

Symbolic Regression on FPGAs for Fast Machine Learning Inference

Ho Fung Tsoi^{1*}, *Adrian Alan Pol*², *Vladimir Loncar*^{3,4}, *Ekaterina Govorkova*³, *Miles Cranmer*^{2,5}, *Sridhara Dasu*¹, *Peter Elmer*², *Philip Harris*³, *Isobel Ojalvo*², and *Maurizio Pierini*⁶

¹University of Wisconsin-Madison, USA

²Princeton University, USA

³Massachusetts Institute of Technology, USA

⁴Institute of Physics Belgrade, Serbia

⁵Flatiron Institute, USA

⁶European Organization for Nuclear Research (CERN), Switzerland

Abstract. The high-energy physics community is investigating the potential of deploying machine-learning-based solutions on Field-Programmable Gate Arrays (FPGAs) to enhance physics sensitivity while still meeting data processing time constraints. In this contribution, we introduce a novel end-to-end procedure that utilizes a machine learning technique called symbolic regression (SR). It searches the equation space to discover algebraic relations approximating a dataset. We use PySR (a software to uncover these expressions based on an evolutionary algorithm) and extend the functionality of `hls4ml` (a package for machine learning inference in FPGAs) to support PySR-generated expressions for resource-constrained production environments. Deep learning models often optimize the top metric by pinning the network size because the vast hyperparameter space prevents an extensive search for neural architecture. Conversely, SR selects a set of models on the Pareto front, which allows for optimizing the performance-resource trade-off directly. By embedding symbolic forms, our implementation can dramatically reduce the computational resources needed to perform critical tasks. We validate our method on a physics benchmark: the multiclass classification of jets produced in simulated proton-proton collisions at the CERN Large Hadron Collider. We show that our approach can approximate a 3-layer neural network using an inference model that achieves up to a 13-fold decrease in execution time, down to 5 ns, while still preserving more than 90% approximation accuracy.

1 Introduction

Symbolic regression (SR) is a machine learning technique that seeks to discover mathematical expressions that best fit a dataset. The outcome of SR is an analytic equation that captures the underlying patterns and relationships within the data. As the equations are interpretable, SR can provide valuable insights into natural sciences, including high-energy physics (HEP). Furthermore, by allowing the selection of models on the Pareto front, SR

*e-mail: ho.fung.tsoi@cern.ch

enables the optimization of the performance-resource trade-off, making it a promising alternative to other machine learning methods, especially deep learning models. This is a crucial feature in the context of the Large Hadron Collider (LHC) experiments, which must process proton-proton collisions at a 40 MHz rate and tens of terabytes of raw data per second. This extreme data rate and the current size of the buffering system impose a maximum latency of $O(1) \mu\text{s}$ for the real-time classification and filtering of data at the edge (or the *trigger system*) [1–4]. In these conditions, lightweight algorithms running on custom hardware such as Field-Programmable Gate Arrays (FPGAs) for ultra low-latency inference are desired.

In this paper, we extend the functionality of the `hls4ml`* [5, 6] (High Level Synthesis for Machine Learning) framework to provide parsing capabilities for the equation chosen by SR and High Level Synthesis (HLS) support for mathematical functions. Our implementation is validated on a physics benchmark, demonstrating the effectiveness and potential of this approach to address the challenges faced by the HEP community. For generating the expressions, we have chosen to utilize `PySR`† [7], an open-source software tool for SR that employs an evolutionary algorithm. `PySR` offers a comprehensive implementation of SR and is built on `Julia` but interfaced from `Python`, making it easily accessible and usable for practitioners in a wide range of fields, including HEP. An example code is available on `github`‡. The remainder of this paper is structured as follows. Section 2 introduces the dataset and the baseline model. Section 3 presents our implementations and results. Lastly, Section 4 summarizes the work and suggests future directions.

