers, each containing $u = v = 20$ operations. It is trained with a batch size of 1024 for 200 epochs. The `Adam` optimizer [51] is used with a learning rate of 0.0015. The target sparsity ratios are set to $\alpha_{\text{weight}} = 0.9$, $\alpha_{\text{input}} = 0.75$, $\alpha_{\text{unary}} = 0.6$, and $\alpha_{\text{binary}} = 0.4$. (a): The individual loss terms in equation (3) as functions of the epoch. (b): The mean trainable parameters as functions of the epoch. (c): The accuracy and sparsity ratios as functions of the epoch.

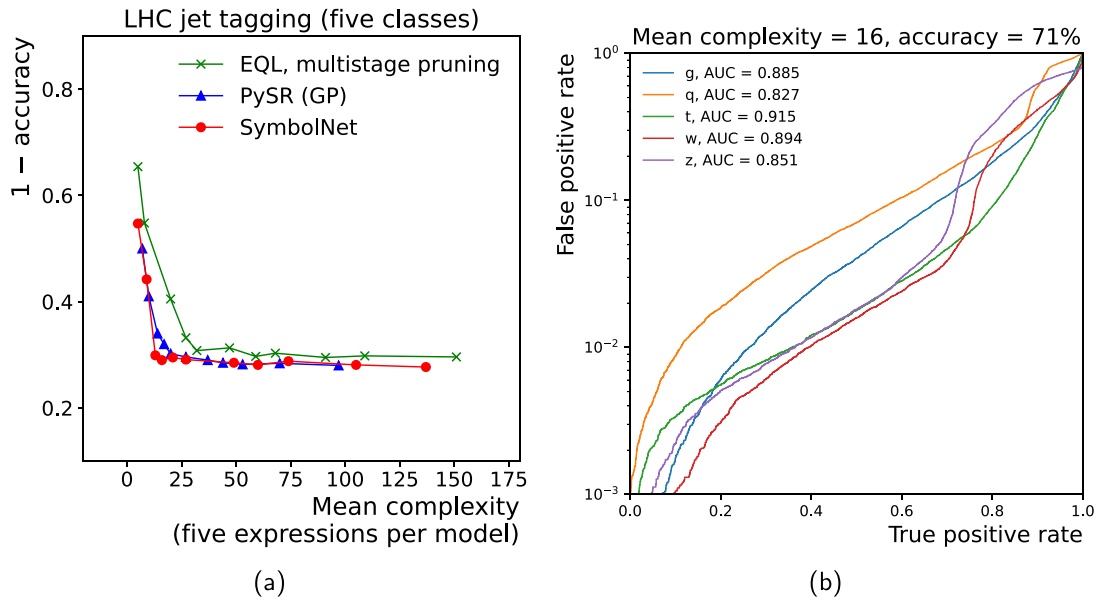

Figure 8. Performance of `SymbolNet` on the LHC jet tagging dataset. (a): The Pareto front, each with models selected from the 40 trials, illustrates the trade-off between accuracy and expression complexity. (b): ROC curves for a compact symbolic model with a mean complexity of 16 and an overall accuracy of 71%, with its expressions tabulated in table 1.

Table 1. An example of a compact symbolic model with **a mean complexity of 16 and an overall accuracy of 71%** learned by `SymbolNet` on the LHC jet tagging dataset. Constants are rounded to 2 significant figures for the purpose of display.

Class	Expression (symbolic model for LHC jet tagging)	Complexity	AUC
g	$-0.041\text{Multiplicity} \times C_1^{\beta=1} + 0.53\textbf{tanh}(0.6\text{Multiplicity} - 0.38C_1^{\beta=1}) + 0.24$	16	0.885
q	$0.073\text{Multiplicity} \times C_1^{\beta=1} - 0.38\textbf{tanh}(0.63m_{\text{mMDT}}) + 0.15$	12	0.827
t	$0.2\textbf{sin}(1.2\text{Multiplicity}) + 0.43\textbf{sin}(0.49C_1^{\beta=2}) - 0.2\textbf{tanh}(0.6\text{Multiplicity} - 0.38C_1^{\beta=1}) + 0.24$	24	0.915
W	$-0.099\textbf{sin}(0.73\text{Multiplicity}) + 0.84\textbf{exp}(-46.0(m_{\text{mMDT}} + 0.14C_1^{\beta=1} + 0.27C_1^{\beta=2})^2) + 0.044$	23	0.894
Z	$0.43\textbf{exp}(-6.9(C_1^{\beta=2})^2)$	8	0.851

Table 1 presents an example model learned by `SymbolNet`. This symbolic model is remarkably compact, consisting of only five lines of expressions, with an average complexity of 16, achieving an overall jet-tagging accuracy of 71%. Figure 8(b) shows the ROC curve for each of these five expressions.

