Enhancing the Computational Efficiency of Genetic Programming Through Alternative Floating-Point Primitives

Christopher Crary[1(✉)], Bogdan Burlacu[2], and Wolfgang Banzhaf[3]

[1] Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
ccrary@ufl.edu

[2] Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Hagenberg, Upper Austria, Austria
bogdan.burlacu@fh-ooe.at

[3] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA
banzhafw@msu.edu

Abstract. Can evolution operate effectively with noisy floating-point function primitives? In this paper, we are motivated by recent work that aims to accelerate genetic programming (GP) through specialized hardware and field-programmable gate arrays (FPGAs), for which it has been shown that additional performance and power/energy benefits could likely be achieved with floating-point function primitives that trade off enhanced computational efficiency for increased error. Although GP is known to be robust in filtering out certain forms of noise (e.g., within input data), it is not immediately clear that less-accurate function primitives would be viable for GP, since GP formulates arbitrary compositions of its primitives, which could potentially compound error to a prohibitive level. In addition, when introducing more complex forms of computation, such as function differentiation and local optimization techniques, it is not readily apparent that using rougher primitive implementations would be tenable. Here, we address both situations by employing the state-of-the-art CPU-based Operon tool on a diverse set of 15 regression problems, and we show that tree-based GP is capable of evolving very similar (and sometimes better) results with alternative high-performance approximations of standard function primitives, while often also allowing for faster CPU runtimes. Most importantly, in the context of specialized hardware, we conclude that our proposed techniques can likely allow for significant speedups over general-purpose computing platforms, as well as improved power/energy efficiency.

Keywords: Genetic programming · Field-programmable gate arrays · Approximate computing · Floating-point · Symbolic regression

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1718033 and CCF-1909244.

M. Affenzeller et al. (Eds.): PPSN 2024, LNCS 15148, pp. 322–339, 2024.
https://doi.org/10.1007/978-3-031-70055-2_20

1 Introduction

Recent trends in machine learning highlight the need for energy-efficient computation, during both training and inference [19,33,50]. For example, despite widespread success, neural networks often consume prohibitive amounts of power and energy for many use cases [5,50], in addition to posing considerable scaling challenges for well-established use cases, such as data centers [1,5,40,50]. Such limitations can motivate other learning systems, such as genetic programming (GP) [4,30,43], where it has been widely shown that the pairing of evolutionary search with alternative model structures (e.g., trees, assembly languages, tangled program graphs, etc.) can sometimes allow for significantly more compact solutions and enhanced efficiency during inference [26,31,43]. However, training often remains complex with current GP techniques, which motivates the exploration of improvements to training efficiency [7,13,14].

There are various ways to improve the training efficiency of GP, and they generally involve either increasing performance (i.e., throughput) or enhancing energy efficiency, for which there are at least four key benefits: (1) with increased performance, useful solutions can potentially be found in a shorter amount of time; (2) with improved energy efficiency, there is the potential for lower operational costs, which (3) can allow for more cost-effective multi-computer GP systems, in turn allowing for higher performance; and (4) with either improved performance or improved energy efficiency, better solutions can potentially be found when allowing the system to consume a similar amount of runtime/energy.

In this paper, we explore the possibility of improving the training efficiency of GP by way of alternative floating-point math approximations for standard function primitives.[1] As described further in Sect. 2, we are motivated by recent work that suggests modern field-programmable gate array (FPGA) devices are capable of providing significant improvements to both the runtime and energy efficiency of GP, when compared to solutions employing general-purpose CPU/GPU systems [11]. Importantly, for floating-point applications, it is suggested in [11] that a key optimization needed for achieving significant computational efficiency with an FPGA relies on the idea of implementing GP function primitives with a minimal amount of shared single-precision floating-point multiply-add (MAD) resources, for which modern FPGA devices have native support. Unfortunately, designing this resource-sharing mechanism for standard implementations of floating-point operators is challenging, due to high-performance requirements of the relevant algorithms often leading to overly complex or obfuscated realizations. However, this begs the question: *are standard floating-point operators even needed in the context of GP?* Could more efficient implementations that more roughly approximate the corresponding continuous operators suffice for evolution? We answer such questions in this paper.

As a proof-of-concept, we consider fifteen diverse benchmark problems within the application domain of symbolic regression [28,39,42,52], and we extend the

[1] We specify "alternative" since standard implementations of floating-point functions are themselves approximations of continuous counterparts [20].

recent state-of-the-art tool *Operon* [7] with several new computational backends, in order to compare various single-precision floating-point approximations that trade off increased error for enhanced performance. Notably, we find that not only can the set of approximations with the *highest* amount of error perform very similar to standard implementations, such approximations can even allow evolution to evolve more compact and higher-quality models (in terms of fitness) within a shorter period of time, depending on the dataset.

Ultimately, there are several important takeaways from this work. First, within the context of specialized hardware, it is clear that there is the potential to leverage high-performance approximations of single-precision floating-point math without dramatically reducing the solution quality of evolution, even when performing complex operations that compound approximation error, such as local optimization through gradient-based techniques. Second, when applying such approximations to CPU systems, there is the possibility for practically-significant performance enhancements, especially for problems with larger datasets. For our tests using an AMD Ryzen 5950X CPU, with 16 cores, 3.4 GHz clock frequency, and AVX2 instruction optimizations, we achieved average speedups of up to 2.02×, and average energy reductions of up to 1.78×. Third, on a more theoretical level, it has been shown in other ML paradigms that noise is beneficial to produce simpler and more general solutions [15,32], a result that can be readily applied in GP [3,55]. Using alternative function implementations can be similar to injecting noise into solutions, with the added benefit that it is faster. Thus, our results give further credence to the idea that evolution can robustly deal with various, sometimes significant, forms of noise and discretization. Overall, our current work represents a significant first step in the direction of enhancing the computational efficiency of GP systems through the use of alternative implementations of standard floating-point function primitives, and it helps lay the foundation for how specialized hardware could exhibit significant performance and energy advantages over general-purpose CPU/GPU systems.