2 Benchmark and Baseline

To demonstrate the application of SR, we choose the jet identification problem from the HEP field. A jet refers to a narrow cone of outgoing particles, and the process of identifying the original particle that initiated this collimated shower of particles with adjacent trajectories is called *jet tagging*. Jets are central to many physics data analyses at the LHC experiments. The data for this case study are generated from simulated jets that result from the decay and hadronization of quarks and gluons produced in high-energy collisions at the LHC. The task is to tag a given jet as originating from either a quark (q), gluon (g), W boson (W), Z boson (Z), or top quark (t). The dataset is publicly accessible from Zenodo [8]. A variety of jet recombination algorithms and substructure tools are implemented to build a list of 16 physics-motivated expert features: ($\sum z \log z$, $C_1^{\beta=0,1,2}$, $C_2^{\beta=1,2}$, $D_2^{\beta=1,2}$, $D_2^{(\alpha,\beta)=(1,1),(1,2)}$, $M_2^{\beta=1,2}$, $N_2^{\beta=1,2}$, m_{mMDT} , Multiplicity), where the description of each of these variables is presented in Ref. [9]. The anti- k_T algorithm [10] with a distance parameter of $R = 0.8$ is used to cluster all jets. A cut on the reconstructed jet p_T is applied to remove extreme events from the analysis [6]. More detailed descriptions of the dataset can be found in Refs. [6, 9, 11].

The architecture of the baseline model is adopted from Ref. [6], which is a fully connected Neural Network (NN) consisting of three hidden layers of 64, 32, and 32 nodes, respectively, and ReLU activation functions. The input layer takes the 16 high-level features as input, and the output layer consists of five nodes with a softmax activation function, yielding the probability of a jet originating from each of the five classes. This architecture was chosen to provide reasonable performance (75% overall accuracy and 90% per class accuracy) while keeping the model lightweight [6, 12–14]. The model is trained with `QKeras`§, where the kernel weights, biases, and activation functions are quantized to a fixed precision and constrained during weight optimization, called *quantization-aware training* (QAT) [12]. This is

*<https://github.com/fastmachinelearning/hls4ml>

†<https://github.com/MilesCranmer/PySR>

‡<https://github.com/fastmachinelearning/hls4ml-tutorial>

§<https://github.com/google/qkeras>

necessary since post-training quantization (no fine-tuning) results in reduced accuracy. The baseline models presented in Section 3 are fine-tuned for each precision considered. For evaluation, the model is converted to HLS firmware using `hls4ml`.

3 Implementations and Results

To deploy symbolic expressions on FPGAs, we use the `hls4ml` library. We extended `hls4ml` with support for expressions through the Xilinx HLS math library. To further optimize resource utilization and reduce latency, we added functionality to enable approximation of mathematical functions with lookup tables (LUTs). The comparison of LUT-based functions with HLS math library functions is illustrated in Fig. 1. We use $\langle B, I \rangle$ to denote fixed point precision, where B is the total number of bits allocated or bit width, and I is the number of integer bits.

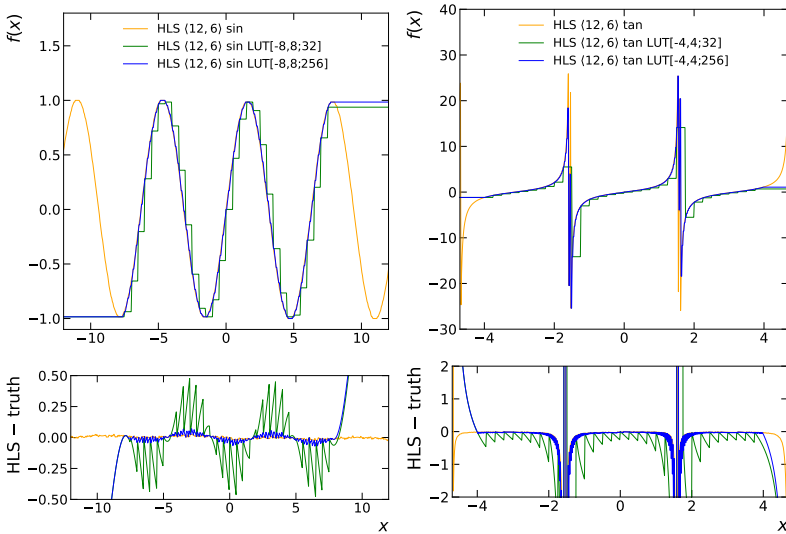


Figure 1. The sine (left) and tangent (right) functions evaluated with and without the use of LUTs, implemented in HLS with precision $\langle 12, 6 \rangle$, i.e., 12 bits variable with 6 integer bits. The LUT notation reads: [range start, range end; table size] for table definition. The lower panel shows the function deviation from the truth.