For comparison, a black-box three-layer NN with $O(10^3)$ parameters achieves an accuracy of 75%. However, such a model would require orders of magnitude more resources to compute on an FPGA than a symbolic model with a similar complexity, as studied in [3].

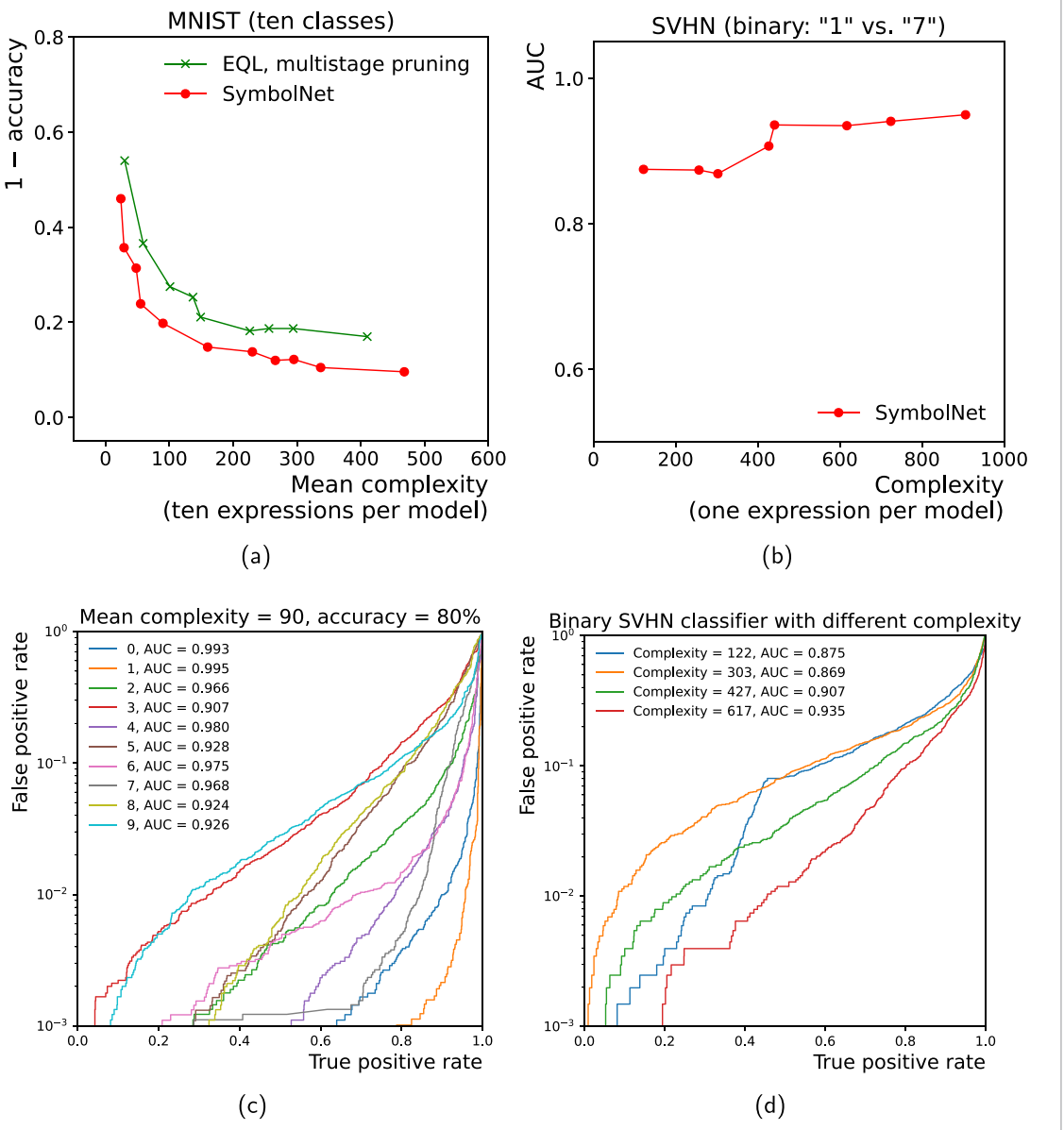

Figure 9. Performance of `SymbolNet` on the MNIST and binary SVHN datasets. (a): The Pareto front, each with models selected from the 40 trials, illustrates the trade-off between accuracy and expression complexity, comparing `SymbolNet` with `EQL` on the MNIST dataset. (b): The ROC AUC, with models selected from 40 trials, is plotted against expression complexity to demonstrate the performance of `SymbolNet` on the binary SVHN dataset. (c): ROC curves for a compact symbolic model with a mean complexity of 90 and an accuracy of 80% on the MNIST dataset (the expressions of this model are listed in table 2). (d): ROC curves for four symbolic models with different complexity values on the binary SVHN dataset (the expression for the model with a complexity of 122 is listed in table 3).