The rest of this paper is organized as follows. In Sect. 2, we present additional background and motivation for the proposed techniques. In Sect. 3, we overview some related work. In Sect. 4, we describe our chosen floating-point math approximations. In Sect. 5, we detail our design of experiments. In Sect. 6, we present and analyze results. In Sect. 7, we conclude the study.

2 Background

Within GP, the performance bottleneck for training and the primary candidate for efficiency improvements is generally the evaluation of individuals [4,43]. Although this subroutine is normally an embarrassingly parallel procedure [43], it can be challenging to accelerate with general-purpose CPU/GPU systems, primarily due to the need for evaluating dynamically-evolving programs [11]. More specifically, although GPUs have numerous computation cores, the need for conditional program execution (e.g., to decide which function primitive to

execute) and large cache sizes generally limits acceleration capabilities, and even though CPUs are better equipped for conditional execution, it is prohibitively expensive to continually scale up the number of CPU cores/threads [7,11,22].

To address such limitations of general-purpose systems, recent work has explored the possibility of creating specialized hardware accelerators for GP using modern field-programmable gate array (FPGA) devices [11]. In brief, FPGAs are programmable computing platforms for which specialized digital circuits can be designed, without the need to manufacture integrated circuits. When compared to FPGA technologies leveraged by some older GP systems (e.g., [17,18,29,47]), contemporary FPGAs are better equipped to handle GP acceleration due to the availability of significantly more resources, as well as more powerful resources, e.g., native components for floating-point multiply-add (MAD) operations [9].

Importantly, for floating-point applications, it is suggested by [11, Sect. 7.2, item 2] that a key optimization needed for achieving significant computational efficiency with an FPGA relies on the idea of designing GP function primitives that both minimize and share low-level floating-point MAD resources, referred to as "floating-point DSPs" in [11]. For example, suppose that we have the function set $\mathcal{F} = \{+, -, *, x^2\}$, where x^2 represents the squaring operation. In essence, if computation cores are meant to compute only one primitive during any given clock cycle, then rather than implement this set of function primitives with four MAD components (where each MAD is a multiplier followed by an adder), a core could instead share a single MAD component across all functions—since no operation here independently requires more than one—which would free up more FPGA resources for more parallel computation.

Perhaps the most important consequence of this aforementioned strategy is that the number of MAD resources required for a single computation engine is dictated by the primitive that requires the *maximum* number of MAD operations. Considering the prior example, if we were to include a primitive that required ten MADs, the minimum amount of MADs that would be required for the sharing scheme would be ten. Therefore, there is significant motivation to *minimize the maximum* number of MAD operations across all functions.

To minimize the maximum number of MADs, the most obvious option is to remove complex function primitives. Unfortunately, even relatively straightforward operations such as divide or logarithm can potentially take a significant number of MADs, which then limits the potential throughput and power/energy benefits of specialized hardware. Fortunately, there is another possibility: *replace complex function primitives with approximations that are more efficient in terms of MADs and any other low-level device resources.* In this case, we would generally be trading off increased error for enhanced performance: the higher error that we can allow, the more likely that we can simplify the relevant hardware implementation. However, we must first establish whether or not GP can operate effectively with primitives containing non-negligible amounts of error.

Although GP is known to be robust in filtering out certain forms of noise (e.g., within input data), it is not immediately clear that approximating function

primitives would be viable for GP, since GP formulates arbitrary compositions of its primitives, which could potentially compound error to a prohibitive level. In addition, even if standard program evaluation could effectively leverage more coarse-grained approximations, it is not readily apparent that using such approximations for other forms of computation—such as function differentiation and local optimization—would be tenable. In this paper, we address both situations.

3 Related Work

Approximate computing is an emerging paradigm that exploits the acceptable error in applications in order to enable more effective approximations that improve application performance/energy [35,46]. Although the widespread usage of machine learning (ML) has made approximation a mainstream design strategy, various forms of approximate computing remain limited within evolutionary computation (EC) domains [46]. Regarding genetic programming (GP) [4,43], numerous works have employed GP in order to construct high-performance solutions, e.g., [24,53], but comparatively few works have developed approximations for GP itself, especially for floating-point applications [46].

Approximate computing often goes unused because most developers are unaware of existing approximations, and they are unlikely to create new approximations, especially for different applications or architectures. However, the automatic nature of EC can potentially alleviate such issues [46]. For example, recent work has demonstrated how competitive floating-point function approximations can be automatically found from scratch using EC [45]. Such techniques are complementary to our current work, which explores how high-performance and less-accurate primitives may affect evolutionary search.

As with other ML domains, the use of low-precision numerical systems (e.g., 8/16-bit floating-point or fixed-point) can likely lend itself to more energy-efficient computation for GP applications [11,21]. Such strategies are complementary to our proposed technique, in which we implement standard GP function primitives with alternative approximations, rather than with alternative number systems. For our current work, we employ the IEEE-754 single-precision floating-point format [20], but future work could potentially employ any numerical format. In addition, we target symbolic regression as a proof-of-concept, but approximate primitives could likely be used for other types of problems.

4 Methodology

For this study, we consider various alternative approximations for several commonly used mathematical primitives, but we leverage standard implementations for addition and multiplication, since standard multiply-add (MAD) operations are natively supported by the FPGA technologies that we intend to target (Sect. 2) [9,23,51]. Ultimately, we use alternative approximations for computing

program outputs and for computing derivatives when performing local optimization techniques. Other computation such as that for fitness measures are implemented using standard math implementations. These other procedures could potentially be approximated as well, but we leave this for future work. Care must likely be taken if approximating fitness measures, since poor approximations might deceptively promote inappropriate solutions.