In the following experiments, we apply SR to fit the LHC jet dataset and demonstrate its resource efficiency in the context of FPGA deployment. We consider models of five independent algebraic expressions as functions of the 16 high-level input features, $\hat{y} = s(\mathbf{x})$ with $s : \mathbb{R}^{16} \rightarrow \mathbb{R}^5$, where the inputs are standardized and the outputs \hat{y} correspond to the score for one of the five jet classes. A jet is identified as the class whose tagger yields the highest score.

The search for expressions is performed using the PySR package. It uses an evolutionary algorithm to construct symbolic expressions, by growing the tree structure using combinations of constants, variables, and operators ($+$, $-$, \times , $/$, $(\cdot)^2$, $\sin(\cdot)$, etc.). The search starts from a random combination without requiring *a priori* knowledge of the underlying functional form. Expressions are evaluated by a specified metric, and the best ones are passed

on to the next generation, where mutation (selecting one node to modify) and crossover (swapping the subtrees of two solutions) can take place to explore more combinations. The PySR measure of complexity, denoted as c , is set to 1 by default for each constant, variable, and operator included in an expression. The complexity of an expression is the sum of the complexity of all its components. We set the model selection strategy so that the candidate model with the lowest loss will be selected regardless of complexity, as long as it does not exceed the maximum value, c_{\max} , of our choice. In this setting, the algorithm attempts to solve the following optimization problem for the dataset $\{(\mathbf{x}^i, \mathbf{y}^i)\}$ with each input $\mathbf{x}^i \in \mathbb{R}^{16}$ and label $\mathbf{y}^i \in \mathbb{R}^5$:

$$h_g^*, h_q^*, h_t^*, h_W^*, h_Z^* = \arg \min_{h_g, h_q, h_t, h_W, h_Z \in \mathcal{S}_{c_{\max}}} \sum_i \sum_{f \in \{g, q, t, W, Z\}} \ell(h_f(\mathbf{x}^i), y_f^i), \quad (1)$$

where $\mathcal{S}_{c_{\max}}$ is the space of equations (i.e., $h_f : \mathbb{R}^{16} \rightarrow \mathbb{R}$) with complexity ranging from 1 to c_{\max} satisfying all constraints specified in the configuration (choice of operators, function nesting, etc.). We use the L2 margin loss, ℓ given by

$$\ell(\hat{y}_f, y_f) = (1 - \hat{y}_f y_f)^2, \text{ with label } y_f = \begin{cases} +1, & \text{if } f \text{ corresponds to the true jet class} \\ -1, & \text{otherwise} \end{cases}. \quad (2)$$

The selection of this loss is due to its domain being \mathbb{R}^2 , since the model outputs are not restricted to any fixed range.

The downside of using evolutionary algorithms for SR is that it is a complex combinatorial problem which does not scale well to high-dimensional datasets. To alleviate this challenge, PySR uses a random forest regressor to evaluate the relative importance of each input feature. We ask PySR to select 6 out of the 16 inputs available for model training in the following experiments.

For resource estimation, each model is converted to FPGA firmware using `hls4ml`, which is then synthesized with Vivado HLS (2020.1) [15], targeting a Xilinx Virtex UltraScale+ VU9P FPGA with part number ‘xcvu9p-flga2577-2-e’. All results are derived after the HLS compilation step. The clock frequency is set to 200 MHz (or clock period of 5 ns), which is typical for the LHC real-time trigger environment [1–4]. The initiation interval is set to 1. In the following studies, we monitor the accuracy, latency, and resource usage (digital signal processors, or DSPs, and LUTs) to compare the models.

3.1 Plain implementation

We first study models with a single class of mathematical functions: polynomial, trigonometric, exponential, and logarithmic. For the polynomial model, only arithmetic operators are considered: $+$, $-$, and \times . For other models, an additional operator is added, respectively: $\sin(\cdot)$ for trigonometric, $\text{Gauss}(\cdot) = \exp(-(\cdot)^2)$ for exponential, and $\log(\text{abs}(\cdot))$ for logarithmic. For simplicity, function nesting (e.g., $\sin(\sin(\cdot))$) is not allowed. Each operator has a complexity of 1 by default. Searches are repeated for $c_{\max} = 20, 40, \text{ and } 80$, to observe how model accuracy and resource usage change with model size. Table 1 shows the expressions per class for the trigonometric model with $c_{\max} = 20$. Table 2 shows expressions for the t tagger in all models with $c_{\max} = 40$. Accuracy is shown in Fig. 2. FPGA resource usage and latency are shown in Fig. 3.