5.2. MNIST and binary SVHN

We analyze the MNIST dataset by considering all ten classes, with each symbolic model generating ten expressions, each corresponding to one of the ten classes. Figure 9(a) shows a Pareto front generated by `SymbolNet` on the MNIST dataset, outperforming `EQL` in the multistage pruning framework. For instance, `SymbolNet` can achieve an overall accuracy of 90% with a mean complexity of around 300. Table 2 presents the ten expressions of a symbolic model learned by `SymbolNet`, which has a mean complexity of 90 and an overall accuracy of 80%. The ROC curves for these expressions are shown in figure 9(c). This example demonstrates the power of symbolic models learned by `SymbolNet`—compact enough to be fully visualized within a table yet capable of making reasonable predictions.

For the SVHN dataset in the binary setting, each symbolic model generates a single expression. We observed that `EQL` struggled to converge on this dataset, producing either overly complex expressions with reasonable accuracy or models that were too sparse to make meaningful predictions. In contrast, `SymbolNet` was able to scale to this high-dimensional dataset and generated compact expressions with reasonable predictive accuracy. Figure 9(b) shows the ROC AUC as a function of expression complexity for models

Table 2. An example of a compact symbolic model with **a mean complexity of 90 and an overall accuracy of 80%**, learned by `SymbolNet` on the MNIST dataset. Constants are rounded to 2 significant figures for the purpose of display.