The math primitives that we choose to approximate are division, sine, cosine, natural exponentiation, natural logarithm, square root, and hyperbolic tangent. For these primitives, we consider three different implementation sets, where we generally trade off lower complexity in terms of *maximum* number of MAD resources for lower function accuracy, since the maximum number of MADs directly affects the possible throughput of specialized hardware (Sect. 2):

MAD-16 (16 MADs). Lowest function error—see Table 1—but also the lowest theoretical performance for hardware. This implementation is based on the VDT library, which has been successfully used at CERN [41].

MAD-10 (10 MADs). Notable middle ground between the number of MAD resources and function error. The threshold of ten was defined primarily based on the number of MAD resources needed to significantly improve the error of sine/cosine when compared to the MAD-04 implementation.

MAD-04 (4 MADs). Highest performance and highest error, when compared to the other implementations. Very little complexity, in terms of MADs.

With recent higher-end FPGA devices, thousands of MAD components can potentially be available [9]. For example, if we suppose that 9,000 (roughly 75%) of the 12,300 MAD resources for the "AGM 039" device listed in [23] could be utilized for specialized GP computation cores (Sect. 2) [11], and if we suppose that a moderate clock frequency of 300 MHz could be utilized [9,37,49,51], then our three proposed primitive set implementations could allow for a theoretical peak throughput of 168.75 billion, 270 billion, and 675 billion node operations per second, respectively. (We divide total number of MAD resources by the numbers listed for our primitive set implementations, and then multiply by 300 million.)

Notably, other optimizations are possible in order to increase throughput even further. First and foremost, the design could be optimized for timing, so that a higher clock frequency may be used [9,51]. Then, in contexts where local optimization is expected and a weight term is to be allocated to each program node—similar to Operon—there effectively can be up to 3× as many nodes in each program once accounting for the multiplication operations, and we can compute the extra multiplication for each node in parallel to a computation core with just *one extra MAD resource*, which would allow theoretical peak throughput values to be multiplied by three. (Given the assumptions of the previous example, it is possible to allocate the extra MAD resource for each computation core.) In fact, we plan to consider also adding bias terms to the Operon system, in which case a similar argument could allow for up to a 5× improvement in peak throughput. We leave other optimizations for future work.

Our work also motivates two novel model deployment strategies: (1) training and deploying with the same alternative primitive implementations, and (2)

training with alternative implementations and deploying with standard implementations, where we apply linear scaling in both cases [25]. Due to the fact that we are using approximations for the standard single-precision floating-point format [20], item (1) offers a clear path to both energy-efficient training and energy-efficient inference, either with general-purpose or specialized computer systems. However, if the use of such primitives during inference is not desirable for any reason, then item (2) offers a meaningful strategy for using standard primitives.

Approximation Details

Below, we discuss the algorithms used by our implementation sets. As a side note, when we employ approximations defined by the VDT library [41], we usually perform additional input validation. Refer to our code for more details [10].

Division. We use $\text{div}(x_1, x_2) = x_1 \times \frac{1}{x_2} = x_1 \times (x_2^{-1})$, and we approximate x_2^{-1} by first exploiting some well-known numerical properties of the single-precision floating-point encoding [20], and then by improving an initial estimate with Newton-Raphson (NR) iterations [36]. The number of NR iterations distinguishes performance and error among our three sets. We use 4, 4, and 1 iterations for the MAD-16, MAD-10, and MAD-04 sets, respectively.

Sine/Cosine. We use polynomial approximations of varying accuracy. For the MAD-16 and MAD-10 sets, we employ the approximation given by the VDT library [41]; for the MAD-04 set, we augment the following simple approximation of a particular sine wave once reducing the input x to the range $[-1, 1]$: $x \cdot (1 - \text{abs}(x))$. We note that sine and cosine were the primary bottlenecks for reducing the maximum number of MAD resources; it is challenging to construct practical implementations of these functions that require less than four MADs. In addition, we note that our chosen approximations necessitate a limited input domain, due to limited capabilities in transforming arbitrary floating-point inputs to a relevant range like $[-1, 1]$, although future work can explore using other methods at the cost of increased complexity.

Natural Exponentiation. For the MAD-16 set, we employ the approximation given by the VDT library [41]. For the other two sets, we use $\exp(x) = 2^{x/\log(2)}$. We split $t \triangleq x/\log(2)$ into an integer i and fraction f such that $t = i + f$ and $0 \leq f < 1$. From this, we can compute $2^{x/\log(2)} = 2^f \cdot 2^i$ by first approximating 2^f with a polynomial, and then by scaling with 2^i, the latter of which can be performed simply by adding i to the encoded exponent value of the floating-point result 2^f [20]. When approximating 2^f, we use polynomials of degree-6 and degree-2 for the MAD-10 and MAD-04 sets, respectively.

Natural Logarithm. For the MAD-16 set, we employ the approximation given by the VDT library [41]. For the other two sets, we consider inputs to be of the form $x = m \cdot (2^e)$, where e is the unbiased exponent value of the floating-point input [20], and where m is $1.0 + m'$ for mantissa $0 \leq m' < 1$. Then, we use

$\log(x) = \log(m) + e \cdot \log(2)$. To compute $\log(m)$, we use polynomials of degree-6 and degree-2 for the MAD-10 and MAD-04 sets, respectively.

Square Root. We employ $\text{sqrt}(x) = x \cdot (x^{-0.5})$, and we approximate $x^{-0.5}$ with the fast inverse square root approximation [36], employing 4, 2, and 1 N-Raphson iterations for the MAD-16, MAD-10, and MAD-04 sets, respectively.

Hyperbolic Tangent. For the MAD-16 and MAD-10 sets, we use $\tanh(x) = 1 - 2/(\exp(2x) + 1)$, saturating to -1 or $+1$ when outside of the range $[-32, 32]$, and we use methods similar to what was previously described for approximating the exponentiation and reciprocal operations. For the MAD-04 set, we use a simple polynomial approximation and saturate when outside of the range $[-3, 3]$. Similar to sine/cosine, it is challenging to construct practical implementations of this function that require less than four MADs.

