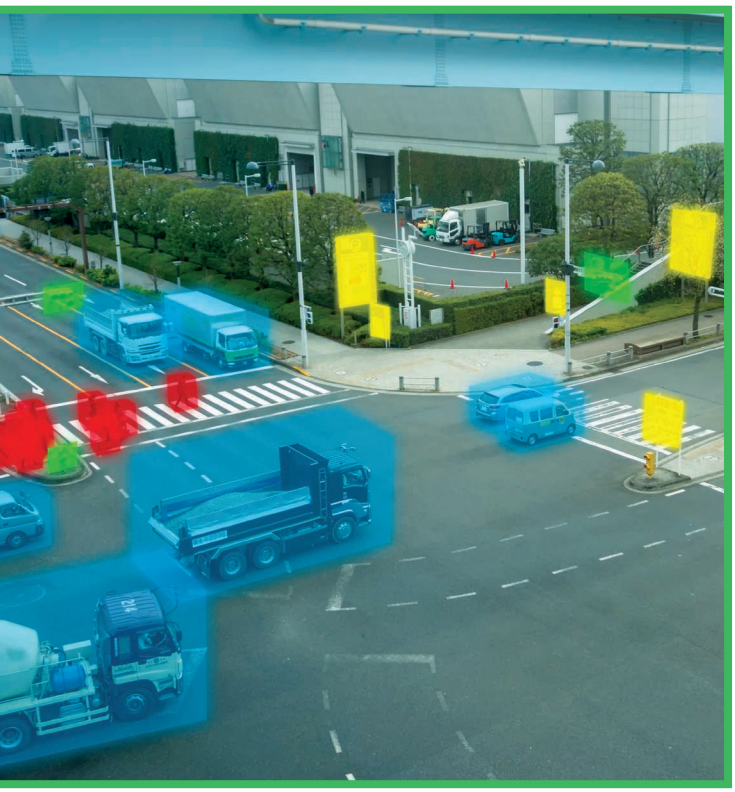
Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi,
Sergio Saponara, and Benoît Dupont de Dinechin

Novel Arithmetic in Deep Neural Network Signal Processing for Autonomous Driving

Challenges and opportunities



©SHUTTERSTOCK.COM/MONOPOLY919

Digital Object Identifier 10.1109/MSP.2020.2988436
Date of current version: 24 December 2020

This article focuses on the trends, opportunities, and challenges of novel arithmetic for deep neural network (DNN) signal processing, with particular reference to assisted- and autonomous driving applications. Due to strict constraints in terms of the latency, dependability, and security of autonomous driving, machine perception (i.e., detection and decision tasks) based on DNNs cannot be implemented by relying on remote cloud access. These tasks must be performed in real time in embedded systems on board the vehicle, particularly for the inference phase (considering the use of DNNs pretrained during an offline step). When developing a DNN computing platform, the choice of the computing arithmetic matters. Moreover, functional safe applications, such as autonomous driving, impose severe constraints on the effect that signal processing accuracy has on the final rate of wrong detection/decisions. Hence, after reviewing the different choices and tradeoffs concerning arithmetic, both in academia and industry, we highlight the issues in implementing DNN accelerators to achieve accurate and low-complexity processing of automotive sensor signals (the latter coming from diverse sources, such as cameras, radar, lidar, and ultrasonics). The focus is on both general-purpose operations massively used in DNNs, such as multiplying, accumulating, and comparing, and on specific functions, including, for example, sigmoid or hyperbolic tangents used for neuron activation.

Introduction

The use of DNNs as a general tool for signal and data processing is increasing in both the automotive industry and academia, proposing a set of algorithms for most of the autonomous driving tasks. The effort in computing these artificial intelligence (AI) algorithms is an open challenge in the field of computing platforms. In particular, when considering strict requirements, such as lowering the power consumption, maximizing the throughput, and minimizing the latency, the computational complexity becomes more critical. Moreover, with modern achievements in sensor components, the complexity and requirements scale further, with data coming in higher volumes and dimensions and at higher speeds [1].

This survey work is focused on the trends, opportunities, and challenges of the adoption of DNN signal processing techniques for autonomous driving and the needs of signal processing acceleration as well as the relevant computing arithmetic. Indeed, autonomous driving is a safety-critical application, as also specified in functional safety standards, such as International Organization for Standardization 26262, with strict requirements in terms of real time (both throughput and latency) [1], [2]. In levels $\mathcal{L}1$ and $\mathcal{L}2$ of the Society of Automotive Engineers autonomous driving scale [3], only assistance to the human driver is needed. Therefore, signal processing based on deterministic algorithms is still enough; e.g., fast Fourier transform-based processing of frequency-modulated continuous-wave radar was done in [1]. Instead, for high autonomous driving levels, from $\mathcal{L}3$ to $\mathcal{L}5$, the complexity of the scenario and need for signal processing, not only for sensing but for localization, navigation, decisions, and actuation, is so high that in recent state-of-the-art DNNs, signal processing is proposed to be used on board [1], [2], [4], [5]. This trend is confirmed by the rise of the Autonomous Systems Initiative within the IEEE Signal Processing Society [6]. DNNs have reached state-of-the-art status in several signal processing domains, such as image processing, segmentation, classification, tracking [7]–[10], computer vision [11], and related areas [12]–[14].

In the automotive field, while sensor raw data processing (from cameras, lidar, radar, and ultrasonics) can be still performed using classical signal processing techniques, DNNs are emerging as more appropriate solutions to solve complex and high-level tasks, such as data fusion, classification, and planning in harsh, unstructured, and continuously changing environments. Tasks such as scene understanding (e.g., image segmentation, region-of-interest extraction, subscene classification, and so on) must be done on board vehicles since cloud-based computing scenarios (where signal processing is done on remote cloud servers and on board, there is only a client unit generating requests to the server) suffer from several issues: privacy, authentication, integrity, connection latency and contention, and even communication unavailability in uncovered areas (highway tunnels, etc.). Onboard DNN signal processing can be done only if a low-computational complex algorithm is used and performing hardware (HW) is adopted. Hence, onboard computing units for DNNs should be optimized in terms of the ratio between the signal processing throughput performance and resources (memory, bandwidth, power consumption, and so on) [15]–[17]. This is the trend that big players, including Google, NVIDIA, and Intel, are following as they try to enter the autonomous driving market. Tesla recently announced its full self-driving (FSD) chip. This concept is also the core of the automotive stream in the Horizon 2020 European Processor Initiative (EPI) (embedded high-performance computing for autonomous driving, with BMW Group as the technology's main end user [17]), where this article's authors are involved.

To address the preceding issues, new computing arithmetic styles are appearing in the state of the art [18]–[26] to overcome the classic fixed-point (INT) versus IEEE Standard 754 floating-point duality in the case of embedded DNN signal processing. Just as an example, Google is proposing Brain Floating-Point Format (BFLOAT) 16, which is equivalent to a standard single-precision floating-point value with a truncated mantissa field. BFLOAT16 is supported in the Google Cloud tensor-processing unit (TPU) and TensorFlow and Intel AI processors. Intel is also proposing Flexpoint [18], [19], a 16-bit-block floating-point format aiming to replace Float32. NVIDIA's Turing architecture supports, in its tensor cores, Float16 to Float16 or Float32 matrix multiply–add operations as well as integer 4 (INT4) or INT8 to INT32 matrix multiply–add operations, the latter for inferencing workloads that tolerate quantization [24]. The Tesla FSD chip exploits a neural processing unit using eight-by-eight-bit integer multiplication and 32-bit integer addition. Transprecision computing for DNNs is also proposed in the state of the art by academia [20] and industry, e.g., IBM and Greenwaves in [21]. Recently, a novel way to represent real numbers, called *Posit*, has been proposed [25], [26]. Basically, the Posit format can be thought of as a compressed floating-point representation, where more mantissa bits are used for numbers around one, with fewer stepping away from one, within a fixed-length format with variable-size fields (the exponent bits adapt accordingly to maintain the format fixed in length).

Review of state-of-the-art DNN signal processing in autonomous driving

Autonomous driving is deeply bounded to vehicle navigation, including vehicle self-localization, motion, mapping, and interaction. A relevant survey on trends and technologies for autonomous driving is presented in [27]. The localization task is aimed at knowing the vehicle's pose (position and orientation) as it is referred to a relative or absolute coordinate system. Traditional approaches to localization include satellite communication, such as GPS. However, these are typically weak radio signals that can be easily occluded by trees and buildings in a metropolitan scenario. There exist other types of equipment, such as inertial measurement units, that, combined with GPS, real-time kinematics, and Kalman-based predictors, can solve this problem, but they increase the implementation cost. Since the task of constantly knowing the vehicle's position is critical, one cannot rely only on these signals.

The mapping task introduces a further level of context awareness. With a map-matching approach, a vehicle is able to know not only its position but its surroundings. An important mapping technique is simultaneous localization and mapping (SLAM) [28], which enables a vehicle to bypass or minimize the need for satellite navigation. SLAM considers the surroundings as a probability distribution of points rather than a snapshot of the context in time, building a world model by making use of lidar sensors or similar devices. The typical output of these sensors is point clouds that represent the surrounding environment and must be processed to give more information about the area. In [29], a way to classify lidar images using DNNs is presented. In [30], a benchmark challenge for DNNs for the German Traffic Road Sign Recognition

Benchmark (GTRSB) is proposed, and in [31], there are some advanced DNN techniques, such as data augmentation and region-of-interest extraction, to maximize DNN recognition and detection accuracy, reaching top-level accuracy on road sign recognition and detection benchmarks. Moreover, with advanced developments in computer vision, vehicles can be equipped with cameras whose signals can be processed by DNNs as well. For example, in [32]–[34], a semantic segmentation of city landscapes challenge is presented, providing benchmarks for DNNs to prove their ability to identify the main components of a road (such as lanes, other vehicles, and pedestrians) from image or video signals. On the industry side, with the advent of companies including Tesla and Google's Waymo, the use of DNNs in processing lidar and camera signals has become more central.