Class	Expression (symbolic model classifying MNIST digits)	Complexity	AUC
0	$0.094\mathbf{sin}(0.41x_{374} - 0.53x_{378} + 0.66x_{484}) + 0.15(-0.3x_{184} - 0.17x_{239} - 0.12x_{269} - 0.27x_{271} - 0.14x_{318} + 0.72x_{352} - 0.6x_{358} - 0.19x_{374} + 0.55x_{377} - 0.32x_{415} - 0.23x_{456} - 0.26x_{485} - 0.4x_{510} - 0.53x_{627} + 0.25x_{637} - 0.19x_{658} + 0.55x_{711}) \times (0.44x_{102} - 0.29x_{156} - 0.41x_{212} - 0.29x_{271} - 0.22x_{302} - 0.11x_{371} - 0.5x_{398} - 0.41x_{428} - 0.24x_{430} + 0.84x_{433} + 0.6x_{436} + 0.11x_{462} + 0.62x_{490} - 0.45x_{509} - 0.066x_{539} - 0.4x_{541} - 0.13x_{568} + 0.22x_{580} - 0.58x_{627} - 0.25x_{658})$	129	0.993
1	$\mathbf{exp}(-26.0(0.11x_{102} + 0.056x_{158} + 0.21x_{176} + 0.08x_{178} + 0.093x_{182} + 0.93x_{205} + 0.11x_{212} + 0.15x_{235} + 0.27x_{248} - 0.033x_{267} + 0.067x_{271} + 0.24x_{302} - 0.063x_{323} + 0.095x_{327} - 0.067x_{350} - 0.12x_{378} + 0.18x_{430} + x_{438} - 0.067x_{462} - 0.092x_{489} + 0.18x_{510} - 0.024x_{568} + 0.12x_{580} + 0.23x_{637} + 0.24x_{711} + 0.13x_{713} + 0.24x_{715} + 0.27x_{96} + 0.28)^2)$	91	0.995
2	$0.54\mathbf{sin}(0.59x_{124} + 0.35x_{156} - 0.39x_{318} - 0.41x_{350} - 0.46x_{371} - 0.41x_{374} - 0.6x_{415} + 0.18x_{431} + 0.14x_{465} + 1.1x_{473} + 0.7x_{509} + 0.38x_{515} + 0.88x_{528} + 0.38x_{554} + 0.77x_{611} + 0.39x_{637} + 0.1x_{99} - 0.8) + 0.53$	58	0.966
3	$-0.042x_{158} + 0.062x_{178} - 0.039x_{235} - 0.12x_{291} - 0.063x_{316} + 0.045x_{318} + 0.061x_{404} - 0.066x_{458} + 0.032x_{485} - 0.1x_{487} - 0.074x_{489} - 0.12x_{490} + 0.038x_{515} + 0.036x_{517} - 0.06x_{541} + 0.36x_{563} - 0.043x_{572} + 0.048x_{611} + 0.28$	56	0.907
4	$0.76\mathbf{exp}(-4.7(0.47x_{124} + 0.42x_{126} + 0.49x_{128} + 0.14x_{176} + 0.28x_{182} + 0.44x_{184} + 0.17x_{212} + x_{239} + 0.88x_{267} + 0.81x_{322} + 0.43x_{323} + 0.33x_{350} + 0.4x_{543} + 0.3x_{554} + 0.5x_{568} + 0.35x_{623})^2) - 0.082(-0.2x_{124} - 0.34x_{182} + 0.39x_{429} - 0.69x_{568} - 0.66x_{713} + 0.68) \times (1.4x_{102} + 0.58x_{182} + 0.75x_{208} + 0.51x_{215} + 0.29x_{235} + 0.47x_{322} - 0.53x_{323} - 0.7x_{325} + 0.23x_{355} + 0.53x_{358} - 1.4x_{374} - 1.5x_{398} - 0.63x_{431} - 1.5x_{456} - 0.68x_{462} - 1.1x_{465} + 0.48x_{541} + 0.83x_{568} + 4.9x_{66} + 1.3x_{71} + 1.3x_{713} + 1.4x_{96})$	141	0.980
5	$\mathbf{exp}(-2.4(-0.15x_{124} + 0.13x_{158} + 0.59x_{190} + 0.98x_{248} - 0.13x_{267} - 0.35x_{323} - 0.68x_{325} - x_{327} - 0.78x_{355} + 0.17x_{404} - 0.5x_{456} - 0.19x_{490} - 0.41x_{510} - 0.6x_{515} + 0.15x_{568} - 0.63)^2) - 0.012x_{128} - 0.12x_{358} + 0.03x_{371} + 0.069x_{374} - 0.031x_{436} - 0.019x_{485} + 0.042x_{580} + 0.026x_{623}$	79	0.928
6	$0.21x_{102} + 0.3x_{103} + 0.42x_{107} - 0.054x_{215} - 0.057x_{217} - 0.093x_{269} - 0.065x_{271} - 0.068x_{302} - 0.08x_{322} + 0.068x_{358} + 0.04x_{374} + 0.12x_{414} + 0.021x_{431} + 0.069x_{485} - 0.063x_{489} - 0.078x_{510} + 0.081x_{515} + 0.047x_{543} - 0.056x_{568} + 0.065x_{572} - 0.05x_{580} + 0.35x_{64} + 0.43x_{66} + 0.22x_{68} + 0.34x_{69} + 0.29x_{71} + 0.35x_{73} + 0.56x_{78} + 0.18x_{99} + 0.1$	89	0.975
7	$0.98\mathbf{exp}(-3.1(-x_{124} - 0.61x_{126} - 0.81x_{128} - 0.97x_{156} - 0.24x_{184} - 0.23x_{350} + 0.073x_{355} - 0.28x_{376} - 0.13x_{377} - 0.62x_{378} - 0.72x_{404} - 0.6x_{415} - 0.62x_{431} - 0.092x_{433} - 0.43x_{458} - 0.87x_{485} - 0.94x_{539} - 0.27x_{541} - 0.84x_{581} - 0.37x_{623})^2)$	68	0.968
8	$-0.68\mathbf{sin}(0.16x_{156} - 0.35x_{176} + 0.43x_{302} + 0.19x_{318} + 0.23x_{327} + 0.41x_{376} - 0.2x_{414} - 0.4x_{428} + 0.46x_{433} - 0.33x_{467} + 0.27x_{487} + 0.3x_{515} - 0.34x_{528} + 0.25x_{541} + 0.58x_{658} + 0.43x_{689} + 1.1) + 0.64$	55	0.924
9	$-0.051\mathbf{sin}(0.59x_{124} + 0.35x_{156} - 0.39x_{318} - 0.41x_{350} - 0.46x_{371} - 0.41x_{374} - 0.6x_{415} + 0.18x_{431} + 0.14x_{465} + 1.1x_{473} + 0.7x_{509} + 0.38x_{515} + 0.88x_{528} + 0.38x_{554} + 0.77x_{611} + 0.39x_{637} + 0.1x_{99} - 0.8) - 0.054x_{126} - 0.066x_{158} - 0.082x_{190} - 0.11x_{205} + 0.059x_{208} + 0.016x_{215} - 0.039x_{217} - 0.0092x_{235} - 0.11x_{248} - 0.047x_{271} + 0.093x_{316} - 0.04x_{322} + 0.069x_{327} + 0.07x_{352} - 0.059x_{414} + 0.069x_{429} + 0.038x_{431} + 0.048x_{436} - 0.057x_{467} - 0.044x_{517} - 0.067x_{541} + 0.065x_{637} - 0.06x_{658} + 0.11x_{711} + 0.1x_{713} + 0.16x_{715} + 0.029$	136	0.926

Table 3. An example of a compact symbolic model with **a complexity of 122 and an ROC AUC of 0.875**, learned by `SymbolNet` on the binary SVHN dataset (classes '1' and '7'). Constants are rounded to 2 significant figures for the purpose of display.