Table 1. Median relative percentage error for the proposed functions and their associated derivatives. See the text about results of tanh derivative for MAD-04.

	Mad-16		Mad-10		Mad-04	
	f	∂f	f	∂f	f	∂f
div	0	0	0	0	0.08353	0.25052
sin	0	0	0	0	6.87843	5.30876
cos	0	0	0	0	5.30876	6.87843
exp	0	0	9.28e−6	9.28e−6	0.11890	0.11890
log	0	0	0	0	0	0.08327
sqrt	0	0	0	0	0	0
tanh	5.96e−6	1.71e−4	1.43e−4	2.24539	0.00897	100

Following from the above, we list in Table 1 median relative percentage error values for each function and its associated derivative(s), for each MAD backend. We include derivatives since we leverage gradient-based local search. For each primitive, we use a set of one million random inputs, uniformly distributed in the interval $[-10, 10]$, and we use the NumPy Python library as a baseline when computing relative error [38]. Note that the relative error of each partial derivative for arity-two functions independently contributes to the relevant error listed. Also, note that the derivative of the tanh function for MAD-04 was frequently zero, which caused the median relative error to be 100%, even though absolute error was reasonable. Ultimately, we consider the listed errors to be acceptable.

5 Empirical Study

We investigate the empirical validity of our proposed methodology on a collection of real-world and synthetic symbolic regression problems, which are described in

Table 2. Summary of the chosen benchmark problems. F_4 to F_{15} are synthetic.

Id	Name	Features	Instances	Training range
F_1	Airfoil Self Noise [6]	5	1503	$[0, 1000)$
F_2	Chemical-I [28]	57	1066	$[0, 711)$
F_3	Concrete [54]	8	1000	$[0, 500)$
F_4	Friedman-I [16]	10	10 000	$[0, 5000)$
F_5	Friedman-II [16]	10	10 000	$[0, 5000)$
F_6	Poly-10 [42]	10	500	$[0, 250)$
F_7	Pagie-1 [39]	2	1676	$[0, 676)$
F_8	Vladislavleva-1 [52]	2	2125	$[0, 100)$
F_9	Vladislavleva-2 [52]	1	321	$[0, 100)$
F_{10}	Vladislavleva-3 [52]	2	5683	$[0, 600)$
F_{11}	Vladislavleva-4 [52]	5	6024	$[0, 1024)$
F_{12}	Vladislavleva-5 [52]	3	3000	$[0, 300)$
F_{13}	Vladislavleva-6 [52]	2	93 666	$[0, 30)$
F_{14}	Vladislavleva-7 [52]	2	1300	$[0, 300)$
F_{15}	Vladislavleva-8 [52]	2	1206	$[0, 50)$

Table 2. In particular, we are interested to observe the extent of GP's ability to evolve solutions when standard function primitives exhibit significant numerical deviations (Table 1), as well as how such noise is compounded by compositional expressions and, separately, by gradient-based local search [7]. We incorporate the proposed MAD implementations in Operon [7], and we compare with three other backends based on the Eigen, STL, and VDT C++ libraries [7,41].

We employ the NSGA-II algorithm [12] with a population of 1,000 individuals and a computational budget that stops evolution after either one million solution evaluations or 1,000 generations. The remaining parameters are given in Table 3. For each combination of computational backend and benchmark problem, we first perform a set of 100 runs without local search. Then, we repeat this experiment, but with three iterations of local search using the Levenberg-Marquardt (LM) algorithm [27], which consumes the computational budget much faster.

The different numerical properties of GP primitives from each backend affect the underlying genotype-to-phenotype maps, thus affecting the evolvability of the representation [2]. Therefore, it is possible to obtain solutions which rely on certain approximate behavior in order to maximize fitness. For this reason, it is important to consider re-evaluating and deploying all solutions with a common computational backend. However, our work also enables the possibility of deploying GP models with the same backend employed during training. Here, we consider both forms of deployment, and when using the former, we leverage the STL backend to re-evaluate all solutions. For both forms of deployment, we apply linear scaling using the training data [25]. When performing linear scaling,

rather than have default weight/bias values of 1.0 and 0.0 when encountering not-a-number (NaN) program outputs, we replace such outputs with zero.

Table 3. Run parameters for Operon

Tree constraints	Depth ≤ 10, Length ≤ 50
Fitness function	R^2 (coefficient of determination)
Crossover probability	100%
Mutation probability	25%
Selection mechanism	Crowded tournament selection, size of 5
Function set	$+$, $-$, $\times$, $\div$, sin, cos, exp, log, sqrt, tanh
Terminal set	constant, constant $\cdot$ variable
Stop criterion	1M solution evaluations or 1,000 generations

To effectively compare backends, we define various performance/energy measures. First, we generalize the traditional notion of the performance measure "GP operations per second (GPops/s)" [8] to allow for node derivative calculations, and we concisely name this measure "nodes per second (NPS)" to better align with our following measures. When a total number of node operations is needed, we use an estimate based on average population statistics [10].

From NPS, we relate performance to energy consumption with the "nodes per watt (NPW)" measure:

$$\text{NPW} \triangleq \frac{\text{NPS}}{\text{Total power (Watt)}} = \frac{\text{Total number of node operations}}{\text{Total energy (Joule)}}. \tag{1}$$

Note that, as with standard "performance-per-watt" measures, the units of time cancel, and we equivalently calculate work per unit energy. Separate from NPW, we define another performance-per-watt measure named "fitness per watt (FPW)," where we consider "performance" to be the self-explanatory "fitness per second (FPS)," and where the GP fitness measure is to be maximized:

$$\text{FPW} \triangleq \frac{\text{FPS}}{\text{Total power (Watt)}} = \frac{\text{Fitness}}{\text{Total energy (Joule)}}. \tag{2}$$

Overall, the NPS/FPS measures quantify forms of throughput, with FPS allowing one to draw conclusions about how solution quality relates to runtime, whereas the NPW/FPW measures additionally relate performance to energy, which is useful when considering operational costs. Note that all measures are to be maximized. Here, we measure energy/runtime via the Linux `perf` tool.