3.2 Function approximation with LUTs

Based on the models in Section 3.1 (except for the polynomial), we approximate all mathematical functions with LUTs and perform the analysis again. In Fig. 2 and 3, these models

Tagger	Expression for the trigonometric model with $c_{\max} = 20$	AUC
g	$\sin(-2C_1^{\beta=1} + 0.31C_1^{\beta=2} + m_{\text{mMDT}} + \text{Multiplicity} - 0.09\text{Multiplicity}^2 - 0.79)$	0.897
q	$-0.33(\sin(m_{\text{mMDT}}) - 1.54)(\sin(-C_1^{\beta=1} + C_1^{\beta=2} + \text{Multiplicity}) - 0.81)\sin(m_{\text{mMDT}}) - 0.81$	0.853
t	$\sin(C_1^{\beta=1} + C_1^{\beta=2} - m_{\text{mMDT}} + 0.22(C_1^{\beta=2} - 0.29)(-C_1^{\beta=2} + C_2^{\beta=1} - \text{Multiplicity}) - 0.68)$	0.920
W	$-0.31(\text{Multiplicity} + (2.09 - \text{Multiplicity})\sin(8.02C_1^{\beta=2} + 0.98)) - 0.5$	0.877
Z	$(\sin(4.84m_{\text{mMDT}}) + 0.59)\sin(m_{\text{mMDT}} + 1.14)\sin(C_1^{\beta=2} + 4.84m_{\text{mMDT}}) - 0.94$	0.866

Table 1. Expressions generated by PySR for the trigonometric model with $c_{\max} = 20$. The operator complexity is set to 1 by default. Constants are rounded to two decimal places for readability. The area under the receiver operating characteristic (ROC) curve, or AUC, is reported.

Model	Expression for the t tagger with $c_{\max} = 40$	AUC
Polynomial	$C_1^{\beta=2} + 0.09m_{\text{mMDT}}(2C_1^{\beta=1} + M_2^{\beta=2} - m_{\text{mMDT}} - \text{Multiplicity} - (1.82C_1^{\beta=1} - M_2^{\beta=2})(C_1^{\beta=2} - 0.49m_{\text{mMDT}} - 3.22) - 0.53$	0.914
Trigonometric	$\sin(0.06(\sum z \log z)M_2^{\beta=2} - 0.25C_1^{\beta=2}(-C_1^{\beta=1} + 2C_1^{\beta=2} - M_2^{\beta=2} + \text{Multiplicity} - 8.86) - m_{\text{mMDT}} + 0.06\text{Multiplicity} - 0.4)$	0.925
Exponential	$0.23C_1^{\beta=1}(-m_{\text{mMDT}} + \text{Gauss}(0.63\text{Multiplicity} + 1) - \text{Gauss}(C_1^{\beta=1}) + 0.45C_1^{\beta=2} - 0.23m_{\text{mMDT}} + 0.23\text{Gauss}((4.24 - 1.19C_2^{\beta=1})(C_1^{\beta=2} - m_{\text{mMDT}})) + 0.15$	0.920
Logarithmic	$C_1^{\beta=2} - 0.1m_{\text{mMDT}}(\text{Multiplicity} \times \log(\text{abs}(\text{Multiplicity})) + 2.2) - 0.02\log(\text{abs}(\text{Multiplicity})) - 0.1(C_1^{\beta=2}(C_1^{\beta=1} - 1.6M_2^{\beta=2} + m_{\text{mMDT}} + 1.28) - m_{\text{mMDT}} - 0.48)\log(\text{abs}(C_1^{\beta=2})) - 0.42$	0.923

Table 2. Expressions generated by PySR for the t tagger in different models with $c_{\max} = 40$. Operator complexity is set to 1 by default. Constants are rounded to two decimal places for readability.

correspond to the dashed lines. Compared to the baseline, the resource usage is dramatically reduced for all SR models, especially for those applying function approximation, sometimes with several orders of magnitude improvements. Besides, SR models require significantly shorter inference time than the baseline, while having minimal drop in accuracy. In particular, the inference time is reduced to as low as 1 clock cycle (5 ns) in certain scenarios in the exponential and logarithmic models with LUT-based functions implemented, which amounts to a reduction by a factor of 13 compared to the baseline, which has a latency of 13 clock cycles (65 ns), while still maintaining a relative accuracy above 90%. The ROC curves of the baseline and trigonometric models are compared in Fig. 4.

3.3 Latency-aware training

Alternatively, resource usage can be improved by guiding PySR to search in a latency-aware manner. By default, PySR assigns the complexity for every operator to 1 so that they are all equally penalized when added to expression trees. However, it is not ideal for FPGA deployment since, for example, an operator $\tan(\cdot)$ typically takes several times more clock cycles than an $\sin(\cdot)$ to evaluate on an FPGA. This time cost can be incorporated in expression searches by setting operator complexity to the corresponding number of clock cycles needed on FPGAs. Note that this strategy is not valid in the context of function approximation with LUTs since every indexing operation requires only one clock cycle.

We demonstrate this latency-aware training (LAT) for two precisions, $\langle 16, 6 \rangle$ and $\langle 18, 8 \rangle$, with c_{\max} ranging from 20 to 80. We consider the following operators: $+(1)$, $-(1)$, $\times(1)$, $\log(\text{abs}(\cdot))(4)$, $\sin(\cdot)(8)$, $\tan(\cdot)(48)$, $\cosh(\cdot)(8)$, $\sinh(\cdot)(9)$, and $\exp(\cdot)(3)$, where the numbers in parentheses correspond to the operator complexity for $\langle 16, 6 \rangle$ as an example. For simplicity, function nesting is not allowed again. We also constrain the total complexity of the subtrees. In this way, we force the model to explore solutions in a different part of the Pareto front. The final expressions are shown in Table 3. Model accuracy, resource usage, and latency are shown in Fig. 5. SR models obtained from LAT use systematically fewer resources and have a lower latency compared to those obtained from plain implementation,

while having comparable accuracy. Implementation of a maximum-latency constraint is also possible. We added a script to generate operator complexity for practitioners[¶].

Operator complexity	Expression for the t tagger with $c_{\max} = 40$	AUC
All 1's (PySR default)	$0.11(C_1^{\beta=1} + C_1^{\beta=2} + \log(\text{abs}(C_1^{\beta=2}))) - 0.48m_{\text{mMDT}} - 0.05\text{Multiplicity}(\text{Multiplicity} + \log(\text{abs}(m_{\text{mMDT}}))) - \sin(-C_1^{\beta=2} + 0.14C_2^{\beta=1}m_{\text{mMDT}}) + 0.11\sinh(C_1^{\beta=1}) - 0.24$	0.930
No. of clock cycles at (16, 6)	$0.04((\sum z \log z) + C_1^{\beta=1} + C_2^{\beta=1} - m_{\text{mMDT}} - (\text{Multiplicity} - 0.2)(\text{Multiplicity} + \log(\text{abs}(C_1^{\beta=2})))) - \sin(-C_1^{\beta=1} - C_1^{\beta=2} + 1.23m_{\text{mMDT}} + 0.58)$	0.924
No. of clock cycles at (18, 8)	$0.04\text{Multiplicity}(C_1^{\beta=2}(C_1^{\beta=2} - m_{\text{mMDT}}) - \text{Multiplicity} - \log(\text{abs}(C_1^{\beta=2}((\sum z \log z) + 0.23)))) - \sin(-C_1^{\beta=1} - C_1^{\beta=2} + 1.19m_{\text{mMDT}} + 0.61)$	0.926

Table 3. Expressions generated by PySR for the t tagger with $c_{\max} = 40$, implemented with and without LAT. Constants are rounded to two decimal places for readability.

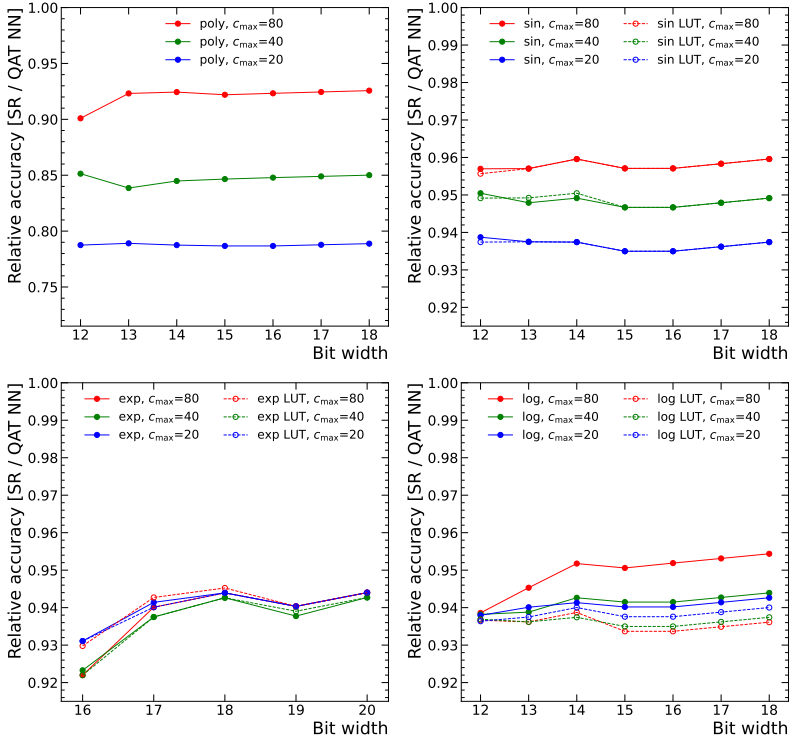


Figure 2. Relative accuracy as a function of bit width, for polynomial (top left), trigonometric (top right), exponential (bottom left), and logarithmic (bottom right) models. The relative accuracy is evaluated with respect to the baseline QAT NN trained and implemented at corresponding precision. The number of integer bits is fixed at $I = 12$ for the exponential model and at $I = 6$ for other models.

[¶]<https://github.com/AdrianAlan/hls4sr-configs>

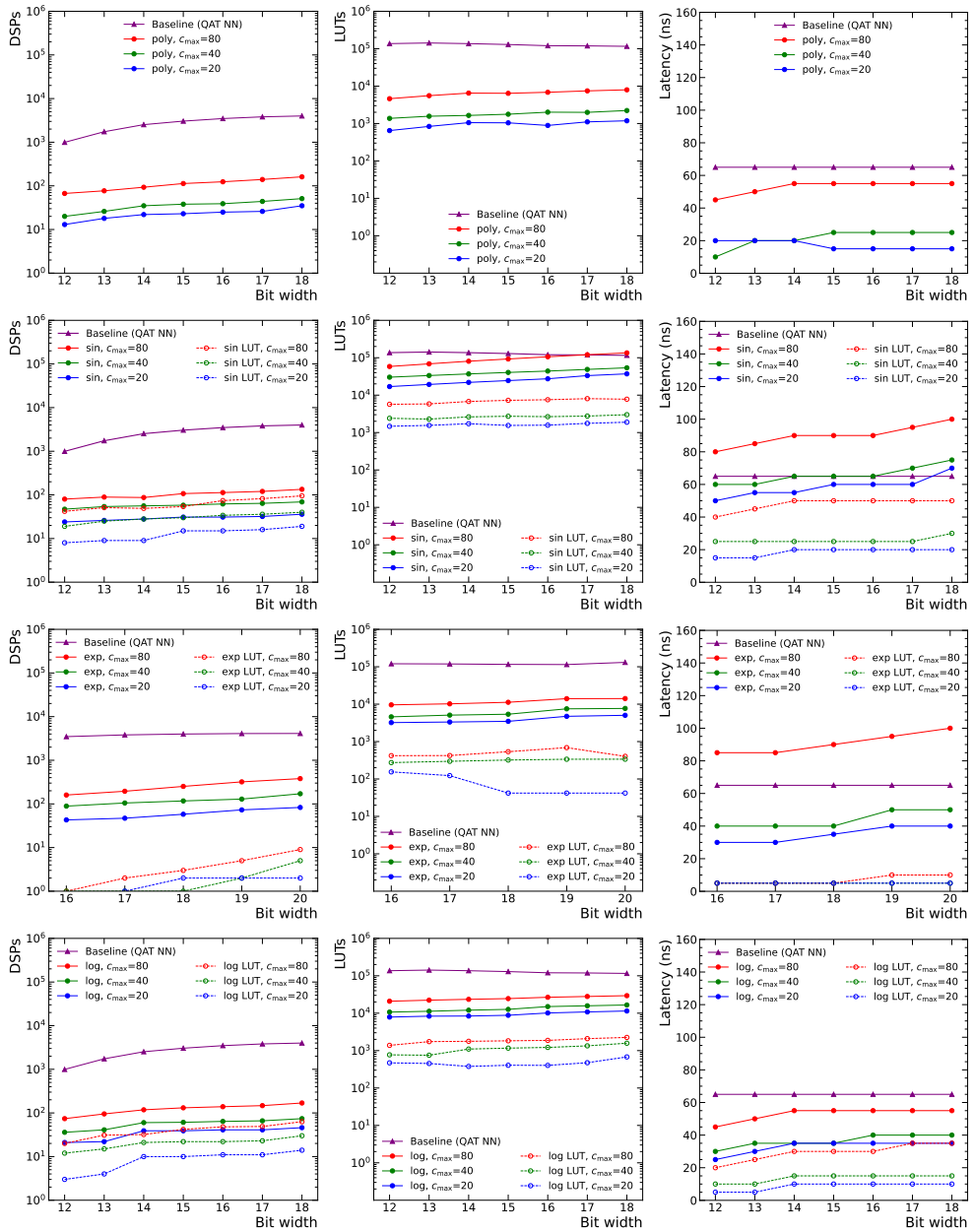


Figure 3. DSPs usage (left), LUTs usage (middle), and latency (right) as a function of bit width. From top to bottom: polynomial, trigonometric, exponential, and logarithmic models. The baseline QAT NN trained and implemented at corresponding precision is shown for comparison. Resource usage and latency are obtained from C-synthesis on a Xilinx VU9P FPGA with part number ‘xcvu9p-flga2577-2-e’.

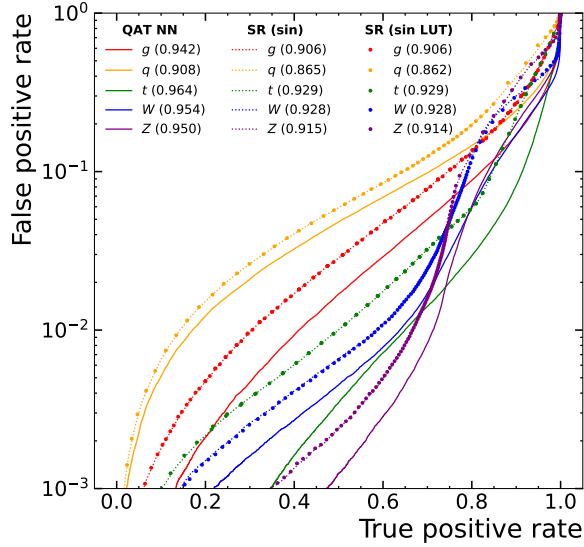


Figure 4. ROC curves for the trigonometric models with $c_{\max} = 80$ implemented with precision $\langle 16, 6 \rangle$, as compared to the baseline QAT NN. Numbers in parentheses correspond to the AUC per class.