Low-precision DNNs

Academia and industry have proposed multiple solutions to the problem of reducing the number of bits used to represents DNNs' weights, compressing the size from 32 to 16, eight, four, and even one bit, resulting in little to no degradation in performance when tested with common DNN tasks and benchmarks. As an emerging trend in the state of the art, the literature is starting to explore the possibility of using the newly introduced Posit representation to halve the weights' size while maintaining the same accuracy and to further reduce the weights' size while sacrificing little to no DNN precision. A very interesting work has been presented in [35], where network weights have been binarized, dramatically reducing the network footprint and increasing the training and inference speed. On the industry side, NVIDIA has led the reduction of weight bits with its TPU, introducing integer weight types, such as eight- and four-bit integers. In [36], a novel method is introduced to train neural networks (NNs) with extremely low precision (e.g., one bit), weights, and activations at runtime. In [37], the authors studied the training of NNs using low-precision fixed-point computations and evaluated the impact of different rounding techniques.

The aim of this article is to develop an NN accelerator based entirely on Posits, while also embedding look-up tables (LUTs) for low-bit Posits, such as four–12-bit Posits. In this way, we ensure a homogeneity of representations that is lost in the NVIDIA approach due to the discontinuity introduced when switching from floating-point half precision to eight- or four-bit integers.

Alternative representations for real numbers

In this section, we review the most interesting representation for real numbers, which could be used as an alternative to the floating-point representation (the IEEE 754 standard, 2008, which will be referred to simply as *Float* from now on). In the following, we will use a homogeneous representation for the different number representation "Type Bits[,Exp]," where Type is the name of the representation (Float, Posit, and Fixed), Bits is the number of bits, and Exp is the number of bits used for the exponent. For fixed-point Exp representations, the scaling factor to be applied to the number is considered to be a signed integer (e.g., Fixed16,8 represents a value with eight bits in the integer part and eight bits in the fractional part). For Float when Exp is missing, the standard value is assumed: 11, eight, and five for Float64, Float32, and Float16, respectively, corresponding to binary64, binary32, and binary16 of the IEEE standard.

BFLOAT16

The research on DNNs has demonstrated that 16-bit Floats could be enough for many classification problems. From this research came the idea to give HW support to the standard half precision (Float16,5) too, in addition (or as an alternative) to Float32. The problem is that pretrained DNN models are usually available with Float32, and thus, lowering them to five bits of exponent could introduce alterations to the classification and affect the overall classification performance. For this reason, the BFLOAT16 format (Brain Float 16-bits, namely, Float16,8 in the present notation) has been recently introduced with eight bits of exponent instead of five. Having the same size of the exponent of Float32, the use of BFLOAT16 introduces a loss of numerical precision but no loss of dynamic range. Also, the conversion with Float32 is bitwise.

Flexpoint

Flexpoint numbers [18] are characterized by a shared tensor exponent used for all number representations in a given NN layer (e.g., a 16-bit Flexpoint plus a five-bit shared exponent). Moreover, the magnitude of the common exponent is dynamically adjusted according to the required numerical range during training. The Flexpoint approach, although interesting and powerful, cannot be used as a drop-in replacement for Floats: changes are required to the DNN software (SW) libraries. This also makes the reuse of pretrained DNNs cumbersome.

Type-3 uniform numbers: Posits

Type-3 uniform numbers are the third proposal of universal numbers offered, again, by Gustafson. They can be exact (Valids) or inexact (Posits). Posits are particularly interesting because they are a drop-in replacement for Floats, while Valids are not. Posits will be presented and deeply investigated in the next section. Before that, we discuss, in the next two subsections, two further representations that are somehow related to Posits.

Universal coding of real numbers using bisection

The bisection method proposed by Lindstrom in [38] is based on Elias codes. It encodes each real number in a binary string based on bisecting intervals, starting from the base interval $(-\text{inf}, +\text{inf})$. Each bit of the string is the result of a comparison with a value contained in a given interval. The framework proposed as universal coding enables building new number systems by defining a generator function to produce the various intervals and a so-called refinement operator to compute the average value between two numbers. Theoretically speaking, this encoding is very interesting due to the possibility to rapidly prototype and verify the representation. However, the encoding is quite

inefficient, involving elaborate expressions in its computations, thus becoming HW-unfriendly. This suggests that this particular encoding is not so interesting in the high-performance HW accelerator topic discussed here, although Posit numbers can also be generated using this powerful encoding technique.

Logarithmic numbers and the Kulisch accumulator

As pointed out by Johnson, a researcher at Facebook AI Research, the problem in [39] with floating-point operations in HW is that the transistors needed to perform multiplication and division occupy the main part of the floating-point unit (FPU), which is significantly more complex than for addition/subtraction. To overcome this problem, the logarithmic number system (LNS) was proposed decades ago in [40]. The LNS consists of representing a number as $y = 2^x$, i.e., in a pure logarithmic way. This makes multiplication and division a matter of just adding and subtracting logarithmic numbers.

However, this requires huge HW LUTs to compute the sum or difference of two logarithmic numbers [39]. This has been one of the main bottlenecks for the format since handling these tables can be more expensive than basic HW multipliers. To avoid common fused multiplication and adding complexity, the Kulisch accumulation can be used. The idea is to not accumulate with a floating-point-type but instead, maintain an accumulator in a fixed-point type. As a drawback, this approach leads to a significant increase in logic circuitry and power consumption, due to the bit count requirements of the Kulisch accumulator. Although this approach is really promising and can be combined with the Posit philosophy, it has not yet been demonstrated that logarithmic numbers are more effective than Floats for DNNs. Thus, more research is clearly needed before resorting to this solution.

A deeper investigation of Posits

Posit numbers have been proposed by Gustafson in [26]. The format is a fixed-length representation for real numbers, and it has two parameters: the total number of bits (totbits) and the number of exponent bits (esbits). It is composed of a maximum of four fields (see Figure 1):

- one-bit sign field S
- variable-length regime field R (1.rebits)
- exponent field E, which has a predetermined maximum length of esbits (field E can even be absent)
- variable-length fraction field F (it can be absent, too).

With the adopted notations, PositN,E refers to a Posit with N total bits and E esbits.

Both the total number of bits and the maximum size of the exponent field (esbits) are decided empirically a priori, depending on the application. These two lengths are those that fully characterize the Posit representation. The regime field length is determined by the number of consecutive zeros after the sign bit ended by one or, vice versa, by the number of consecutive ones ended by one zero. In the former case, the regime value is negative. After having determined the regime length, the associated value k can be retrieved according to the procedure illustrated in Figure 2. The bits that follow the regime field are, if present, the ones associated to the exponent. Their number can be, at maximum, equal to the esbits (the a priori predetermined maximum number of exponent bits). When the field is missing, the exponent e is assumed to be zero. When fewer bits than esbits are present, the value of e can be obtained by filling the missing bits with zeros before decoding it (see Figure 3).

If there are additional bits after the exponent field, they are the ones associated to the fractional part of the mantissa. If the Posit is negative (the first bit is equal to one) before decoding it to retrieve k, e, and f, the two's complement of its remaining bits must be computed. Therefore, let p be the integer represented by the Posit bit string, k the correspondent integer indexed by the regime bits into a run-length table (see Figure 2), e the unsigned integer associated to the exponent field E, and $f = 0.f_1 f_2 \ldots f_n$ [the fractional part of the mantissa m $(m = 1 + f)$, associated to the F field]; the expression that maps the bit set to the real value is

$$x = \begin{cases} 0, & \text{if } p = 0 \\ \text{NaR}, & \text{if p} = -2^{(\text{totbits}-1)} \\ \text{sign}(p) \times u^k \cdot 2^e \cdot (1+f), & \text{otherwise} \end{cases},$$

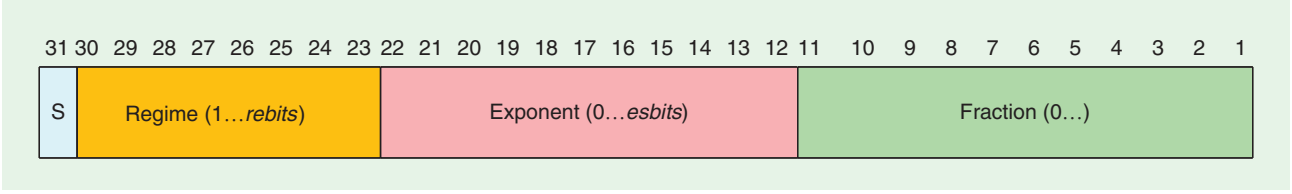


FIGURE 1. The 32-bit Posit data type.