Lastly, we note that we can meaningfully generalize the above measures to represent aggregate values across different datasets as long as we compute only one ratio involving total sums over all datasets, similar to how it has been documented for the "floating-point operations per second (FLOPS)" measure [48].

6 Results

All of the following results except those described for Fig. 2 are given within the context of the first deployment strategy listed in Sect. 4, i.e., scaling the outputs of learned solutions based on re-evaluation with the STL backend.

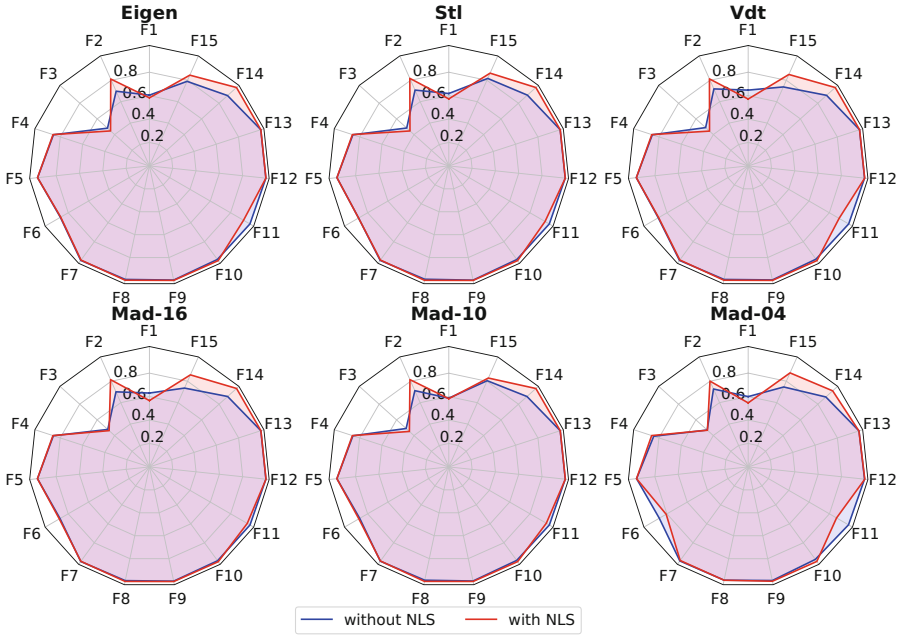

Fig. 1. Median R^2 scores on test sets, for each backend, for each problem

First, we consider the median test R^2 fitness scores for all backends, for all problems, without/with local search using nonlinear least squares (NLS), as depicted by the radar charts in Fig. 1. Immediately, we see that the shapes and area of the polygons given for all the backends appear largely similar, which intuitively suggests that the different implementations of the relevant function primitives do not cause catastrophic discrepancies in solution quality. This result is not too surprising for the Eigen, STL, VDT, and MAD-16 backends, since the relative difference in function outputs between these backends is more or less negligible, but this result is remarkable for our proposed MAD-10 and MAD-04 implementation sets, due to the presence of significantly higher amounts of function error, which can become further pronounced via arbitrary function compositions and the use of automatic differentiation for local search.

In Table 4, we further compare each MAD backend against Eigen—the default backend of Operon—using the Mann-Whitney U statistical test [34] between R^2 test set values obtained for each problem, for runs without/with local search.

Table 4. Statistical significance of Eigen's test R^2 values being "statistically greater" than the MAD backends, and relative percentage differences between median R^2 values. White/black squares indicate runs without/with local search.

	Mad-16				Mad-10				Mad-04			
F	□	□	■	■	□	□	■	■	□	□	■	■
F_1	8.4e−01	4.31	7.9e−02	−2.64	1.9e−01	−2.91	4.6e−01	0.17	5.5e−01	−0.76	1.4e−03	−6.07
F_2	8.3e−01	0.45	6.3e−01	0.35	8.4e−01	2.07	6.8e−01	0.40	9.8e−01	4.21	3.1e−01	−1.38
F_3	4.5e−01	−0.75	6.4e−01	3.62	7.8e−01	1.79	5.7e−01	1.37	4.2e−01	−2.32	9.7e−01	4.72
F_4	4.9e−01	−0.01	2.2e−01	0.00	1.0e−02	−0.12	7.2e−02	0.00	1.6e−25	−1.72	9.5e−01	0.02
F_5	8.2e−01	0.02	2.7e−01	−0.01	5.2e−01	0.01	3.9e−01	−0.01	1.8e−03	−0.32	5.4e−19	−0.33
F_6	6.6e−01	0.00	8.1e−01	0.14	3.6e−02	−0.26	3.8e−01	−0.11	1.9e−08	−0.61	2.9e−05	−0.73
F_7	6.4e−01	0.34	8.2e−01	1.47	4.1e−01	0.03	8.3e−01	1.37	2.6e−01	−0.29	2.6e−02	−7.66
F_8	5.7e−01	0.10	7.8e−01	0.08	1.3e−01	−0.23	7.4e−01	0.09	3.6e−02	−0.34	8.5e−10	−0.98
F_9	4.2e−01	−0.02	7.3e−01	0.03	2.6e−01	−0.09	6.1e−01	0.02	1.2e−06	−0.60	9.1e−17	−0.26
F_{10}	8.9e−01	0.22	4.7e−01	0.00	2.4e−01	−0.09	7.3e−01	−0.02	8.7e−05	−1.28	2.3e−13	−0.33
F_{11}	3.2e−01	−0.02	9.4e−01	3.88	7.1e−01	0.00	9.5e−01	3.58	8.4e−05	−0.32	2.0e−02	−6.23
F_{12}	1.4e−01	−0.06	1.0e+00	0.00	2.4e−01	−0.02	4.1e−02	0.00	2.0e−02	−0.21	1.2e−13	−0.02
F_{13}	1.4e−02	−0.02	9.9e−01	0.00	2.1e−01	0.00	9.4e−01	0.00	2.0e−13	−0.63	1.8e−30	−0.72
F_{14}	7.8e−01	0.05	4.7e−01	0.00	4.9e−01	0.07	4.5e−01	0.00	2.4e−02	−0.69	1.2e−06	−3.07
F_{15}	2.0e−01	−6.84	5.6e−01	1.35	4.4e−01	1.95	1.6e−01	−2.11	1.6e−01	−5.57	8.2e−01	3.48