Expression (symbolic model classifying SVHN digits in a binary setting: '1' vs. '7')	Complexity	AUC
$0.58\mathbf{exp}(-4.7(0.35x_{1191}+0.2x_{1282}+0.29x_{1285}+0.53x_{1384}+0.3x_{1566}+0.35x_{1788}-0.56x_{2156}+0.38x_{2179}+0.51x_{2460}+0.22x_{2470}+0.33x_{2746}-0.6x_{429}-0.26x_{612}-0.45x_{628}-0.32x_{637}-0.33x_{732}-0.26x_{733}-0.28x_{813}-0.15x_{913}-1)^2)+0.61\mathbf{cos}(1.4x_{1282}-1.4x_{1298}-1.6x_{1486}+1.7x_{1863}-1.2x_{2357}-0.53x_{2460}+0.79x_{2609}+0.79x_{3046}-0.94x_{485}+0.71x_{511}-1.4x_{527}+1.9x_{617}+0.76x_{637}-1.8x_{720}+1.8x_{824}+0.68x_{831}-0.99x_{913})$	122	0.875

learned by `SymbolNet`, while figure 9(d) shows the ROC curves for four symbolic models with different complexity values. Table 3 lists the expression of a symbolic model with a complexity of 122 and an ROC AUC of 0.875. Remarkably, even such a single-line expression can provide decent predictions in classifying SVHN digits from noisy real-world scenes, as depicted in figure 6(c).

5.3. FPGA resource utilization and latency

Table 4 presents a comparison of resource utilization and latency on an FPGA between symbolic models and NNs with typical compression techniques. The FPGA resources considered include on-board FPGA memory (BRAMs), digital signal processors (DSPs), flip-flops (FFs), and lookup tables (LUTs), as estimated from the logic synthesis step. Symbolic models generally consume significantly fewer resources and require much less latency than NNs, even when the NNs are strongly QP, while still achieving comparable accuracy across the three datasets.

6. Discussion

We have introduced `SymbolNet`, a NN-based approach to SR for model compression, featuring a novel pruning framework designed to generate compact expressions capable of fitting high-dimensional data. Our method aims to address the latency and bandwidth bottlenecks encountered in modern high-energy physics experiments, such as those at the CERN LHC, where datasets are typically high-dimensional and do not have ground-truth equations. By leveraging the efficiency of compact symbolic representations, which can be implemented on custom hardware such as FPGAs, we further extend the applicability of SR to higher-dimensional data, as commonly found in LHC experiments.

Most existing SR methods, whether based on GP or DL, primarily focus on datasets with input dimensions less than $\mathcal{O}(10)$ and cannot scale efficiently beyond. In GP approaches, the search algorithms create and evolve equations using a combinatorial strategy, which becomes highly inefficient as the equation search space scales exponentially with its building blocks. Even with an external feature selector, the GP-based algorithms can still be impractical for datasets with input dimensions beyond $\mathcal{O}(100)$. On the other hand, while DL techniques have demonstrated their ability to handle complex datasets in other domains, they have not been extensively explored for SR in high-dimensional problems.

Our proposed method, equipped with a novel pruning framework dedicated to SR, aims to fill this gap. Its effectiveness has been demonstrated on datasets with input dimensions ranging from $\mathcal{O}(10)$ to $\mathcal{O}(1000)$. While there is always a trade-off between prediction accuracy and computational resources, this work provides an option to minimize the latter. Our method is specifically designed to be a model compression technique by leveraging the compact representation of symbolic expressions for low-latency deployment in resource-constrained environments.

However, NNs currently have certain limitations when used for SR compared to GP. For example, it is challenging to create a unified framework that can incorporate mathematical operators that are not differentiable everywhere, such as logarithms and division, without specifically regularizing each of these operators. While these operators could expand model expressivity, they can introduce singular points in gradient-based optimization. Potential future extensions include tackling time series or sequence data, integrating domain-specific constraints or prior knowledge, and developing methods to improve the interpretability of generated expressions from high-dimensional datasets. To fully maximize the potential of efficient hardware deployment of symbolic expressions, further work is needed to optimize the expressions while maintaining numerical precision within fixed bit widths, a process known as quantization-aware training [57]. These topics will be addressed in future work.