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical Meaning, k	–4	–3	–2	–1	0	1	2	3

FIGURE 2. The mapping table between the regime bits and k value for a five-bit string. Amber bits represent the regime bits, and brown ones terminate the regime run.

where

$$u = 2^{2^{\text{esbits}}}.$$

Notably, it is possible to prove that for PositN,0, the numbers in the range $[-1, 1]$ are encoded as signed fixed points across $N-1$ bits. This property is important for the level-1 (L1) operations discussed later. Figure 3 provides an example of Posit16,3 (16 b, with a maximum of three exponent bits) and its decoding procedure.

Posit advantages over Floats and industrial adoption

As shown in [26], the main advantages of Posits over IEEE floating points are represented by less waste of representations (such as unique-zero and not-a-number bit configurations) and higher decimal accuracy when compared to the same bit length floating point. Moreover, the simplicity of the Posit number systems theoretically facilitates a more HW-friendly implementation, simplifying circuitry and thus reducing area occupation and power consumption. Even if the Posit format is relatively new, it has already attracted the attention of researchers from Facebook, IBM, Google, Microsoft, Intel, Bosch, Huawei, Fujitsu, Qualcomm, Kalray, Micron, Altair, Etaphase, Posit Research, Rex Computing, Stillwater Supercomputing, and Comma, as reported by Gustafson during a recent talk [41].

SW and HW implementations of Posits

SW implementations

Having SW implementation of Posit arithmetic is useful to test the applicability of the type to existing libraries and algorithms to compare performance against traditional floats and in the absence of proper HW support for Posit operations.

SoftPosit

This is a library endorsed by the Next Generation Arithmetic committee. Among its positive factors, it is multiplatform, supporting C, C++, Julia, and Python. However, it presents hard-coded Posit configurations and nonmodern implementation without templatized classes for the various configurations. It also lacks support for tabulated Posits.

Beyond Floating Point

Beyond Floating Point is one of the first C++ Posit arithmetic libraries developed. However, it is still incomplete and does not support Posit tabulation.

StillWater

StillWater is a complete library with modern C++ features and class templatization, although it is computationally heavy and missing Posit tabulation.

cppPosit

This library (available in [42]), developed by the authors of the present work, exploits some of the modern C++ features, such as templates (i.e., generic programming), and traits. It supports Posit tabulation and logic separation between the front-end interface and back-end underlying type used for computation: the front end is the Posit number expressed in its packed form, while the back end enables choosing different approaches for performing mathematical operations.

The library identifies four operational levels, with increasing computational cost. At level 1 (called L1), operations are just bit manipulation of the bits of the encoding. The cost is the same as integer operations performed in the arithmetic logic unit (ALU). At level 2 (L2), Posit data are extracted to fields (sign, regime, exponent, and fraction), with no need to compute the exponent completely. Computations are performed in these fields, and the cost includes encoding and decoding of the format. At level 3 (L3), we have the unpacked version that is completely built (including the sign, exponent, and fraction). In addition to L2 operations, here, there is the need to build the full exponent. At level 4 (L4), the unpacked version is used to perform the operations in either SW or HW floating point or using fixed-point representations. The most efficient level is, of course, L1 since it comprehends operations that only require bit manipulation of the Posit representation, which can be computed on existing ALUs without having to wait for Posit-processing units. Table 1 reports the most important L1 operations provided by the library. The library offers the possibility to use different back ends for Posit operations:

- a fixed-number back end (using a quire-like approach)
- a tabulated back end (see the section "The LUT Approach")

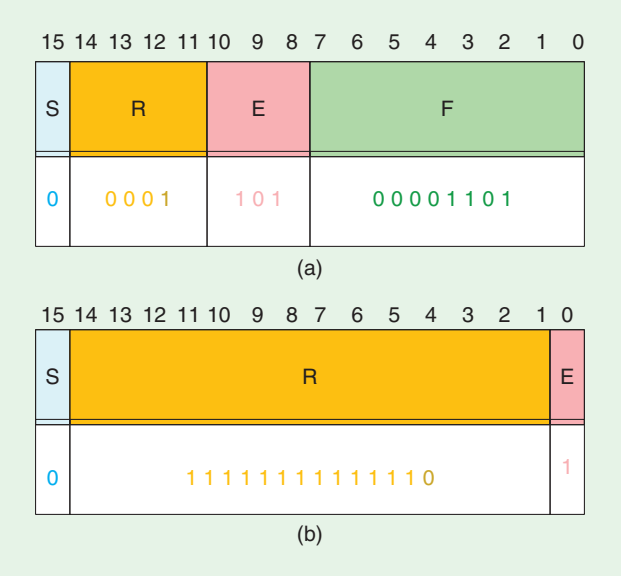


FIGURE 3. Two examples of Posit16,3, that is a 16-bit Posit with esbits = 3. (a) The associated real value is $+256^{-3} \cdot 2^{5} \cdot (1 + 13/256)$ (13/256 is the value of the fraction, and 1 + 13/256 is the value of the mantissa). The final value is therefore $+1.907348 \times 10^{-6} \cdot (1 + 13/256) \cong +2.0042 \times 10^{-6}$. (b) The associated real value is: $+256^{+12} \cdot 2^{4} \cdot (1 + 0)$ (since the fractional part of the mantissa is missing, we set it to zero). The final value is therefore $2^{96} \cdot 2^{4} \cong +1.2676506 \times 10^{30}$. The second example enables us to clarify that 1) the fractional part can be missing and 2) the exponent field can be shorter than its maximum size (in that case, the missing bits are assumed to be zero: the exponent four comes from reconstructed exponent field 100).

- a floating-point back end: either SW (SoftFloat) or HW (FPU).

Each L3 operation in the cppPosit library undergoes three different phases: 1) decode, 2) operation back end, and 3) encode. Each of these phases requires different functionalities in the processor architecture:

- *Decode*: mostly bit manipulation; the core function that is used here is the count-leading zeros (CLZ) built-in function.
- *Back end*:
  - *Fixed*: requires big-integer (64–128 bit) support
  - *Float*: requires an FPU
  - *SoftFloat*: requires 32/64-bit integer manipulations
- *Encode*: bitwise operations.

Table 2 shows a summary of the requirements support for two common architectures (both have been used for the benchmarks executed in the next sections; respectively, they are Intel i7560u and ARM Cortex A72). The two architectures do not differ in terms of HW requirements for the aforementioned phases. However, speaking about big-integer support, the Intel instruction set architecture (ISA) offers a single instruction (mulq) to perform a 64–128-bit integer multiplication; on the other hand, the ARM ISA requires the execution of two instructions.

HW implementations of the Posit processing unit

Some work has already been done to implement Posit units on field-programmable gate arrays (FPGAs) to provide efficient and optimized HW implementation of Posit arithmetic. In [44], an algorithmic flow and architecture generator for Posit numbers is proposed, including a Float-to-Posit converter unit and base arithmetic units. For the converter, the flow follows two major parts: floating-point unpacking and Posit construction. The first part works as any FPU, while the other determines the impact of the design on the HW. This has been implemented on a Xilinx Virtex-6 device, resulting in roughly 600 FPGA slices for a 32-bit Posit adder and 300 for a 16-bit Posit adder.

In [45], a Posit core generator, called *Poisson Generator* (*POSGEN*), is proposed. In addition, the FPGA design has been enriched with an extension of the Basic Linear Algebra Subprograms (BLAS) library for the Posit numbers, called *Posit BLAS*, to connect the FPGA through the Intel Open Computing Language libraries. The results show that the maximum frequency reached by the proposed implementation matches the state-of-the-art floating-point cores (FloPoCo) floating-point implementation. However, the area consumed by the POSGEN implementation is much higher than the FloPoCo one.

Another Posit arithmetic core, called the *Posit arithmetic unit*, generator is presented in [46], where generators for the Posit adder and multiplier are proposed. The design results show a reduction in the area occupation, referring to [44] for both the adder and multiplier as well as a reduction in power consumption for eight-bit Posits. For 16-bit Posits, the results are overturned in favor of the other implementation as well as for 32-bit Posits. Moreover, from the comparison between the Posit realization and the standard IEEE floating point, it is evident that a 32-bit Posit adder occupies less area and has a lower delay than a 32-bit Float adder. The 32-bit multiplier, instead, occupies the same area but with a higher delay. Finally, a 16-bit adder occupies a higher area with a higher delay.

In [47], another Posit arithmetic core generator has been introduced, called *PACoGEN*. The work presents different generators for HW description language adder/subtractor and multiplier/division cores. An interesting aspect of this implementation is the pipelined Posit arithmetic architecture, aimed at increasing the throughput of the unit and trying to produce a new result at each clock cycle (when at regime), making the three phases of an operation independent (Posit data extraction, core arithmetic process, and Posit construction). Design results show that the proposed implementation has a lower area (LUT) · period (ns) when compared to proposals in the literature, such as [46]. However, when the design is compared to standard floating-point ones, the results show that 32-bit Posit adder/multiplier units occupy more area than some 32-bit floating-point ones.

An accelerator for Posit-based BLAS operations is proposed in [48]. The work presents a modular framework for Posit arithmetic with the common three-step dataflow: Posit data extraction, operation, and construction. The implementation consists in a Posit adder, multiplier, and Posit accumulator. The proposed BLAS library enables vectorized operations, such as element-wise addition, subtraction, and multiplication, as well as the dot product and vector sum. Experimental results show a consistent speedup obtained when using the vectorized approach compared to an SW implementation.

When considering FPGA implementation of Posit arithmetic units, we need to consider the area occupation (thus, the power consumption) of the realized design and compare it to an FPU realization. Having a 32-bit HW Posit unit makes sense if the area of the realized Posit unit is less than the FPU one. If this does not hold, it still makes sense to have a 16-bit

Table 1. The cppPosit's most important L1 operations.