The test operates under the null hypothesis that the R^2 values have the same underlying distribution, and under the alternate hypothesis that Eigen's results are "statistically greater," i.e., that Eigen delivers better R^2 values. In addition to p-values (shown in white cells), we list the relative percentage difference between Eigen's median test R^2 score and the median test R^2 score of each MAD backend (shown in gray cells), where a negative value can be interpreted as the relevant MAD backend typically performing worse than Eigen.

For the MAD-16 and MAD-10 backends, we observe many p-values greater than 0.05 and many non-negative relative percentage differences, which allows us to conclude that the use of these implementation sets often allows for similar or better fitness scores than the standard Operon system. For the MAD-04 backend, the analysis needs to be more nuanced. First, although we see many smaller p-values and many negative relative differences, the median relative difference is always greater than −7.7% and on average greater than −1.1%. Thus, the practical difference in fitness may be negligible depending on the dataset, but importantly, we are not yet considering the trade-offs involving performance and power/energy that may be possible when allowing for the MAD-04 backend.

In Table 5, we consider various average measures for the entire set of benchmark problems, which identify in their own regard that the MAD-04 backend is generally more efficient than other backends—consider the bold values. (We can safely compute an average across all problems by computing only one ratio, as described in Sect. 5.) For example, we see that the MAD-04 backend achieves the highest mean scores for almost all of the "nodes per second (NPS)," "nodes

per watt (NPW)," "fitness per second (FPS)," and "fitness per watt (FPW)" measures defined in Sect. 5, often achieving values between 2–10% greater than Eigen. This sharp contrast in results between Tables 4 and 5 illustrates how considering only plain fitness scores can lead to incomplete conclusions.

Table 5. Various average measures for the set of benchmark problems. See Sect. 5 for details on NPS, NPW, FPS, and FPW. Runtime and energy are given in seconds and Joules, respectively. Size is for final models. ΔR^2 is the mean difference in test R^2 values when compared to Eigen, and ΔR^2 (Rel.) is a relative percentage equivalent. The "NLS" subscript identifies runs with local search.

	Eigen	Stl	Vdt	Mad-16	Mad-10	Mad-04
NPS	2.12e+10	1.12e+10	2.05e+10	1.96e+10	1.99e+10	**2.23e+10**
NPS_{NLS}	**3.07e+10**	2.39e+10	3.07e+10	3.00e+10	3.02e+10	3.01e+10
NPW	1.61e+08	1.00e+08	1.47e+08	1.39e+08	1.43e+08	**1.65e+08**
NPW_{NLS}	3.05e+08	3.08e+08	3.00e+08	2.90e+08	2.97e+08	**3.17e+08**
FPS	0.615	0.323	0.595	0.562	0.581	**0.639**
FPS_{NLS}	0.488	0.383	0.496	0.480	0.484	**0.499**
FPW	4.6e−03	2.9e−03	4.3e−03	4.0e−03	4.2e−03	**4.7e−03**
FPW_{NLS}	4.8e−03	4.9e−03	4.9e−03	4.6e−03	4.8e−03	**5.3e−03**
Runtime	1.35	2.59	1.39	1.47	1.42	**1.28**
$\text{Runtime}_{\text{NLS}}$	1.73	2.22	1.71	1.78	1.76	**1.67**
Energy	178.17	307.25	193.15	207.02	197.25	**172.60**
$\text{Energy}_{\text{NLS}}$	174.22	190.32	174.30	183.46	178.78	**158.17**
Size	29.5	29.7	29.4	29.7	**29.2**	30.0
Size_{NLS}	38.6	38.7	38.4	38.9	38.6	**37.8**
ΔR^2	0	7.08e−03	−1.64e−03	−2.68e−03	−6.20e−03	−1.09e−02
ΔR^2_{NLS}	0	6.86e−03	2.37e−03	7.46e−03	5.16e−03	−1.18e−02
ΔR^2 (Rel.)	0	0.855	−0.197	−0.324	−0.749	−1.32
ΔR^2_{NLS} (Rel.)	0	0.812	0.281	0.883	0.611	−1.40

Most importantly, we emphasize that the values reported in Table 5 are for a CPU, whereas our ultimate intention is to leverage specialized hardware, which can likely exhibit even more attractive performance and power/energy trade-offs [11]. For example, the aggregate nodes-per-second (NPS) values for Operon in this study are frequently on the order of 20–30 billion, whereas our discussion in Sect. 4 illustrates how leveraging the proposed MAD backends with an FPGA could potentially allow for NPS values in the hundreds of billions, if not trillions. Separately, it has been widely shown that FPGA devices can often infer significant power/energy benefits [37,44,49,51]. Therefore, with considerable potential for improving both performance and power/energy when employing an FPGA

device, it is clear that any minor differences in fitness scores caused by the proposed MAD implementations may be far outweighed by practical improvements in runtime and operational costs when using such systems. At the very least, these results should validate the introduction and future consideration of alternative floating-point primitives for improving the computational efficiency of GP, which was the overarching goal of this work.