Table 4. Resource utilization and latency on an FPGA for quantized and pruned (QP) NNs and symbolic expressions learned by `SymbolNet`. The model size is expressed in terms of the number of neurons per hidden layer for DNNs and the number of filters for CNNs, where, for example, $(16)_3$ indicates 16 filters with a kernel size of 3×3. The initiation interval (II) is quoted in clock cycles. The numbers in parentheses indicate the percentage of total available resource utilization. The relative accuracy and ROC AUC are evaluated with respect to the same DNN/CNN implemented in floating-point precision and without pruning.

LHC jet tagging (five classes)									
	Model size (input dim. = 16)	Precision	BRAMs	DSPs	FFs	LUTs	II	Latency	Rel. acc.
QP DNN	[64, 32, 32, 5], **90% pruned**	⟨6, 0⟩	4 (0.1%)	28 (0.4%)	2739 (0.1%)	7691 (0.7%)	1	55 ns	94.7%
SR	**Mean complexity of the five expr. = 18**	⟨12, 8⟩	0 (0%)	3 (0%)	109 (0%)	177 (0%)	1	**10 ns**	93.3%
MNIST (ten classes)									
	Model size (input dim. = 28×28×1)	Precision	BRAMs	DSPs	FFs	LUTs	II	Latency	Rel. acc.
QP CNN	$[(16, 16, 24)_3, 42, 64, 10]$, **92% pruned**	⟨6, 0⟩	66 (1.5%)	216 (3.2%)	18 379 (0.8%)	29 417 (2.5%)	788	4.0 μs	86.8%
SR	**Mean complexity of the ten expr. = 133**	⟨18, 10⟩	0 (0%)	160 (2.3%)	6424 (0.3%)	7592 (0.6%)	1	**125 ns**	85.3%
SVHN (binary '1' vs. '7')									
	Model size (input dim. = 32×32×3)	Precision	BRAMs	DSPs	FFs	LUTs	II	Latency	Rel. AUC
QP CNN	$[(16, 16, 24)_3, 42, 64, 1]$, **92% pruned**	⟨6, 0⟩	62 (1.4%)	77 (1.1%)	16 286 (0.7%)	27 407 (2.3%)	1029	5.2 μs	94.0%
SR	**Complexity = 311**	⟨10, 4⟩	0 (0%)	38 (0.6%)	1945 (0.1%)	3029 (0.3%)	1	**195 ns**	94.5%

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://doi.org/10.5281/zenodo.3602260, http://yann.lecun.com/exdb/mnist/ and http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

Acknowledgments

H F T and S D are supported by the U S Department of Energy under the Contract DE-SC0017647. V L and P H are supported by the NSF Institute for Accelerated AI Algorithms for Data-Driven Discovery (A3D3), under the NSF Grant #PHY-2117997. P H is also supported by the Institute for Artificial Intelligence and Fundamental Interactions (IAIFI), under the NSF Grant #PHY-2019786.

ORCID iDs

Ho Fung Tsoi https://orcid.org/0000-0002-2550-2184
Vladimir Loncar https://orcid.org/0000-0003-3651-0232
Sridhara Dasu https://orcid.org/0000-0001-5993-9045
Philip Harris https://orcid.org/0000-0001-8189-3741