Operation	Approximated	Requirements
$2x$	No	esbits = 0
$x/2$	No	esbits = 0
$1-x$	No	esbits = 0, $x \in [-1, 1]$
$1/x$	Yes	esbits = 0
FastSigmoid	Yes	esbits = 0
FastTanh [43]	Yes	esbits = 0
FastELU	Yes	esbits = 0

The table shows whether the operation produces an exact or approximated result and reports the requirements to be fulfilled. For instance, notice how $1-x$ can be computed using fast bit manipulations only when $x \in [-1, 1]$.

Table 2. The requirements support of Intel and ARM for the cppPosit library.

Requirements	Intel Seventh Generation (Kaby Lake)	ARM v8
CLZ built in	✓	✓
Big integer	✓ (one single instruction)	✓ (two instructions needed)
FPU	✓	✓
Integer manipulation	✓	✓
Bitwise operations	✓	✓

HW Posit unit if its area is less than a 32-bit FPU one since 16-bit Posits achieve similar performance to 32-bit Floats in different application fields (in the "Benchmark Data Sets and Examples of Achievable Results" section, we show that in DNNs, even a Posit8 can match the performance of a Float32).

Posit-based DNNs for signal processing

Nonlinear activation functions are a very important part of DNNs. Their efficient implementation is therefore crucial. In the next sections, we will see how some widely used activation functions can be efficiently computed when using Posits.

DNN activation functions

In this section, we present special implementations of well-known mathematical functions and algorithms adapted to the Posit format. When considering these implementations, it is crucial to build them mostly with L1 operations (see the "cppPosit" section).

Sigmoid

The sigmoid function

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

has a very efficient approximation when using a Posit format with zero exponent bits, consisting only in a manipulation of the representation's bits. This discovery is due to Yonemoto and Gustafson [26]. Although this formula is appealing in NNs since it leads to faster training, there are intrinsic limitations when reducing the total number of bits (precision). Indeed, the sigmoid function does not exploit the dynamic range of the Float or Posit format enough since its codomain varies in [0, 1]. For this reason, we have developed a fast approximation of the hyperbolic tangent (see the next section).

Hyperbolic tangent

To solve said problem, an expression for the hyperbolic tangent has been derived using a linear combination of the sigmoid function:

$$\tanh(x) = 2 \cdot \text{sigmoid}(2 \cdot x) - 1.$$

This leads to a fast and approximated version of the hyperbolic tangent (FastTanh, from now on) when using the aforementioned fast sigmoid approximation:

$$\text{FastTanh}(x) = 2 \cdot \text{FastSigmoid}(2 \cdot x) - 1.$$

To have an L1 expression, we initially restrict the domain to negative numbers only. The doubling operation and sigmoid function are L1 when using zero exponent bits, and the result of the first term of the expression is contained in the unitary range. This means that computing $-(1-y)$ is also an L1 operation, according to Table 1. Finally, thanks to Tanh symmetry, we can also extend back the domain to positive numbers. Figure 4 shows the time comparison between the fast approximated version and the exact version of the hyperbolic tangent. As we can see, the FastTanh approximated version is six times faster than the exact Tanh version. Moreover, we computed the mean square error (MSE) between the two, resulting in $\text{MSE} = 2.947 \cdot 10^{-3}$ in the entire Posit interval.

A similar approach can be applied to the extended linear unit (ELU) activation function. This function solves the common problem of vanishing gradients of sigmoid-like functions, such as the hyperbolic tangent, and the effects of the flattening of the rectified linear unit (ReLu) for negative numbers:

$$\text{ELU}(x) = \begin{cases} e^x - 1, & \text{if } x \le 0 \\ x & \text{otherwise} \end{cases}.$$

Starting from the Sigmoid function, we can obtain the negative argument case as follows, where each step of the ensuing equation can be executed as an L1 operation with contained approximation:

$$\text{ELU}(x) = 2 \cdot \left(\frac{1}{2 \cdot \text{Sigmoid}(-x)} - 1 \right).$$

If we switch from the Sigmoid to the fast-approximated version already exploited with the hyperbolic tangent, we can get a fast approximation of the ELU (called *FastELU*). Table 3 shows an example of accuracy and timing improvements when using the approximated ELU function in place of the exact one. We trained a LeNet-like [49] model with the different activation functions until negligible improvements in the validation accuracy were obtainable. Then, we tested the three previously mentioned trained models with the different Posit types, reporting the accuracy and processing time. As we can see, the approximated FastELU model outperforms the ReLu model in terms of accuracy and, in particular, the type Posit8,0 shows lower degradation in terms of accuracy

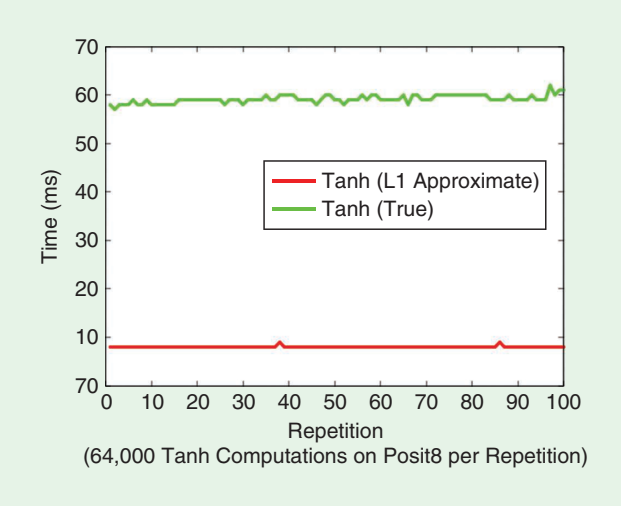


FIGURE 4. The time comparison in various repetitions of ~60,000 executions of Tanh and FastTanh for Posit16,0 [benchmarks were executed on an Intel seventh-generation (Kaby Lake) i7-7560U processor with two cores at 2.4 GHz]. The latter appears to be roughly six times faster, with a computed mean square error of $2.947 \cdot 10^{-3}$.

with FastELU/ELU rather than ReLu. In terms of timing, the FastELU and ReLu are comparable with PositN,0, both being L1 operations, while ELU is costlier. More mathematical details about FastTanh and FastELU can be found in [50].

The LUT approach

When using a low number of bits, the application of LUTs quickly becomes appealing. In theory, one could profile a specific application (i.e., computing the histogram of the most-used values and the most significant range) and then create an ad hoc series of values. For this set of values, one has only to compute the four LUTs for the four elementary operations plus the tabulation of significant unary functions (exponential, log, trigonometric functions, square root, square, and so on). There also exist some optimized soft mathematical libraries in the Sun Cephes collection ([51]). The collection consists of more than 400 mathematical functions entirely implemented in *C* and mostly delivered in different arithmetic precisions (32-, 64-, 80-, 96-, 144-, and 336-bit operands).

LUTs for Posits

The Posit LUT size depends on the overall number of Posit bits. Without any optimization, a table for a binary operation for x-bit Posits is a square one, with the number of rows and columns equal to $R = C = 2^x - 1$. Each table entry occupies b bits, depending on the underlying type used to hold the Posit number. The overall occupation for a naive table is thus $S = R \cdot C \cdot b$. For an eight-bit Posit represented across an eight-bit unsigned integer type, a single table occupies 64 kilobits. To reduce the table size, the symmetry of addition/subtraction operations can be exploited to halve the table size and number. Moreover, multiplication and division tables can be discarded by exploiting logarithm properties, thus using just the addition/subtraction tables.

Multiply and accumulate

The task of multiplying two numbers and summing the result into an accumulator is very common during DNN operations (such as convolution or matrix multiplication). The presence of an HW multiplier–accumulator is crucial since it helps in reducing by one the number of roundings involved in the computation at each step. The authors of [52] present the implementation of an exact multiply-and-accumulate (MAC) function for low-precision Posits and other floating/fixed-point types, resulting in eight-bit Posit matching and even overcoming 32-bit Floats.

Fused/exact dot product

When dealing with low-bit number representations, the dot product is a critical operation. The dot product is intensively used in DNNs during convolution operations, and overflows can occur with high probability during the accumulation of term products. To avoid most of these overflows, two solutions can be adopted.

Fused dot product

While a MAC technique computes the product result, rounds it, adds it to the accumulator, and then rounds it again, a fused dot product (FDP) (also known as *fused multiply–add*) computes the entire expression at the maximum available precision, typically using an accumulator that has twice the bits of the single operands. In [26], the potentiality of Posits for overcoming rounding issues when using fused operations is shown, such as the possibility to use 32-bit Posits for high-performance computing instead of 64-bit Posits, thus increasing the computation speed and reducing the power consumption and storage requirements.

Exact dot product

The exact dot product technique makes use of the concept of quires (a very-high-bit-count scratch area) as the accumulator, deferring rounding only at the very last operation, thus minimizing rounding errors. The concept of quires was introduced by Kulisch in [53] to minimize the number of transistors used to build a fixed-size register inside a processor. A quire is a very-high-bit-count fixed-size scratch area used to perform arithmetic operations at the maximum possible precision given by that fixed-size type. If the quire is properly dimensioned, the rounding error will affect only the very last operation when converting the result back to the original low-precision type. To prevent the quire from underflow or overflow during these operations, we need to dimension it depending on the Posit configuration (https://posithub.org/docs/Posits4.pdf). Suppose that to have a totbits–bit Posit, the maximum possible value for the Posit will be $maxpos = u^{\text{totbits}-2}$, while the minimum possible value will be $minpos = 1/maxpos$, where $u = 2^{2^{\text{esbits}}}$; each number is,

Table 3. The comparison of different activation functions when applied to an NN for traffic sign classification.