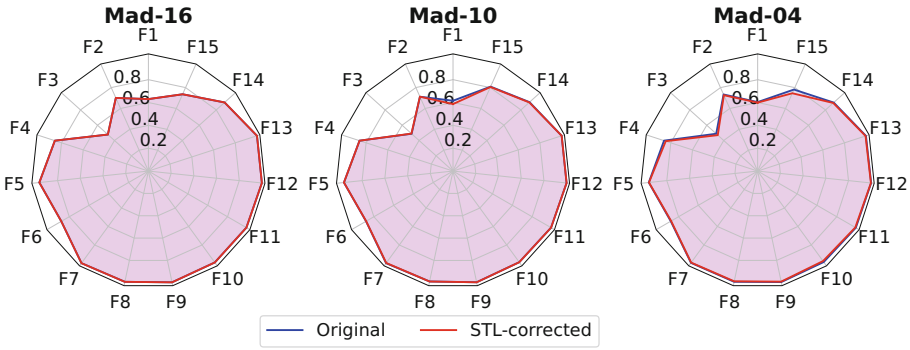

Fig. 2. Comparing median test R^2 scores between both deployment strategies

To conclude this section, we briefly consider the second deployment strategy listed in Sect. 4, i.e., training and deploying with the same computational backend. In Fig. 2, we examine the MAD-based runs without local search. Overall, the figure shows that median test R^2 values when employing the same backend used during training are very similar or better when compared to median R^2 values for STL-corrected models. (Trends for mean test R^2 values and for local search were similar.) Therefore, we establish that GP inference may also benefit from alternative backends, for the purposes of better computational efficiency.

7 Conclusion

Can we allow for the use of rough approximations of floating-point primitives? Our study indicates yes, with the performance and power/energy benefits being notable for CPU devices, yet likely very significant for specialized hardware, e.g., with FPGA devices. Our study also indicates that such alternative primitives can potentially be leveraged during both training and inference, thus allowing for continued performance/energy benefits. Future work should explore how simple we can make the numerical system in order to extract additional efficiency. For example, can we meaningfully utilize alternative primitives with 8/16-bit floating-point encodings? Separately, although similar program sizes resulted from the use of alternative primitives, future work should consider possible differences in interpretability. Overall, this work marks a considerable initial step toward improving the computational efficiency of GP systems through the use

of alternative implementations of standard floating-point function primitives. Furthermore, our findings help lay the groundwork for potential advancements in specialized hardware for GP, which can likely offer notable performance and power/energy advantages over traditional general-purpose CPU/GPU systems.

Disclosure of Interests. The authors have no competing interests to declare.

References

1. Acun, B., et al.: Carbon explorer: a holistic framework for designing carbon aware datacenters. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, pp. 118–132 (2023)
2. Altenberg, L.: The evolution of evolvability in genetic programming. In: Kinnear, K. (ed.) Advances in Genetic Programming, vol. 1, pp. 47–74. MIT Press (1994)
3. Bakurov, I., Haut, N., Banzhaf, W.: Sharpness minimization in genetic programming. In: Winkler, S., et al. (eds.) Genetic Programming - Theory and Practice XXI, p. forthcoming. Springer (2025). https://arxiv.org/abs/2405.10267
4. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction. Morgan Kaufmann, Estes Park (1998)
5. Bashir, N., et al.: Enabling sustainable clouds: the case for virtualizing the energy system. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2021, pp. 350–358. Association for Computing Machinery, New York (2021)
6. Brooks, T.F., Pope, D.S., Marcolini, M.A.: Airfoil self-noise and prediction. Technical report 1218, NASA (1989)
7. Burlacu, B., Kronberger, G., Kommenda, M.: Operon C++: an efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO 2020, pp. 1562–1570. Association for Computing Machinery, New York (2020)
8. Chitty, D.M.: Faster GPU-based genetic programming using a two-dimensional stack. Soft. Comput. **21**(14), 3859–3878 (2017)
9. Chromczak, J., et al.: Architectural enhancements in Intel Agilex FPGAs. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2020, pp. 140–149. Association for Computing Machinery, New York (2020)
10. Crary, C., Burlacu, B., Banzhaf, W.: PPSN 2024 Conference Software Code (2024). https://github.com/christophercrary/conference-ppsn-2024
11. Crary, C., Piard, W., Stitt, G., Bean, C., Hicks, B.: Using FPGA devices to accelerate tree-based genetic programming: a preliminary exploration with recent technologies. In: Pappa, G., Giacobini, M., Vasicek, Z. (eds.) EuroGP 2023. LNCS, vol. 13986, pp. 182–197. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-29573-7_12
12. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., Fast, A.: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
13. Díaz-Álvarez, J., Castillo, P.A., de Vega, F.F., Chávez, F., Alvarado, J.: Population size influence on the energy consumption of genetic programming. Meas. Control **55**(1–2), 102–115 (2022)

14. Fernández de Vega, F., Díaz, J., García, J.Á., Chávez, F., Alvarado, J.: Looking for energy efficient genetic algorithms. In: Idoumghar, L., Legrand, P., Liefooghe, A., Lutton, E., Monmarché, N., Schoenauer, M. (eds.) EA 2019. LNCS, vol. 12052, pp. 96–109. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45715-0_8
15. Foret, P., Kleiner, A., Mobahi, H., Neyshabur, B.: Sharpness-aware minimization for efficiently improving generalization. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, 3–7 May 2021. OpenReview.net (2021). https://openreview.net/forum?id=6Tm1mposlrM
16. Friedman, J.H.: Multivariate adaptive regression splines. Ann. Stat. **19**(1), 1–67 (1991)
17. Funie, A.I., Grigoras, P., Burovskiy, P., Luk, W., Salmon, M.: Run-time reconfigurable acceleration for genetic programming fitness evaluation in trading strategies. J. Signal Process. Syst. **90**(1), 39–52 (2018)
18. Funie, A.I., Salmon, M., Luk, W.: A hybrid genetic-programming swarm-optimisation approach for examining the nature and stability of high frequency trading strategies. In: 2014 13th International Conference on Machine Learning and Applications, pp. 29–34 (2014)
19. García-Martín, E., Rodrigues, C.F., Riley, G., Grahn, H.: Estimation of energy consumption in machine learning. J. Parallel Distrib. Comput. **134**, 75–88 (2019)
20. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (CSUR) **23**(1), 5–48 (1991)
21. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: International Conference on Machine Learning, pp. 1737–1746. PMLR (2015)
22. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 6th edn. Morgan Kaufmann Publishers Inc., San Francisco (2017)
23. Intel: Intel Agilex™M-Series FPGA and SoC FPGA Product Table [Online] (2015). https://cdrdv2.intel.com/v1/dl/getContent/721636
24. Jia, H., Verma, N.: Exploiting approximate feature extraction via genetic programming for hardware acceleration in a heterogeneous microprocessor. IEEE J. Solid-State Circuits **53**(4), 1016–1027 (2018)
25. Keijzer, M.: Scaled symbolic regression. Genet. Program Evolvable Mach. **5**, 259–269 (2004)
26. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multitask reinforcement learning. Evol. Comput. **26**(3), 347–380 (2018)
27. Kommenda, M., Burlacu, B., Kronberger, G., Affenzeller, M.: Parameter identification for symbolic regression using nonlinear least squares. Genet. Program Evolvable Mach. **21**(3), 471–501 (2020)
28. Kordon, A.K., Castillo, F.A., Smits, G., Kotanchek, M.E.: Application issues of genetic programming in industry. In: Yu, T., Riolo, R., Worzel, B. (eds.) Genetic Programming - Theory and Practice III, pp. 241–258. Springer, Boston (2006). https://doi.org/10.1007/0-387-28111-8_16
29. Koza, J.R., Bennett, F.H., Hutchings, J.L., Bade, S.L., Keane, M.A., Andre, D.: Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming. In: Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, FPGA 1998, pp. 209–219. Association for Computing Machinery, New York (1998)
30. Koza, J.: Genetic Programming - On Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)

31. La Cava, W., et al.: Contemporary symbolic regression methods and their relative performance. In: Advances in Neural Information Processing Systems, vol. 35, pp. 1–16 (2021)
32. Liu, J., Cai, J., Zhuang, B.: Sharpness-aware quantization for deep neural networks. arXiv:2111.12273 (2023)
33. Martín, E.G., Lavesson, N., Grahn, H., Boeva, V.: Energy efficiency in machine learning: a position paper. In: Annual Workshop of the Swedish Artificial Intelligence Society (2017). https://api.semanticscholar.org/CorpusID:44010140
34. McKnight, P.E., Najab, J.: Mann-Whitney U Test. The Corsini Encyclopedia of Psychology, pp. 1–1 (2010)
35. Mittal, S.: A survey of techniques for approximate computing. ACM Comput. Surv. (CSUR) **48**(4), 1–33 (2016)
36. Moroz, L.V., Walczyk, C.J., Hrynchyshyn, A., Holimath, V., Cieśliński, J.L.: Fast calculation of inverse square root with the use of magic constant - analytical approach. Appl. Math. Comput. **316**, 245–255 (2018)
37. Nurvitadhi, E., et al.: Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, pp. 5–14. Association for Computing Machinery, New York (2017)
38. Oliphant, T.E., et al.: Guide to Numpy, vol. 1. Trelgol Publishing, USA (2006)
39. Pagie, L., Hogeweg, P.: Evolutionary consequences of coevolving targets. Evol. Comput. **5**, 401–418 (1997)
40. Patros, P., Spillner, J., Papadopoulos, A.V., Varghese, B., Rana, O., Dustdar, S.: Toward sustainable serverless computing. IEEE Internet Comput. **25**(6), 42–50 (2021)
41. Piparo, D., Innocente, V., Hauth, T.: Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions. J. Phys. Conf. Ser. **513**(5), 052027 (2014). https://dx.doi.org/10.1088/1742-6596/513/5/052027
42. Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 204–217. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36599-0_19
43. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises UK Ltd, Egham (2008)
44. Putnam, A., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro **35**(3), 10–22 (2015)
45. Real, E., et al.: AutoNumerics-Zero: automated discovery of state-of-the-art mathematical functions. arXiv preprint arXiv:2312.08472 (2023)
46. Sekanina, L.: Evolutionary algorithms in approximate computing: a survey. arXiv preprint arXiv:2108.07000 (2021)
47. Sidhu, R.P.S., Mei, A., Prasanna, V.K.: Genetic programming using self-reconfigurable FPGAs. In: Lysaght, P., Irvine, J., Hartenstein, R. (eds.) FPL 1999. LNCS, vol. 1673, pp. 301–312. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-48302-1_31
48. Smith, J.E.: Characterizing computer performance with a single number. Commun. ACM **31**(10), 1202–1206 (1988)
49. Stitt, G., Gupta, A., Emas, M.N., Wilson, D., Baylis, A.: Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, pp. 173–182. Association for Computing Machinery (2018)

50. Strubell, E., Ganesh, A., McCallum, A.: Energy and policy considerations for modern deep learning research. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 13693–13696 (2020)
51. Tan, T., Nurvitadhi, E., Shih, D., Chiou, D.: Evaluating the highly-pipelined Intel Stratix 10 FPGA architecture using open-source benchmarks. In: 2018 International Conference on Field-Programmable Technology (FPT), pp. 206–213 (2018)
52. Vladislavleva, E.J., Smits, G.F., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. IEEE Trans. Evol. Comput. **13**(2), 333–349 (2009)
53. Wilson, G., Banzhaf, W.: Linear genetic programming GPGPU on microsoft's Xbox 360. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), pp. 378–385. IEEE Press (2008)
54. Yeh, I.C.: Modeling of strength of high-performance concrete using artificial neural networks. Cem. Concr. Res. **28**(12), 1797–1808 (1998)
55. Zhang, H., Chen, Q., Xue, B., Banzhaf, W., Zhang, M.: Sharpness-aware minimization for evolutionary feature construction in regression. IEEE Trans. Pattern Anal. Mach. Intell. (submitted). https://arxiv.org/abs/2405.06869