References

[1] Cava W L *et al* 2021 Contemporary symbolic regression methods and their relative performance *Thirty-fifth Conf. on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)* (arXiv:2107.14351)
[2] Planck M 1900 On an improvement of Wien's equation for the spectrum *Verh. Dtsch. Phys. Ges.* **2** 202
[3] Tsoi H F *et al* 2024 Symbolic Regression on FPGAs for fast machine learning inference *EPJ Web Conf.* **295** 09036
[4] Virgolin M and Pissis S P 2022 Symbolic regression is NP-hard *Trans. Mach. Learn. Res.* (available at: https://openreview.net/forum?id=LTiaPxqe2e)
[5] Schmidt M and Lipson H 2009 Distilling free-form natural laws from experimental data *Science* **324** 81–85
[6] Cranmer M 2023 Interpretable machine learning for science with PySR and SymbolicRegression.jl (arXiv:2305.01582)
[7] Stephens T 2016 Genetic programming in Python, with a scikit-learn inspired API: gplearn (available at: https://gplearn.readthedocs.io/en/stable/)
[8] Burlacu B, Kronberger G and Kommenda M 2020 Operon C++: an efficient genetic programming framework for symbolic regression *Proc. 2020 Genetic and Evolutionary Computation Conf. Companion, GECCO'20* (Association for Computing Machinery) pp 1562–70
[9] Virgolin M, Alderliesten T, Witteveen C and Bosman P A N 2021 Improving model-based genetic programming for symbolic regression of small expressions *Evol. Comput.* **29** 211–37
[10] Martius G and Lampert C H 2016 Extrapolation and learning equations (arXiv:1610.02995)
[11] Sahoo S S, Lampert C H and Martius G 2018 Learning equations for extrapolation and control (arXiv:1806.07259)
[12] Werner M, Junginger A, Hennig P and Martius G 2021 Informed equation learning (arXiv:2105.06331)
[13] Kim S *et al* 2021 Integration of neural network-based symbolic regression in deep learning for scientific discovery *IEEE Trans. Neural Netw. Learn. Syst.* **32** 4166–77
[14] Abdellaoui I A and Mehrkanoon S 2021 Symbolic regression for scientific discovery: an application to wind speed forecasting (arXiv:2102.10570)
[15] Costa A *et al* 2021 Fast neural models for symbolic regression at scale (arXiv:2007.10784)
[16] Petersen B K *et al* 2021 Deep symbolic regression: recovering mathematical expressions from data via risk-seeking policy gradients *Int. Conf. on Learning Representations* (available at: https://openreview.net/forum?id=m5Qsh0kBQG)
[17] Zhou H and Pan W 2022 Bayesian learning to discover mathematical operations in governing equations of dynamic systems (arXiv:2206.00669)
[18] Kubalík J, Derner E and Babuška R 2023 Toward physically plausible data-driven models: a novel neural network approach to symbolic regression *IEEE Access* **11** 61481–501
[19] Abadi M *et al* 2015 TensorFlow: large-scale machine learning on heterogeneous systems (available at: https://www.tensorflow.org/)
[20] Liu J, Xu Z, SHI R, Cheung R C C and So H K 2020 Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers *Int. Conf. on Learning Representations* (available at: https://openreview.net/forum?id=SJlbGJrtDB)
[21] Udrescu S-M and Tegmark M 2020 Ai feynman: a physics-inspired method for symbolic regression *Sci. Adv.* **6** eaay2631
[22] Keren L S, Liberzon A and Lazebnik T 2023 A computational framework for physics-informed symbolic regression with straightforward integration of domain knowledge *Sci. Rep.* **13** 2
[23] Kaptanoglu A *et al* 2022 Pysindy: a comprehensive python package for robust sparse system identification *J. Open Source Softw.* **7** 3994
[24] Koza J 1994 Genetic programming as a means for programming computers by natural selection *Stat. Comput.* **4** 87–112
[25] Wadekar D, Villaescusa-Navarro F, Ho S and Perreault-Levasseur L 2020 Modeling assembly bias with machine learning and symbolic regression (arXiv:2012.00111)
[26] Shao H *et al* 2022 Finding universal relations in subhalo properties with artificial intelligence *Astrophys. J.* **927** 85
[27] Delgado A M *et al* 2022 Modelling the galaxy–halo connection with machine learning *Mon. Not. R. Astron. Soc.* **515** 2733–46
[28] Wadekar D *et al* 2023 The SZ flux-mass ($Y - M$) relation at low-halo masses: improvements with symbolic regression and strong constraints on baryonic feedback *Mon. Not. R. Astron. Soc.* **522** 2628–43
[29] Lemos P, Jeffrey N, Cranmer M, Ho S and Battaglia P 2023 Rediscovering orbital mechanics with machine learning *Mach. Learn.: Sci. Technol.* **4** 045002