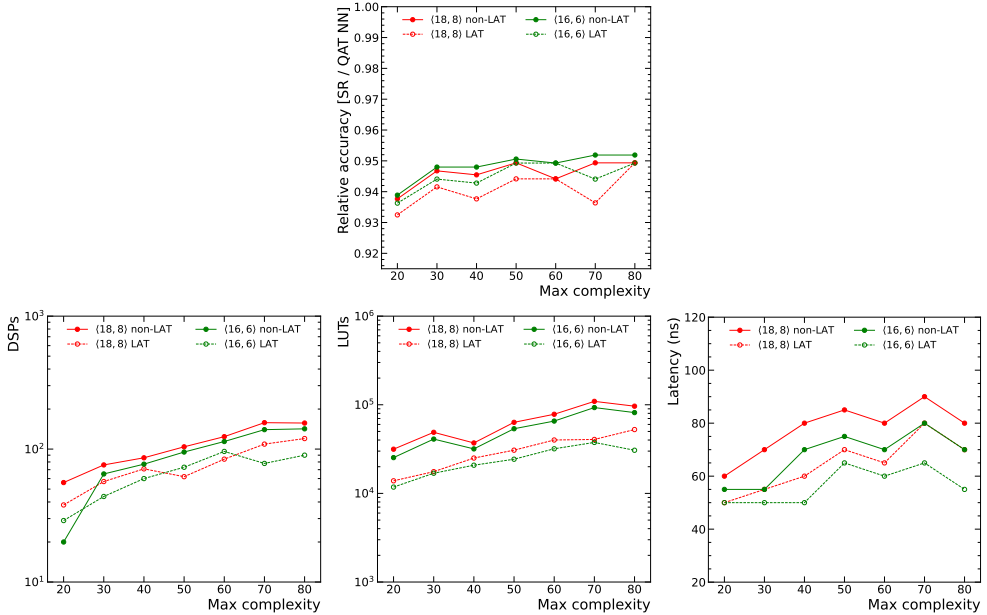


Figure 5. Relative accuracy (top), DSPs usage (bottom left), LUTs usage (bottom middle) and latency (bottom right) as a function of c_{\max} ranging from 20 to 80, comparing models obtained from plain implementation (solid) and LAT (dashed). Two precision settings are implemented: $\langle 16, 6 \rangle$ and $\langle 18, 8 \rangle$. The relative accuracy is evaluated with respect to the baseline model. Resource usage and latency are obtained from C-synthesis on a Xilinx VU9P FPGA with part number ‘xcvu9p-flga2577-2-e’.

4 Summary and Outlook

In this paper, we have presented a novel end-to-end procedure to utilize symbolic regression (SR) in the context of FPGAs for fast machine learning inference. We extended the functionality of the `hls4ml` package to support the expressions generated by PySR. We demonstrated the effectiveness of our approach on a physics benchmark (jet tagging at the LHC) and showed that our implementation of SR on FPGAs provides a way to dramatically reduce the computational resources needed to perform critical tasks, making it a promising alternative to deep learning models. The utilization of SR in HEP provides a solution to meet the sensitivity and latency demands of modern physics experiments. The results of this study open up new avenues for future work, including further optimization of the performance-resource trade-off and the exploration of other application domains for SR on FPGAs.

5 Acknowledgments

We acknowledge the Fast Machine Learning collective as an open community of multi-domain experts and collaborators. This community was important to the development of this project. H.F.T. and S.D. are supported by the U.S. Department of Energy (Award No. DE-SC0017647). A.A.P. is supported by the Eric and Wendy Schmidt Transformative Technology Fund. V.L. and P.H. are supported by A3D3 (NSF 2117997). P.H. is also supported by the IAIFI grant. M.P. is supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (Grant No. 772369).

References

- [1] ATLAS Collaboration, *JINST* **15**, P10004 (2020), [2007.12539](#)
- [2] ATLAS Collaboration, CERN-LHCC-2017-020, ATLAS-TDR-029 (2017)
- [3] CMS Collaboration, *JINST* **15**, P10017 (2020), [2006.10165](#)
- [4] CMS Collaboration, CERN-LHCC-2020-004, CMS-TDR-021 (2020)
- [5] FastML Team, *fastmachinelearning/hls4ml* (2021), <https://github.com/fastmachinelearning/hls4ml>
- [6] J. Duarte et al., *JINST* **13**, P07027 (2018), [1804.06913](#)
- [7] M. Cranmer, *PySR: Fast & parallelized symbolic regression in python/julia* (2020), <https://github.com/MilesCranmer/PySR>
- [8] M. Pierini, J.M. Duarte, N. Tran, M. Freytsis, *HLS4ML LHC Jet dataset (150 particles)* (2020), <https://doi.org/10.5281/zenodo.3602260>
- [9] E. Coleman, M. Freytsis, A. Hinzmann, M. Narain, J. Thaler, N. Tran, C. Vernieri, *JINST* **13**, T01003 (2018), [1709.08705](#)
- [10] M. Cacciari, G.P. Salam, G. Soyez, *JHEP* **04**, 063 (2008), [0802.1189](#)
- [11] E.A. Moreno, O. Cerri, J.M. Duarte, H.B. Newman, T.Q. Nguyen, A. Perival, M. Pierini, A. Serikova, M. Spiropulu, J.R. Vlimant, *Eur. Phys. J. C* **80**, 58 (2020), [1908.05318](#)
- [12] C.N. Coelho, A. Kuusela, S. Li, H. Zhuang, T. Aarrestad, V. Loncar, J. Ngadiuba, M. Pierini, A.A. Pol, S. Summers, *Nature Mach. Intell.* **3**, 675 (2021), [2006.10159](#)
- [13] V. Loncar et al., *Mach. Learn. Sci. Tech.* **2**, 015001 (2021), [2003.06308](#)
- [14] B. Hawks, J. Duarte, N.J. Fraser, A. Pappalardo, N. Tran, Y. Umuroglu, *Front. Artif. Intell.* **4**, 676564 (2021), [2102.11289](#)
- [15] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf (2020)