Activation	GTRSB								
	FastELU			ReLu			ELU		
	Accuracy (%)	Time (ms)	NCT[1]	Accuracy (%)	Time (ms)	NCT[1]	Accuracy (%)	Time (s)	NCT[1]
Posit16,0	94	5.8	—	92	5	—	94.2	6.4	—
Posit14,0	94	4.6	0.79	92	4.3	0.86	94.2	5.2	0.81
Posit12,0	94	4.6	0.79	92	4.3	0.86	94.2	5.1	0.79
Posit10,0	94	4.6	0.79	92	4.2	0.84	94.2	5	0.78
Posit8,0	92	4.6	0.79	86.8	4	0.8	91.8	5	0.78

Benchmarks were executed on an Intel seventh-generation (Kaby Lake) i7-7560U processor with two cores at 2.4 GHz.
[1]NCT: normalized computing time. (Posit computing times are normalized against the Posit16,0 computing times.)

then, an integer multiple of *minpos*. Suppose we need to perform the dot product $\{maxpos, minpos\} \cdot \{maxpos, minpos\}$; we'll need the quire to be able to accommodate the value $maxpos^2/minpos^2$. After some transformation, we can compute the maximum value to hold as

$$2^{(4 \cdot \text{totbits} - 8) \cdot 2^{\text{esbits}}}.$$

Moreover, one bit has to be reserved for the sign, and more bits must be held to handle the sum (e.g., Gustafson chooses 30 more bits to guarantee the absence of overflows). Practically, for example, this means that with an eight-bit Posit (esbits = 0), we will need one 64-bit quire register; for a 16-bit Posit (esbits = 1), we will need a 256-bit quire register (four 64-bit registers); and for a 32-bit (esbits = 2) Posit, we will need a 512-bit quire register (eight 64-bit registers).

Kalray massively parallel processor array approach

To address the challenges of high-performance embedded computing with time predictability, Kalray has been refining a homogeneous manycore architecture, called the *massively parallel processor array* (*MPPA*), based on very-long instruction word (VLIW) cores. On the third-generation MPPA processor [54], each VLIW core is paired with a coprocessor designed for 2D data processing, especially the mixed-precision tensor operations of deep learning inference. In particular, each coprocessor implements matrix multiply–accumulate operations on INT8/32 and Float16/32, where we use the forward slash to describe the two bandwidths of the multiplicand and accumulator. Exploitation of INT8/32 operations relies on TensorFlow Lite quantization support [55], while exploitation of the Float16/32 arithmetic through standard frameworks is the same as for NVIDIA general-purpose GPUs. However, unlike the NVIDIA tensor cores, the Kalray MPPA-3 coprocessors perform exact dot–product processes inside the Float16.32 matrix multiply–accumulate operations by applying Kulisch's principles on an $80 + \epsilon$ accumulator [56].

Following [52], the Posit8 numbers have been identified by Kalray as an effective compressed representation for the Float32 network parameters: instead of rounding the Float32 parameter values to Float16 values, the results of rounding can be restricted to Posit8,0 or Posit8,1 numbers, with the primary benefit of reducing by half the memory capacity and bandwidth required by the network parameters. Kalray focuses on the Posit8,0 and Posit8,1 numbers because they are exactly represented as Float16 numbers and thus can benefit from the exact Float16/32 dot–product operator of the MPPA-3 coprocessors. Conversely, the Posit8,2 numbers include eight values of magnitude 65,536 and larger that are out of range of the Float16 numbers, while the Posit8,3 numbers overflow even the range of the BFLOAT16 numbers. Evaluation of the HW costs and application benefits of using Posit8,0 numbers as a compressed format for Float32 network parameters is ongoing. This evaluation should lead to the inclusion of new arithmetic instructions to expand Posit8,0 to Float16 in the MPPA Internet Protocol delivered to the Horizon 2020 EPI.

Preliminary results obtained by comparing the use of Float32, Float16, and Posit8,E (with an E from zero to three) for data storage (while computation is still done in Float32) during the inference phase using network models for both the classification task [e.g., SqueezeNet, Alexnet, Visual Geometry Group (VGG)-16, VGG-19, GoogleNet and a custom convolutional NN (CNN) on the Modified National Institute of Standards and Technology (MNIST) database, and Canadian Institute for Advanced Research (CIFAR)-100] and detection task [e.g., You Only Look Once (YOLO) v3] show that Posit8,1 or Posit8,2 offers the best performance, with an accuracy loss below 1% versus Float32 but a data compression of factor four. This will lead to reduced complexity for the data transfer and storage that are dominating DNN applications. It should be noted that 1) the networks were pretrained using Float32 and 2) the used data sets in the reported results had thousands of images. Indeed, the ImageNet Large Scale Visual Recognition Challenge 2012 data set has been used for classification and the Visual Object Classes Challenge 2012 data set for detection.

Vectorization of Posit operations (tested on random images)

While in the absence of proper HW support for Posits (i.e., a posit processing unit), we can still accelerate DNN core functions and operators using already-existing HW accelerators. This is the case of the ARM Scalable Vector Extension (SVE) single instruction, multiple data engine. We have also ported our cppPosit library to provide a vectorized version of Posit functions exploiting the ARM SVE library. When talking about vectorized functions, L1 operations are the easiest ones to vectorize. In fact, since they rely only on integer arithmetic and logic, we can effortlessly exploit the native ARM SVE vectorization of integer operations. Benchmarks were executed on a HiSilicon Hi1616 CPU with a 32-bit, 2.4-GHz ARM Cortex-A72 processor, using the ARM SVE Instruction Emulator.

Table 4 shows some timing results between the vectorized and nonvectorized approaches. Furthermore, we have provided an interface between the Posit floating-point back end and ARM SVE types to vectorize L3/4 operations, as well. This enabled implementation of the Posit-accelerated version of convolution and pooling operations. Table 5 provides an example of the timing results with 3×3 convolution and maximum-pooling operations. Finally, Table 6 gives the vectorization performance in terms of processing time on the low-precision inference on Posit8,0. The performance was obtained on the tiny-DNN library on various very-deep NNs. All benchmarks have been executed on the ARM instruction emulator. As reported, the processing time with SVE vectorization enabled dramatic speedups. Note that, in terms of absolute values, the processing time is quite large. Clearly, this is due to the fact that SVE-enabled HW is not available at the time of writing, and all benchmarks are executed inside the ARM SVE instruction emulator.

DNN signal processing performance: Accuracy and complexity

In [52] and [57], Carmichael et al. show an architecture using Posits in DNNs called *Deep Positron*, using an exact MAC

technique on eight-bit low-precision formats. The architecture has been tested on the MNIST, Fashion–MNIST, and other data sets, reporting no drop in accuracy with regard to Float32. Another approach to deep learning with low-bit numbers has been tested in [39], using logarithmic numbers with a residential NN (ResNet)-50 architecture on Imagenet, resulting in a 0.9 percentage point drop when shifting from Float32 to logarithmic representation. We have integrated the cppPosit library in a DNN C++ library called *tiny-DNN* [58] that is capable of supporting various computing arithmetic, such as BFLOAT16, Flexpoint, and Posits. Then, we tested the accuracy of different network models in image classification benchmarks, such as MNIST, Fashion–MNIST, CIFAR-10, and GTRSB, using the FDP technique. For the MNIST data set, we registered a drop of 0.9 percentage points when testing the model from Float32 to Posit8. For GTRSB, we registered a drop of 0.2 percentage points, instead. For other Posit configurations with 16, 14, 12, and 10 bits, we registered no drop in accuracy from Float32 to the Posit type.

Table 4. The L1 operations performance processing-time comparison between nonvectorized (naive) and vectorized (SVE-X) approaches.

	FastSigmoid (ms)		FastTanh (ms)		FastELU (ms)	
Posit	**8,0**	**16,0**	**8,0**	**16,0**	**8,0**	**16,0**
Version						
Naive	3.08	3.41	5.76	7.24	8.12	8.54
SVE-128	0.73	1.51	1.32	2.65	1.29	2.6
SVE-256	0.59	1.05	1.18	1.83	1.16	1.79
SVE-512	0.43	0.62	0.69	1.09	0.69	1.05
SVE-1024	0.29	0.39	0.48	0.72	0.46	0.68
SVE-2048	0.22	0.28	0.36	0.5	0.35	0.47

Each timing result comes from the function computation on a vector of 8,192 items.

Table 5. The 3 × 3 convolution and pooling processing-time comparison on two common Posit configurations with 225 × 225 random images.

	Maximum Pooling (ms)		Convolution (ms)	
Posit	**8,0**	**16,0**	**8,0**	**16,0**
Version				
Naive	49.7	59.41	80.67	80.84
SVE-128	9.51	26.52	24.02	37.99
SVE-256	8.89	22.06	11.66	21.49
SVE-512	6.96	14.69	6.85	14.03
SVE-1024	5.12	11.84	6.38	12.88
SVE-2048	4.13	9.76	3.65	8.81

The naive approach is the nonvectorized one. The other approaches are with incremental SVE–vector registers.

Table 6. The image processing time for various very deep NN models using Posit8,0.

Version	Alexnet Time(s)	ResNet-34 Time(s)	VGG-16 Time(s)	VGG-19 Time(s)	ResNet-150 Time(s)
Naive	40.06	146.07	590.68	675.32	779.7
SVE-128	2.76	10.07	40.74	46.57	53.77
SVE-256	2.64	9.61	38.88	44.45	51.32
SVE-512	2.54	8.93	36.12	41.3	47.68
SVE-1024	2.44	8.92	36.06	41.23	47.6
SVE-2048	2.34	8.9	35.97	41.13	47.48

For this benchmark, random red–green–blue 224 × 224 images are employed.

Benchmark data sets and examples of achievable results

We have considered different standard data sets, such as the one shown in Figure 5, and standard CNN architectures, including the one in Figure 6. In particular, for the MNIST and GTSRB benchmarks, we trained customized CNN variants of the one reported in Figure 6, including Posit-related optimizations to the convolutional and activation layers. For the Fashion–MNIST benchmark, we used a pretrained model with a starting accuracy of 95%. For CIFAR-10, we used the VGG-16 pretrained model [59]. All the networks were initially trained using Float32 and then evaluated on the corresponding test sets, converting each Float32-trained model using different Posit configurations. Furthermore, to provide a fair timing–accuracy tradeoff comparison, the Float32 model has been tested exploiting the SoftFloat library for SW-emulated floating-point numbers.

MNIST, Fashion–MNIST, and CIFAR-10

Table 7 presents the results obtained on three well-known classification benchmarks: MNIST, Fashion–MNIST, and CIFAR-10. MNIST is a digit-recognition problem, while Fashion–MNIST has been designed as a more complex drop-in replacement for the MNIST data set, providing more general classes to be recognized (such as fashion products). Furthermore, CIFAR-10 consists in an even more complex task, bringing three-channel images in the data set. As reported, the tests on the model with the different types show that Posits with zero exponent bits and sized from 12 to 14 bits can be a perfect replacement for Float32, while those with 10 and eight bits can replace Float32 with some drop in accuracy. The same holds for the Fashion–MNIST data set.

Note how the processing time [on an Intel seventh-generation (Kaby Lake) i7 processor] for a single-image inference of the VGG-16 model on a CIFAR-10 sample is expressed in

FIGURE 5. The GTRSB data set example.

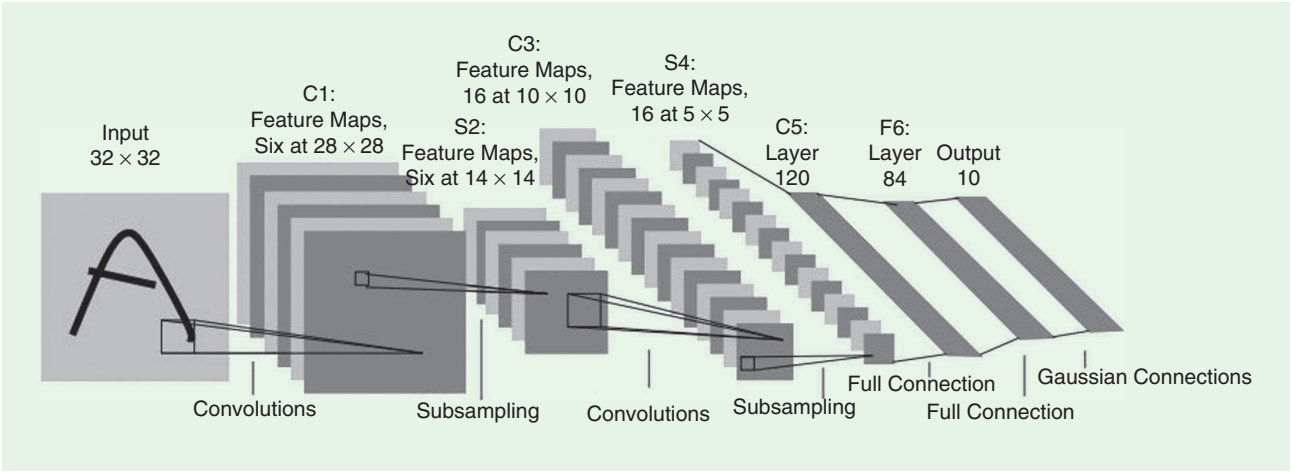


FIGURE 6. The LeNet-5 architecture as described in [49]. Some customization has been added to the network to better fit our goals: the activation function has been changed to FastTanh (as described before) for the MNIST data set and to a fast approximation of the ELU for the GTSRB data set. The input size of the first layer has been extended to hold the $64 \times 64 \times 3$ color images of the GTSRB data sets.

Table 7. The accuracy and processing time obtained on MNIST, Fashion–MNIST, and CIFAR-10 data sets.

Type	MNIST			Fashion–MNIST			CIFAR-10		
	Accuracy (%)	Time (ms)	NCT[1]	Accuracy (%)	Time (ms)	NCT[1]	Accuracy (%)	Time (s)	NCT[1]
SoftFloat32	99.4	8.8	—	95	41.9	—	93.75	7.75	—
Posit16,0	99.4	5.2	0.59	95	13.6	0.32	93.75	2.55	0.32
Posit14,0	99.4	4.6	0.52	95	13.5	0.32	93.75	2.49	0.32
Posit12,0	99.4	4.6	0.52	95	13.5	0.32	93.75	2.44	0.31
Posit10,0	99.3	4.6	0.52	95	13.4	0.32	93.75	2.4	0.3
Posit8,0	98.5	3.8	0.43	94	13.4	0.32	85	2.34	0.3

The processing time is evaluated as the mean per-sample inference time on the test set of the relative data set.
[1]Posit computing times are normalized against SoftFloat32 computing times.

Table 8. The accuracy–processing time tradeoff obtained on the GTRSB data set.

	GTRSB		
Type	Accuracy (%)	Time (ms)	NCT[1]
SoftFloat32	94	15.86	—
Posit16,0	94	6.37	0.4
Posit14,0	94	5.21	0.32
Posit12,0	94	5.08	0.32
Posit10,0	94	5	0.31
Posit8,0	93.8	4	0.25

[1]Posit computing times are normalized against SoftFloat32 computing times.

seconds, highlighting the infeasibility of this model for traditional CPU architectures. However, we are moving toward GPU-enabled DNN libraries, as described in the "Conclusions and Road Maps" section. For comparison, an entire training epoch of 60,000 CIFAR-10 samples on a Resnet-50 architecture takes only approximately 30 s on a dual-GPU (Tesla T4) configuration, thus only 0.5 ms for the forward and backward passes (including the weight update). It should also be noted that to make the comparison fair, we evaluate, in Tables 6 and 7, the SW implementation of Posits (using our developed cppPosit library) against an SW implementation of Floats (the SoftFloat library). From Tables 6 and 7, we can observe that moving from SoftFloat32 to Posit8,0, we get (roughly) the same classification accuracy on all considered data sets but with a reduction in the computing time of roughly a factor of three.

Automotive benchmarks: The traffic sign recognition problem

In this section, we report the results obtained on a classification benchmark related to assisted/autonomous driving. Benchmarks were executed on an Intel seventh-generation (Kaby Lake) i7-7560U processor with two cores at 2.4 GHz. The GTRSB is a baseline benchmark for road sign recognition, which is very interesting as an automotive task. Table 8 shows that, in this case also, Posits from 12 to 16 bits, and even 10 bits, can be a perfect replacement for Float32, while Posit8,0 performs well with a little drop in accuracy. We have also started an activity to assess the performance of Posits using the YOLO approach [60], [61] and Apollo [62] (http://apollo.auto/) heterogeneous framework, and the achieved results confirm what we already obtained with the GTRSB, MNIST, and Fashion–MNIST data sets. Moreover, we began an activity to assess Posit performance in semantic segmentation tasks (such as pixel- and instance-level classification [33], [34]) on famous data sets, such as CityScapes (see Figure 7).

FIGURE 7. The CityScapes data set example of the semantic segmentation of a road in Stuttgart, Germany.

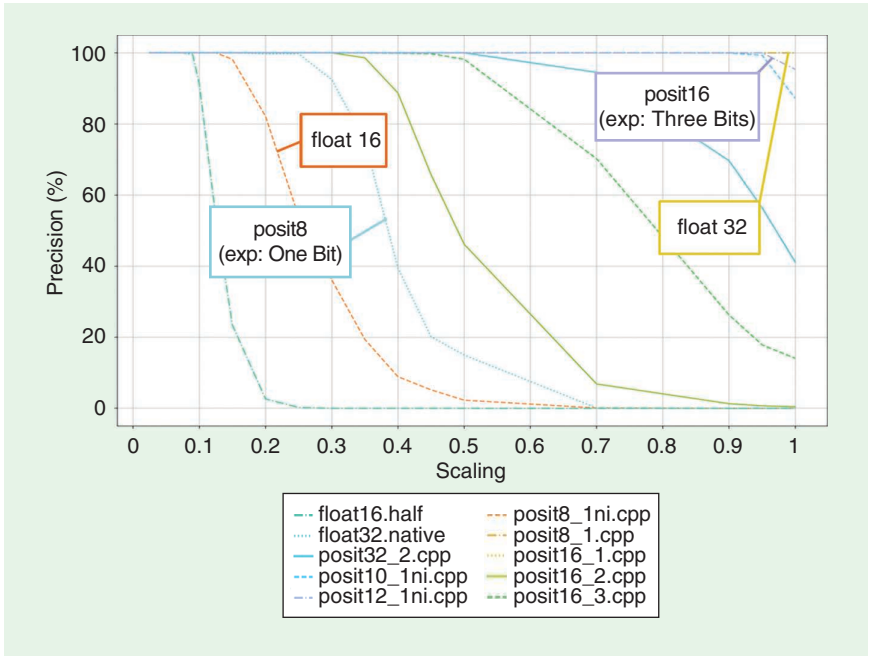


FIGURE 8. The performance of the *k*-NN using various data types on a single data set employing different values for the scaling factor.

The results we are obtaining are in line with those from the MNIST, Fashion–MNIST, and GTRSB data sets.

k-nearest neighbors results

The *k*-nearest neighbors (*k*-NN) algorithm is ubiquitous in pattern-recognition problems. It can be used to segment images and to compute the normal vectors to each point of a point cloud obtained by a lidar sensor mounted on a car. The *k*-NN algorithm finds the *k* nearest neighbors of a given point from those in a given data set. We have compared the performance of the *k*-NN when using Posits and Floats and, again, found that the accuracy of Posit16,0 is very close to that of Float32 (see Figure 8) and that a Posit8,0 outperforms Float16. These results have been obtained on a single data set, scaling it multiple times to reduce the dynamic range of the input data (thus enabling low-precision data types to be competitive with Float32). More details can be found in [63]. The obtained results confirm that Posits are powerful in a number of machine learning applications, meaning that implementing Posit-based HW accelerators will be beneficial for numerous different applications.

Next experiments

We are working toward the implementation of other fast approximated functions (e.g., the ELU). We are currently porting our cppPosit-based tiny-DNN library on the ARM instruction emulator used within the Horizon 2020 EPI [64] to exploit the SVE-2 as much as possible (providing a vectorization back end for the cppPosit library). We are also planning to test our SW on available simulators, such as GEM5, SESAM, and MUSA, to provide useful feedback to the ongoing EPI processor codesign process.

Conclusions and road maps

In this article, we have reviewed the state of the art of DNN signal processing for autonomous driving applications and the quest for novel representations of real numbers that must be both efficient and reliable. We have seen how Posit is a suitable drop-in replacement for the IEEE 754 standard, and we have assessed its potentialities in autonomous driving applications. Implementations with both SW libraries and HW–SW embedded systems, from academia and industry, have been discussed. The achieved results when combining Posit arithmetic with DNNs are promising in terms of the tradeoff between accuracy and processing time. From this and related works, it is clear that the current challenges are 1) the development of real-time and low-power accelerators for performing DNN inference at the edge, 2) the development of methods for DNN verification and validation for the high coverage rates required by standards for safety-critical applications, and 3) moving toward a GPU-enabled DNN library, such as Tensorflow, to build, train, and evaluate even more complex models once they are integrated with our cppPosit library. Furthermore, we plan to test our approach on GPU-enabled ARM devices, such as NVIDIA

Jetson boards; mobile devices that do not employ GPUs; and even without the FPU.

Acknowledgments

This work was partially funded by the Horizon 2020 European Processor Initiative (grant agreement 826647) and the Italian Ministry of Education and Research through the CrossLab project (departments of excellence).

Authors

Marco Cococcioni (marco.coccioni@unipi.it) has been an associate professor in the Department of Information Engineering, University of Pisa, Pisa, Italy, since 2016. He is on the editorial board of four journals indexed by Scopus and coauthored more than 90 scientific contributions. He has been the general chair of three IEEE conferences and a program committee member of 50-plus international conferences in the area of computational intelligence. He is a member of three IEEE task forces: Genetic Fuzzy Systems, Computational Intelligence in Security and Defense, and Intelligent System Applications. He is a Senior Member of IEEE and ACM.

Federico Rossi (federico.rossi@ing.unipi.it) is a Ph.D. student in the Department of Information Engineering, University of Pisa, Pisa, Italy. In 2019, he received his M.S. degree magna cum laude in computer engineering. He is currently involved in the European Processor Initiative project. His research interests include alternative real-number representations and their applications to deep neural networks for the automotive environment.

Emanuele Ruffaldi (emanuele.ruffaldi@mmimicro.com) is a senior software engineer at MMI, Pisa, Italy, working on robotic-assisted microsurgery. Formerly, he was an assistant professor in the Perceptual Robotics Laboratory, Scuola Superiore Sant'Anna, Pisa, Italy. His research interests include machine learning for human–robot interaction and embedded artificial intelligence. He coauthored more than 100 scientific publications and one patent. He is a Senior Member of IEEE and has served as publicity chair for the IEEE Haptics Technical Committee.

Sergio Saponara (sergio.saponara@unipi.it) is a full professor at the University of Pisa, Pisa, Italy, where he is responsible for automotive-electronics, hardware-security, and embedded-system activities; president of B.Sc. and M.Sc. degrees in electronic engineering; and head of the university's participation in the European Processor Initiative project. He is also director of a summer school on the industrial Internet of Things and a specialization course on automotive powertrain electrification. The coauthor of more than 300 scientific publications and approximately 20 patents, he is an associate editor of many peer-reviewed journals. He is a Senior Member of IEEE and is an IEEE Distinguished Lecturer.

Benoît Dupont de Dinechin (benoit.dinechin@kalray.eu) is the chief technology officer at Kalray, Montbonnot-Saint-Martin, France. He received his engineering degree from ISAE-SUPAERO, Toulouse, France, and his Ph.D. degree in computer systems from the Université Pierre et Marie Curie, Paris, France. He completed his postdoctoral studies at McGill University, Montréal. He is the main architect of the Kalray very-long instruction word cores and coarchitect of the company's massively parallel processor arrays. He defined the Kalray software road map and still contributes to its implementation. Before joining Kalray, he led R&D for the Software Tools division at STMicrelectronics, becoming a fellow in 2008. Prior to STMicrelectronics, he developed the software pipeliner of the Cray T3E production compilers at the Cray Research Park, Eagan, Minnesota.

References

[1] S. Saponara and B. Neri, "Radar sensor signal acquisition and multidimensional FFT processing for surveillance applications in transport systems," *IEEE Trans. Instrum. Meas.*, vol. 66, no. 4, pp. 604–615, 2017. doi: 10.1109/TIM.2016.2640518.

[2] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Trans. Ind. Informat.*, vol. 15, no. 2, pp. 1038–1051, 2019. doi: 10.1109/TII.2018.2879544.

[3] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, SAE Standard J3016_201806, 2018.

[4] J. Royo-Alvarez, M. Martínez-Ramón, J. Muñoz-Marí, and G. Camps-Valls, Eds., *From Signal Processing to Machine Learning*. Hoboken, NJ: Wiley, 2018, ch. 1, pp. 1–11.

[5] L. Deng, "Artificial intelligence in the rising wave of deep learning: The historical path and future outlook [perspectives]," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 180–177, 2018. doi: 10.1109/MSP.2017.2762725.

[6] IEEE Signal Processing Society (SPS) ASI. Accessed: Oct. 1, 2020. [Online]. Available: https://ieeeasi.signalprocessingsociety.org/

[7] M. T. McCann, K. H. Jin, and M. Unser, "Convolutional neural networks for inverse problems in imaging: A review," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 85–95, 2017. doi: 10.1109/MSP.2017.2739299.

[8] A. Arnab, S. Zheng, S. Jayasumana, B. Romera-Paredes, M. Larsson, A. Kirillov, B. Savchynskyy, C. Rother et al., "Conditional random fields meet deep neural networks for semantic segmentation: Combining probabilistic graphical models with deep learning for structured prediction," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 37–52, 2018. doi: 10.1109/MSP.2017.2762355.

[9] S. F. Dodge and L. J. Karam, "Quality robust mixtures of deep neural networks," *IEEE Trans. Image Process.*, vol. 27, no. 11, pp. 5553–5562, 2018. doi: 10.1109/TIP.2018.2855966.

[10] P. Nousi, A. Tefas, and I. Pitas, "Deep convolutional feature histograms for visual object tracking," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP 2019)*, 2019, pp. 8375–8379. doi: 10.1109/ICASSP.2019.8683004.

[11] A. Voulodimos, N. D. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Comput. Intell. Neurosci.*, vol. 2018, Feb. 2018, Art. no. 7068349. doi: 10.1155/2018/7068349.

[12] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond Euclidean data," *IEEE Signal Process. Mag.*, vol. 34, no. 4, pp. 18–42, 2017. doi: 10.1109/MSP.2017.2693418.

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, 2017. doi: 10.1109/MSP.2017.2743240.

[14] J. Han, D. Zhang, G. Cheng, N. Liu, and D. Xu, "Advanced deep-learning techniques for salient and category-specific object detection: A survey," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 84–100, 2018. doi: 10.1109/MSP.2017.2749125.

[15] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 126–136, Jan. 2018. doi: 10.1109/MSP.2017.2765695.

[16] P. Nousi, E. Patsiouras, A. Tefas, and I. Pitas, "Convolutional neural networks for visual information analysis with limited computing resources," in *Proc. 25th IEEE Int. Conf. Image Processing (ICIP'18)*, 2018, pp. 321–325. doi: 10.1109/ICIP.2018.8451600.

[17] D. Reinhardt, U. Dannebaum, M. Scheffer, and M. Traub, "High performance processor architecture for automotive large scaled integrated systems within the european processor initiative research project," SAE International, Warrendale, PA, SAE Tech. Paper 2019-01-0118, 2019.

[18] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray et al., "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proc. Advances Neural Information Processing Systems*, 2017, pp. 1742–1752.

[19] V. Popescu, M. Nassar, X. Wang, E. Tumer, and T. Webb, "Flexpoint: Predictive numerics for deep learning," in *Proc. IEEE 25th Symp. Computer Arithmetic (ARITH)*, June 2018, pp. 1–4. doi: 10.1109/ARITH.2018.8464801.

[20] G. Tagliavini, A. Marongiu, and L. Benini, "FlexFloat: A software library for transprecision computing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 1, pp. 145–156, 2020. doi: 10.1109/TCAD.2018.2883902.

[21] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Toms, D. S. Nikolopoulos et al., "The transprecision computing paradigm: Concept, design, and applications," in *Proc. Design, Automation Test Europe Conf. Exhibition (DATE)*, Mar. 2018, pp. 1105–1110. doi: 10.23919/DATE.2018.8342176.

[22] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating deep convolutional networks using low-precision and sparsity," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2017, pp. 2861–2865. doi: 10.1109/ICASSP.2017.7952679.

[23] G. Srivastava, D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti, and J. Seo, "Joint optimization of quantization and structured sparsity for compressed deep neural networks," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP 2019)*, 2019, pp. 1393–1397. doi: 10.1109/ICASSP.2019.8682791.

[24] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, Quantized neural networks: Training neural networks with low precision weights and activations. 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[25] M. Cococcioni, E. Ruffaldi, and S. Saponara, "Exploiting posit arithmetic for deep neural networks in autonomous driving applications," in *Proc. Int. Conf. Electrical and Electronic Technologies Automotive*, 2018, pp. 1–6. doi: 10.23919/EETA.2018.8493233.

[26] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.*, vol. 4, no. 2, pp. 71–86, 2017. doi: 10.14529/jsfi170206.

[27] P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha, "Autonomous vehicles: State of the art, future trends, and challenges," in *Automotive Systems and Software Engineering*, Y. Dajsuren and M. van den Brand, Eds. Cham, Switzerland: Springer-Verlag, 2019, pp. 347–367.

[28] A. Woo, B. Fidan, and W. W. Melek, "Localization for autonomous driving," in *Handbook of Position Location: Theory, Practice, and Advances*. Hoboken, NJ: Wiley, 2019, ch. 29, pp. 1051–1087.

[29] Y. Wenhui and Y. Fan, "Lidar image classification based on convolutional neural networks," in *Proc. Int. Conf. Computer Network, Electronic and Automation (ICCNEA)*, 2017, pp. 221–225. doi: 10.1109/ICCNEA.2017.37.

[30] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Netw.*, vol. 32, pp. 323–332, Aug. 2012. doi: 10.1016/j.neunet.2012.02.016.

[31] V.-D. Hoang, M.-H. Le, T. T. Tran, and V.-H. Pham, "Improving traffic signs recognition based region proposal and deep neural networks," in *Intelligent Information and Database Systems*, N. T. Nguyen, D. H. Hoang, T.-P. Hong, H. Pham, and B. Trawiński, Eds. Cham, Switzerland: Springer-Verlag, 2018, pp. 604–613.

[32] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth et al., "The cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 3213–3223.

[33] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proc. European Conf. Computer Vision (ECCV)*, 2018, pp. 833–851.

[34] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu et al., "Searching for MobileNetV3," in *Proc. Int. Conf. Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[35] M. Courbariaux, Y. Bengio, and, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Red Hook, NY: Curran Associates Inc., 2015, pp. 3123–3131.

[36] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.

[37] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Machine Learning (ICML'15)*, 2015, vol. 37, pp. 1737–1746.

[38] P. Lindstrom, "Universal coding of the reals using bisection," in *Proc. Conf. Next Generation Arithmetic (CoNGA'19)*, 2019, pp. 7:1–7:10. doi: 10.1145/3316279.3316286.

[39] J. Johnson, Rethinking floating point for deep learning. 2018. [Online]. Available: http://arxiv.org/abs/1811.01721

[40] M. G. Arnold, J. Garcia, and M. J. Schulte, "The interval logarithmic number system," in *Proc. 16th IEEE Symp. Computer Arithmetic*, 2003, pp. 253–261. doi: 10.1109/ARITH.2003.1207686.

[41] J. Gustafson, "Posits and quires: Freeing programmers from mixed-precision decisions," in *Proc. 34th Int. Supercomputing Conf. High Performance (ISC'19)*, Frankfurt, Germany, June 16–20, 2019.

[42] E. Ruffaldi, "cppPosit," GitHub, San Francisco. Accessed: Oct. 1, 2020. [Online]. Available: https://github.com/eruffaldi/cppPosit

[43] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "A fast approximation of the hyperbolic tangent when using Posit numbers and its application to deep neural networks," in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds. Cham, Switzerland: Springer-Verlag, 2020, pp. 213–221.

[44] M. K. Jaiswal and H. K.-H. So, "Universal number posit arithmetic generator on FPGA," in *Proc. Design, Automation Test Europe Conf. Exhibition (DATE)*, 2018, pp. 1159–1162. doi: 10.23919/DATE.2018.8342187.

[45] A. Podobas and S. Matsuoka, "Hardware implementation of POSITs and their application in FPGAs," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, 2018, pp. 138–145. doi: 10.1109/IPDPSW.2018.00029.

[46] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *Proc. IEEE 36th Int. Conf. Computer Design (ICCD)*, 2018, pp. 334–341. doi: 10.1109/ICCD.2018.00057.

[47] M. K. Jaiswal and H. K. So, "PACoGen: A hardware Posit Arithmetic Core Generator," *IEEE Access*, vol. 7, pp. 74,586–74,601, June 2019. doi: 10.1109/ACCESS.2019.2920936.

[48] L. van Dam, J. Peltenburg, Z. Al-Ars, and H. P. Hofstee, "An accelerator for posit arithmetic targeting posit level 1 BLAS routines and pair-HMM," in *Proc. Conf. Next Generation Arithmetic (CoNGA'19)*, 2019, pp. 5:1–5:10. doi: 10.1145/3316279.3316284.

[49] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. doi: 10.1038/nature14539.

[50] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Fast approximations of activation functions in deep neural networks when using Posit arithmetic," *Sensors*, vol. 20, no. 5, p. 1515, 2020. doi: 10.3390/s20051515.

[51] Netlib, "Cephes mathematical function library." Accessed: Oct. 1, 2020. [Online]. Available: http://www.netlib.org/cephes/

[52] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," in *Proc. Conf. Next Generation Arithmetic 2019 (CoNGA'19)*, 2019, pp. 3:1–3:9. doi: 10.1145/3316279.3316282.

[53] U. Kulisch, "An axiomatic approach to rounded computations," *Numerische Mathematik*, vol. 18, pp. 1–17, Feb. 1971. doi: 10.1007/BF01398455.

[54] B. D. de Dinechin, "Consolidating high-integrity, high-performance, and cybersecurity functions on a manycore processor," in *Proc. 56th ACM/IEEE Design Automation Conf. (DAC'19)*, 2019, pp. 1–4, doi: 10.1145/3316781.3323473.

[55] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR 2018)*, Salt Lake City, UT, June 18–22, 2018, pp. 2704–2713. doi: 10.1109/CVPR.2018.00286.

[56] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *Proc. IEEE 24th Symp. Computer Arithmetic (ARITH)*, July 2017, pp. 106–113. doi: 10.1109/ARITH.2017.29.

[57] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep Positron: A deep neural network using the Posit number system," in *Design, Automation and Test Europe Conf. and Exhibition (DATE'19)*, 2019, pp. 1421–1426. doi: 10.23919/DATE.2019.8715262.

[58] "tiny-dnn," GitHub, San Francisco. Accessed: Oct. 1, 2020. [Online]. Available: https://github.com/tiny-dnn/tiny-dnn

[59] K. Simonyan and A. Zisserman, Very deep convolutional networks for large-scale image recognition. 2014. [Online]. Available: https://arxiv.org/abs/1409.1556

[60] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J. Frahm, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2019, pp. 305–317. doi: 10.1109/RTAS.2019.00033.

[61] S. Goel, A. Baghel, A. Srivastava, A. Tyagi, and P. Nagrath, "Detection of emergency vehicles using modified YOLO algorithm," in *Intelligent Communication, Control and Devices*, S. Choudhury, R. Mishra, R. G. Mishra, and A. Kumar, Eds. Singapore: Springer-Verlag, 2020, pp. 671–687.

[62] H. Tabani, L. Kosmidis, J. Abella, F. J. Cazorla, and G. Bernat, "Assessing the adherence of an industrial autonomous driving framework to ISO 26262 software guidelines," in *Proc. 56th ACM/IEEE Design Automation Conf. (DAC'19)*, 2019, pp. 1–6. doi: 10.1145/3316781.3317779.

[63] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Novel arithmetics to accelerate machine learning classifiers in autonomous driving applications," in *Proc. 26th IEEE Int. Conf. Electronics Circuits and Systems (ICECS'19)*, 2019, pp. 779–782. doi: 10.1109/ICECS46596.2019.8965031.

[64] European Processor Initiative, "European Processor Initiative, an H2020 project," 2019–2021. [Online]. Available: https://www.european-processor-initiative.eu/

SP