[30] Wadekar D et al 2023 Augmenting astrophysical scaling relations with machine learning: Application to reducing the sunyaev–zeldovich flux–mass scatter Proc. Natl Acad. Sci. 120 20
[31] Grundner A, Beucler T, Gentine P and Eyring V 2023 Data-driven equation discovery of a cloud cover parameterization (arXiv:2304.08063)
[32] McConaghy T 2011 FFX: Fast, Scalable, Deterministic Symbolic Regression Technology (Springer) pp 235–60
[33] Kammerer L, Kronberger G and Kommenda M 2022 Symbolic Regression With Fast Function Extraction and Nonlinear Least Squares Optimization (Springer) pp 139–46
[34] Zeng P et al 2023 Differentiable genetic programming for high-dimensional symbolic regression (arXiv:2304.08915)
[35] LeCun Y, Bengio Y and Hinton G 2015 Deep learning Nature 521 436–44
[36] LeCun Y, Denker J and Solla S 1989 Optimal brain damage Advances in Neural Information Processing Systems vol 2, ed D Touretzky (Morgan-Kaufmann)
[37] Louizos C, Welling M and Kingma D P 2018 Learning sparse neural networks through l_0 regularization Int. Conf. on Learning Representations (available at: https://openreview.net/forum?id=H1Y8hhg0b)
[38] Han S, Mao H and Dally W J 2016 Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding Int. Conf. on Learning Representations (ICLR) (available at: https://arxiv.org/abs/1510.00149)
[39] Biggio L, Bendinelli T, Neitz A, Lucchi A and Parascandolo G 2021 Neural symbolic regression that scales Proc. 38th Int. Conf. on Machine Learning (Proc. of Machine Learning Research vol 139), ed M Meila and T Zhang (PMLR) pp 936–45
[40] Valipour M, You B, Panju M and Ghodsi A 2021 Symbolicgpt: a generative transformer model for symbolic regression (arXiv:2106.14131)
[41] Kamienny P-A, d'Ascoli S, Lample G and Charton F 2022 End-to-end symbolic regression with transformers Advances in Neural Information Processing Systems (available at: https://openreview.net/forum?id=GoOuIrDHG_Y)
[42] Vastl M, Kulhánek J, Kubalík J, Derner E and Babuška R 2022 Symformer: end-to-end symbolic regression using transformer-based architecture (arXiv:2205.15764)
[43] Rosenblatt F 1958 The perceptron: a probabilistic model for information storage and organization in the brain Psychol. Rev. 65 386–408
[44] Cybenko G V 1989 Approximation by superpositions of a sigmoidal function Math. Control Signals Syst. 2 303–14
[45] Hornik K, Stinchcombe M B and White H L 1989 Multilayer feedforward networks are universal approximators Neural Netw. 2 359–66
[46] Rumelhart D E, Hinton G E and Williams R J 1986 Learning representations by back-propagating errors Nature 323 533–6
[47] Meurer A et al 2017 Sympy: symbolic computing in python PeerJ Comput. Sci. 3 e103
[48] Pierini M, Duarte J M, Tran N and Freytsis M 2020 HLS4ML LHC Jet dataset (150 particles) (available at: https://doi.org/10.5281/zenodo.3602260)
[49] LeCun Y and Cortes C 2010 MNIST handwritten digit database (available at: http://yann.lecun.com/exdb/mnist/)
[50] Netzer Y et al 2011 Reading digits in natural images with unsupervised feature learning NIPS Workshop on Deep Learning and Unsupervised Feature Learning vol 2011 p 4 (available at: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf)
[51] Kingma D P and Ba J 2017 Adam: a method for stochastic optimization (arXiv:1412.6980)
[52] ATLAS Collaboration 2017 Technical design report for the phase-II upgrade of the ATLAS TDAQ System (available at: https://doi.org/10.17181/CERN.2LBB.4IAL)
[53] Zabi A, Berryhill J W, Perez E and Tapper A D 2020 The phase-2 Upgrade of the CMS Level-1 Trigger (available at: https://cds.cern.ch/record/2714892)
[54] Duarte J et al 2018 Fast inference of deep neural networks in FPGAs for particle physics JINST 13 07027
[55] Moreno E A et al 2020 JEDI-net: a jet identification algorithm based on interaction networks Eur. Phys. J. C 80 58
[56] Coleman E et al 2018 The importance of calorimetry for highly-boosted jet substructure JINST 13 T01003
[57] Coelho C N et al 2021 Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors Nat. Mach. Intell. 3 675–86
[58] Gupta S, Agrawal A, Gopalakrishnan K and Narayanan P 2015 Deep learning with limited numerical precision Proc. 32nd Int. Conf. on Machine Learning (Proc. Machine Learning Research) vol 37, ed F Bach and D Blei (PMLR) pp 1737–46
[59] Aarrestad T et al 2021 Fast convolutional neural networks on FPGAs with hls4ml Mach. Learn. Sci. Tech. 2 045015
[60] LeCun Y et al 1989 Handwritten digit recognition with a back-propagation network Advances in Neural Information Processing Systems vol 2, ed D Touretzky (Morgan-Kaufmann)
[61] Lecun Y, Bottou L, Bengio Y and Haffner P 1998 Gradient-based learning applied to document recognition Proc. IEEE 86 2278–324
[62] Zhu M and Gupta S 2017 To prune, or not to prune: exploring the efficacy of pruning for model compression (arXiv:1710.01878)
[63] FastML Team 2023 fastmachinelearning/hls4ml (available at: https://doi.org/10.5281/zenodo.1201549)
[64] Xilinx 2020 Vivado design suite user guide: high-level synthesis (available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf)