Novel Arithmetics in Deep Neural Networks Signal Processing for Autonomous Driving: Challenges and Opportunities

Marco Cococcioni, *Senior Member, IEEE,* Federico Rossi, Emanuele Ruffaldi, *Senior Member, IEEE,* Saponara Sergio, *Senior Member, IEEE* and Benoît Dupont de Dinechin

***Abstract*—This paper focuses on trends, opportunities and challenges of novel arithmetics for DNN signal processing, with particular reference to assisted and autonomous driving applications. Due to strict constrains in terms of latency, dependability and security of autonomous driving, machine perception (i.e. detection or decisions tasks) based on DNN can not be implemented relying on a remote cloud access. These tasks must be performed in real-time on embedded systems on-board the vehicle, particularly for the inference phase (considering the use of DNNs pre-trained during an off-line step). When developing a DNN computing platform, the choice of the computing arithmetics matters. Moreover, functional safe applications like autonomous driving pose severe constraints on the effect that signal processing accuracy has on final rate of wrong detection/decisions. Hence, after reviewing the different choices and trade-off concerning arithmetics, both in academia and industry, we highlight the issues in implementing DNN accelerators to achieve accurate and low-complex processing of automotive sensor signals (the latter coming from diverse sources like cameras, radars, lidars, ultrasonics). The focus is on both on general-purpose operations massively used in DNN like multiply, accumulation, compare, or on specific functions like for example sigmoid or hyperbolic tangent, used for neuron activation.**

***Index Terms*—Deep Neural Networks (DNN), Autonomous Driving, Real-Time Image & Signal Processing/Classification, Alternative Real Number Representations, HW Accelerators.**

I. Introduction

THE use of deep neural networks (DNNs) as a general tool for signal and data processing is increasing both in automotive industry and academia, proposing a set of algorithms for most of the autonomous driving tasks.

The effort in computing these artificial intelligence algorithms is an open challenge in the field of computing platforms nowadays. In particular, when considering strict requirements, such as lowering the power consumption, maximizing the throughput and minimizing the latency the computational complexity becomes more and more critical. Moreover, with the modern achievements in sensor components, the complexity and requirements further scale with data coming in higher volumes and dimensions and at higher speed [1].

M. Cococcioni, F. Rossi and S. Saponara are with the Department of Information Engineering, University of Pisa, 56122 Pisa – Italy, e-mail: {marco.cococcioni, federico.rossi, sergio.saponara}@unipi.it

E. Ruffaldi is with MMI spa, e-mail: emanuele.ruffaldi@mmimicro.com

B. Dupont de Dinechin is with Kalray, e-mail: benoit.dinechin@kalray.eu

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, and the relevant computing arithmetic. Indeed, autonomous driving is a safety critical application, as specified also in functional safety standards like ISO26262, with strict requirements in terms of real-time (both throughput and latency) [1, 2]. In Levels $\mathcal{L}1$ and $\mathcal{L}2$ and of the SAE autonomous driving scale [3] just an assistance to human driver is needed. Therefore, signal processing based on deterministic algorithms is still enough, e.g. FFT-based processing of Frequency Modulated Continuous Wave Radar (FMCW) as done in [1]. Instead, for high autonomous driving levels, from $\mathcal{L}3$ to $\mathcal{L}5$, the complexity of the scenario and the need of signal processing not only for sensing, but also for localization, navigation, decision and actuation, is so high that in recent state-of-art DNN 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 SP society [6]. DNNs have reached state-of-art in several signal processing domains like 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, lidars, radars, 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, sub-scene classification, etc.) must be done on-board the vehicles, since cloud-based computing scenarios (where the signal processing is done on remote cloud server and on-board there is only a client unit generating requests to the server) suffers of several issues: privacy, authentication, integrity or connection latency and contention or even communication unavailability in uncovered areas (highway tunnels, etc). On-board DNN signal processing can be done only if a low computational complex algorithm is used and a performing hardware is adopted. Hence, on-board computing units for DNN should be optimized in terms of the ratio between signal processing throughput performance and resources (memory, bandwidth, power consumption, etc.) [15]–[17].

This is the trend that also big players are following like Google, NVIDIA or Intel, that are trying to enter in the autonomous driving market, or the recently announced Full Self Driving (FSD) chip from Tesla. This concept is also the core of the automotive stream in the H2020 European Processor Initiative (embedded HPC for autonomous driving with the BMW group as main technology end user [17], where the article's authors are involved. To address the above issues new computing arithmetic styles are appearing in state of art [18]–[26] to overcome the classic fixed-point (INT) vs. IEEE-754 floating point duality in case of embedded DNN signal processing. Just as an example, Google is proposing BFLOAT16 (Brain FLOAT), equivalent to a standard single-precision floating point value with a truncated mantissa field. Basically, they are less precise than Float16, but with a range similar to Float32. BFLOAT16 are supported in Google cloud TPU and TensorFlow and Intel AI processors. Intel is also proposing flexpoint [18, 19], a 16-bit block floating-point format aiming at replacing Float32. NVIDIA Turing architecture is supporting in its tensor cores Float16 to Float16 or Float32 matrix multiply-add operations, and also 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 units using 8-bit by 8-bit integer multiply and a 32-bit integer addition. Transprecision computing for DNN is also proposed in state of 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 as a compressed floating point representation, where more mantissa bits are used for numbers around 1, and less mantissa bits stepping away from 1, within a fixed-length format with variable sized fields (the exponent bits adapts accordingly, to maintain the format fixed in length).

The rest of this tutorial paper is organized as follows. Section II reviews the onboard computing challenge of DNN signal processing for sensing and machine perception, and the emerging trends in the state of art to solve this issue. Sections III and IV will review computing arithmetic proposed for DNN signal processing as alternative to classic IEEE-754 floats and integer both in academia and industry. In particular, Section IV focuses on the new Posit format. Posit implementations based on software (SW) libraries or custom hardware (HW) accelerator are discussed in Section V. In Section VI a specific focus is on special signal processing functions like sigmoid and hyperbolic tangent for neural networks. Section VI also addresses the problem of the efficient implementation of computing approaches based on Look-up-tables (LUTs), Multiply and Accumulate (MAC), Fused/Exact dot product, Vectorization/loop-unrolling/Intrinsics, low-resolution DNN. Sections VII and VIII discuss tradeoff of DNN signal processing in terms of accuracy vs. complexity, taking care of results we achieved using different types of datasets and DNNs (considering both training and inference phases, or just inference of a pre-trained network). Section IX will draw some conclusions and will discuss upcoming trends in the sector.

II. State-of-the-art review of DNN signal processing in autonomous driving

Autonomous driving is deeply bounded to vehicle navigation, including vehicle self-localisation, motion, mapping and interaction. A relevant survey on trends and technologies for autonomous driving is presented in [27]. Localization task is aimed to know the vehicle pose (position and orientation) referred to a relative or absolute coordinate system. Traditional approaches to localization include satellite communication like GPS. However these signals are typically weak radio ones that can be easily occluded by trees or buildings in a metropolitan scenario. There exist other types of equipment like Inertial Measurements Unit that, combined with GPS, RTK (Real Time Kinematic) and Kalman-based predictors, can solve this problem, but they increase implementation cost. Since the task of constantly knowing the vehicle position is critical one cannot rely only on these type of 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 also its surroundings. An important mapping technique is Simultaneous Localization and Mapping (SLAM, [28]) that allows a vehicle to bypass or minimize the need for satellite navigation. SLAM considers the surrounding as a probability distribution of points rather than a snapshots in time of the context, building a world model making use of lidar sensors or similar. The typical output of these sensors are point clouds representing the surrounding environment, that must be processed in order to give more information about it. In [29] a way to classify lidar images using DNNs is presented. In [30] a benchmark challenge for DNNs for the German Traffic Road Signs (GTRSB) is proposed and in [31] there are some advanced DNN techniques like data augmentation and region of interest extraction to maximize DNN recognition and detection accuracy, reaching top-level accuracy on the road signs recognition and detection benchmarks. Moreover, with the advanced developments in computer vision, vehicles can be equipped with cameras, whose signals can be processed by DNN as well. For example, in [32]–[34] a semantic segmentation of city landscapes challenge is presented, providing a benchmarks for DNNs to prove the 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 like Tesla or Google's Waymo, the use of DNNs in processing lidar or camera signals has become more and more central.

A. *Low-precision DNNs*

Academia and industry have proposed multiple solutions to the problem of reducing the number of bit used to represents DNNs' weights, compressing the size from 32-bit to 16, 8, 4 and even 1-bit, resulting in little-to-none degradation in performance when tested with common DNN tasks and benchmarks. As an emerging trend in state of art, literature is starting to explore the possibility to use the newly introduced Posit representation in order to halve the weights' size though maintaining the same accuracy and to further reduce the weights' size sacrificing little-to-none in DNN accuracy. 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 lead the reduction of weight bits with its Tensor Processing Units (TPU), introducing integer weight types such as 8 and 4-bit integers.

In [36] a novel method is introduced to train neural networks with extremely low precision (eg, 1-bit) weights and activations, at run-time. In [37] the authors studied the training of NNs using low-precision fixed-point computations and evaluated the impact of different rounding techniques.

The vision presented in this work aims to develop neural network accelerator entirely based on Posits, also embedding look-up tables for low-bit Posits such as 4-to-12-bit Posits. In this way we ensure an homogeneity of representations that is lost in the NVIDIA approach, due to the discontinuity introduced when switching from floating point half-precision to 8 or 4-bit integers.

III. Alternative Representations for Reals

In this section we review the most interesting representation for real numbers, which could be used as an alternative to the floating point representation (IEEE-754 standard, 2008, that will be referred simply as Float from now on). In the following we will use an homogeneous representation for the different number representation "Type Bits[,Exp]", where Type is the name of the representation (Float, Posit, Fixed), Bits is the number of bits, and Exp is the number of bits used for the exponent. For fixed point Exp represents the scaling factor to be applied to the number considered signed integer (e.g Fixed16,8 represents value with 8 bit of integer part and 8 of fractional part). For Float when Exp is missing the standard value is assumed: 11, 8 and 5 for Float64, Float32 and Float16 respectively corresponding to binary64, binary32 and binary16 of the IEEE standard.

A. BFLOAT16

The research on DNNs has demonstrated that 16-bit Floats could be enough for many classification problems. From this 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 pre-trained DNN models are usually available with Float32, and thus their lowering to 5 bits of exponent could introduce alterations to the classification and thus could overall affect the classification performance. For this reason, the BFLOAT16 format (Brain Float 16-bit, namely Float16,8 in the present notation) has been recently introduced with 8 bits of exponent instead of 5. Having the same size of the exponent of Float32 the use of BFLOAT16 introduces loss of numerical precision but not loss of dynamic range. Also the conversion with Float32 is bitwise.

B. FlexPoint

Flexpoint numbers [18] are characterized by a *shared tensor exponent* used for all the number representations in a given neural network layer (e.g. 16-bit flexpoint plus 5-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 to Floats: software changes are required to the DNN software libraries. This also makes cumbersome the reuse of pre-trained DNNs.

C. Type-III Uniform numbers: Posits

Type-III uniform numbers are the third proposal of universal numbers, proposed 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 next section. Before that, we present on the next two sub-sections two further representations that are somehow related to Posits.

D. Universal Coding of the Reals using Bisection

The bisection method proposed by Peter Lindstrom in [38] is based on Elias codes. It encodes each real number in a binary string based on bi-secting intervals, starting from the base interval $(-\inf, +\inf)$. 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 allows to build 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 non-hardware-friendly. This suggests that this particular encoding is not so interesting in the high performance hardware accelerator topic discussed here, although also Posit numbers can be generated using this powerful encoding technique.

E. Logarithmic Numbers and the Kulisch Accumulator

As pointed out by Jeff Johnson in [39], a researcher at Facebook AI Research, the problem with floating point operations in hardware is that the transistors needed to perform multiplication and division occupy the main part of the FPU, being significantly more complex than that for addition/subtraction. To overcome this problem, the Logarithmic Number

System (LNS) has been proposed decades ago in [40]. LNS consists in representing a number as $y = 2^x$, i.e., in a pure logarithmic way. This makes multiplication and division just a matter of adding and subtracting logarithmic numbers.

However this requires huge hardware lookup tables 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 hardware multipliers.

In order to avoid common fused multiply and add complexity, the Kulisch accumulation can be used. The idea is not to accumulate with a floating point type but instead maintaining an accumulator in a fixed-point type. As a drawback this approach leads to a significant increase of 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 been demonstrated yet that logarithmic numbers are more effective than Floats for DNNs. Thus more research is clearly needed before resorting to this solution.

IV. A deeper investigation on Posits

Posit numbers have been proposed by John L. 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 by a maximum of four fields (see Fig. 1):

- 1-bit *sign* field S;
- variable length *regime* field R ($1..rebits$);
- *exponent* field E, having a pre-determined *maximum* length of $esbits$ (the field E can even be absent);
- variable length *fraction* field F (can be absent too).

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

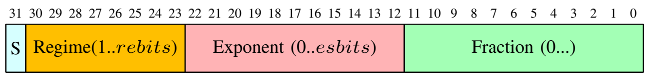

Fig. 1: Illustration of the of 32-bit Posit data type.

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 0s after the sign bit ended by one 1 or, vice versa, by the number of consecutive 1s ended by one 0. 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 Fig. 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 $esbits$ (the a-priori predetermined maximum number of exponent bits). When the field is missing, the exponent e is assumed zero. When less bit than $esbits$ are present, the value of e can be obtained by filling the missing bits with zeros before decoding it (see Fig.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 (first bit equal to one), before decoding it to retrieve k, e and f, the 2's complement of its remaining bits must be computed.

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

Fig. 2: Mapping table between regime bits and k value for a 5-bit string. Amber bits represent the regime bits, brown ones terminate the regime run.

Therefore, let p the integer represented by the Posit bitstring, 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 $f = 0.f_1 f_2 ... f_n$ (the fractional part of the mantissa m ($m = 1 + f$), associated to the F field); the expression that maps the bitset to the real value is:

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

where

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

Notably it is possible to prove that for PositN,0 the numbers in the range $[-1, 1]$ are encoded as signed fixed points over N−1 bits. This property is important for L1 operations discussed later.

Figure 3 shows an example of Posit16,3 (16-bit with max 3 exponent bits) and its decoding procedure.

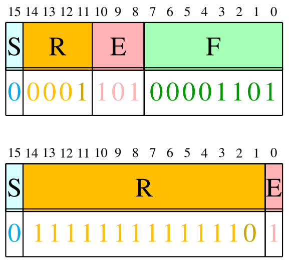

Fig. 3: Two examples of Posit16,3 that is 16-bit Posit with $esbits$=3. For the top case, the associated real value is: $+256^{-3} \cdot 2^5 \cdot (1 + 13/256)$ (13/256 is the value of the fraction, 1 + 13/256 is the value of the mantissa). The final value is therefore $+1.907348 \times 10^{-6} \cdot (1 + 13/256) \cong +2.0042 \times 10^{-6}$. For the bottom case, 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 allows to clarify that: i) the fractional part can be missing, ii) the exponent field can be shorter than its maximum size (in that case the missing bits are assumed zero: the exponent 4 comes from the reconstructed exponent field 100).

A. *Posit advantages over Floats and industrial adoption*

A shown in [26], the main advantages of Posits over IEEE floating points are represented by less waste of representations (such as unique 0 and NaN bit configurations) and higher decimal accuracy when compared to same bit length floating point. Moreover, the simplicity of the Posit number systems theoretically allows a more hardware friendly implementation, simplifying circuitry 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 Corp, as reported by Gustafson during a recent talk [41].

V. Software and Hardware Implementations of Posits

A. *Software Implementations*

Having software implementation of Posit arithmetic is useful in order to test the applicability of the type to existent libraries and algorithms in order to compare performances against the traditional floats, also in the absence of proper hardware support for Posit operations.

1) SoftPosit: This is a library, endorsed by NGA (Next Generation Arithmetic committee). Positive factors include multi-platform, supporting C, C++, Julia and Python. However it presents hard-coded Posit configurations and non-modern implementation, without templatized classes for the various configurations. It also lacks support for tabulated Posits.

2) bfp (beyond floating point): BFP is one of the first C++ Posit arithmetic libraries developed. However it is still incomplete and does not support Posit tabulation.

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

4) cppPosit: This library (available in [42]), developed by the authors of the present work, exploits some of the modern C++ features like templates (i.e., generic programming) and traits. It supports Posit tabulation and logic separation between frontend interface and backend underlying type used for computation: the frontend is the Posit number expressed in its packed form, while the backend allows to choose 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 of integer operations performed in ALU. At level 2, Posit data is extracted to its fields (sign, regime, exponent, and fraction), with no need to compute the exponent completely. Computations are performed on these fields and the cost includes encoding and decoding of the format. At level 3 we have the unpacked version that is completely built (including sign, exponent, fraction). In addition to level 2 operations, here there is the need to build the full exponent. At level 4 the unpacked version is used to perform the operations in either software or hardware floating point or using fixed point representations. The most efficient level is of course the 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 I reports most important L1 operations provided by the library. The library offers the possibility to use different backends for Posit operations:

Operation	Approximated	Requirements
$2 \cdot x$	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*

TABLE I: cppPosit most important L1 operations, stating whether the operation produces an exact or an approximated result and reporting the requirements to be fulfilled. For instance notice how $1 - x$ can be computed using fast bit manipulations only when $x \in [-1, 1]$.

- Fixed number backend (using a quire-like approach);
- Tabulated backend (see VI-B);
- Floating point backend: either SW (SoftFloat) or HW (FPU).

Each L3 operation in the cppPosit library undergoes three different phases: i) decode, ii) operation backend, iii) 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) builtin function
- Backend:
  - Fixed: requires big integer (64-128 bits) support
  - Float: requires a Floating Point Unit (FPU)
  - SoftFloat: requires 32/64-bit integer manipulations
- Encode: bitwise operations

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

B. *Hardware Implementations of the Posit Processing Unit*

Some work has already been done to implement Posit units on FPGAs, in order to provide efficient and optimized hardware implementation of Posit arithmetic. In [44] an algorithmic flow and architecture generator for Posit numbers

TABLE II: Requirements support of Intel and ARM for the cppPosit library.

Requirements	Intel 7th gen. (Kaby Lake)	ARMv8
CLZ built-in	✓	✓
Big-integer	✓(1 single instr.)	✓(2 instr. needed)
FPU	✓	✓
Integer manip.	✓	✓
Bitwise ops	✓	✓

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 floating point unit, while the other determines the impact of the design on the hardware. This has been implemented on a Xilinx Virtex-6 device, resulting in around 600 FPGA slices for 32-bit Posit adder and 300 for a 16-bit Posit adder.

In [45] a Posit core generator called POSGEN is proposed. In addition, the FPGA design has been enriched with an extension of the BLAS library for the Posit numbers called PBLAS, in order to connect the FPGA through the Intel OpenCL libraries. The results show that the maximum frequency reached by the proposed implementation matches the state-of-art FloPoCo floating point implementation. However the area consumed by the POSGEN implementation is much higher than the FloPoCo one.

Another Posit arithmetic core (called PAU, Posit Arithmetic Unit) generator is presented in [46] where generators for Posit adder and multiplier are proposed. The design results show a reduction in area occupation referring to [44] both for adder and multiplier, as well as a reduction in power consumption for 8-bit Posits. For 16-bit Posits the results are overturned in favour 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 on it results that a 32-bit Posit adder occupies less area and has a lower delay than a 32-bit Float adder. 32-bit multiplier instead occupies the same area but with higher delay. Finally, a 16-bit adder occupies an higher area with higher delay.

In [47] another Posit arithmetic core generator has been introduced, called PACoGEN. The work presents different generators for HDL adder/subtractor and multiplier/division cores. An interesting aspect of this implementation is the pipelined Posit arithmetic architecture, aimed to increase the throughput of the unit 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 shows that the proposed implementation has a lower *area (LUT)* $\cdot$ *period (ns)* when compared to literature proposal such as [46]. However, when the design is compared to standard FP ones, results show that 32-bit Posit adder/multiplier units occupy more area than some 32-bit FP ones.

An accelerator for Posit-based BLAS operations is proposed in [48]. The work presents a modular framework for Posit arithmetic with the common 3-step dataflow: Posit data extraction, operation and construction. The implementation consists in a Posit adder, multiplier and a Posit accumulator. The BLAS library proposed enables vectorized operations such as element-wise addition, subtraction and multiplication, as well as dot product and vector sum. Experimental results show a consistent speed-up obtained when using the vectorized approach when compared to a software 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 a FPU realization. Having a 32-bit hardware 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 hardware Posit unit if its area is less than a 32-bit FPU one, since 16-bit Posit achieve similar performance of 32-bit Floats in different application fields (in Section VIII we show that in DNNs even a Posit8 can match the performance of a Float32).

VI. Posit-based DNNs for Signal Processing

Non-linear activation functions are a very important part in DNNs. Its efficient implementation is therefore crucial. In the next paragraphs we will see how some widely used activation function can be efficiently computed when using Posits.

A. DNN Activation Functions

In this subsection 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 V-A4).

1) Sigmoid: The sigmoid function:

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

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

2) Hyperbolic Tangent: In order to solve said problem, an expression for the hyperbolic tangent has been derived, using a linear combination of the sigmoid function:

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

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

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

In order to have an L1 expression we initially restrict the domain to the negative numbers only. Doubling operation and sigmoid function are L1 when using 0 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 I. Finally thanks to *tanh* symmetry we can extend back the domain also 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 time faster than the exact tanh version. Moreover, we computed the mean squared error between the two, resulting in $mse = 2.947 \cdot 10^{-3}$ 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 like the hyperbolic tangent and the effects of the flattening of the ReLU for negative numbers.

$$\mathrm{ELU}(x) = \begin{cases} e^x - 1, \text{ if } x \leq 0 \\ x \text{ otherwhise} \end{cases}$$

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

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

If we switch from Sigmoid to the fast approximated version already exploited with the hyperbolic tangent, we can get a fast approximation of the ELU (called FastELU). Table III 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 validation accuracy were obtainable. Then we tested the three mentioned trained models with the different Posit types, reporting 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 less degradation in terms of accuracy with FastELU/ELU rather than RELU. In terms of timing the FastELU and RELU are comparable with PositN,0 being both L1 operations, while ELU is costlier. More mathematical details on the FastTanh and the FastELU can be found in [50].

B. The Look-up-table approach

When using a low number of bits, the use of LUT becomes appealing very soon. In theory, one could profile a specific application (i.e., computing the histogram of most used values and the most significant range), and then create an ad-hoc series of values. For this set of values, one only has to compute the four LUT for the four elementary

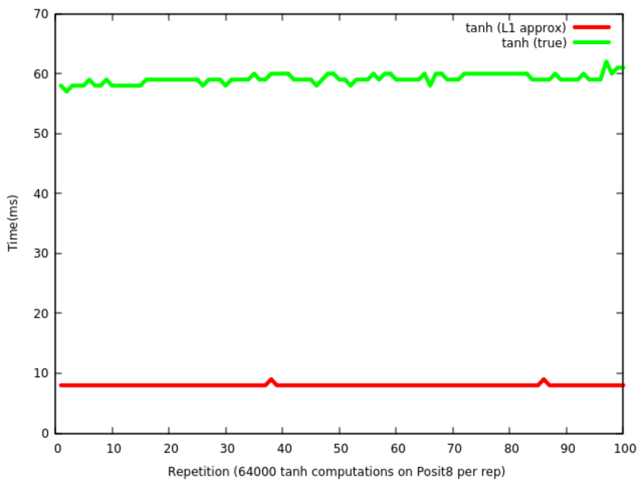

Fig. 4: Time comparison in various repetitions of $\sim 60k$ executions of tanh and FastTanh for a Posit16,0 (benchmarks were executed on a Intel 7th generation (Kaby Lake) `Intel i7-7560U` processor with 2 cores @ 2.4 GHz.). The latter shows to be around six time faster with a computed mean squared error of $2.947 \cdot 10^{-3}$.

operations, plus the tabulation of significant unary functions (exp, log, trigonometric functions, sqrt, square, etc.). There also exists some optimized soft mathematical libraries in the Sun Cephes collection ([51]). The collection consists in more than 400 mathematical functions entirely implemented in *C*, mostly delivered in different arithmetic precisions (32, 64, 80, 96, 144, and 336 bits operands).

1) Look-up-tables for Posits: 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 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 a 8-bit Posit represented over a 8-bit unsigned integer type a single table occupies $64kB$. In order to reduce table size symmetry of addition/subtraction operations can be exploited to halve table size and number. Moreover multiplication and division tables can be discarded by exploiting logarithm properties thus just using the addition/subtraction tables.

C. MAC: Multiply and Accumulate

The task of multiplying two numbers and summing the result into an accumulator is a very common operation during DNN operations (such as convolution or matrix multiplication). The presence of an hardware multiplier-accumulator is crucial since it helps reducing by one the number of roundings involved in the computation at each step. In [52] is presented the implementation of an exact MAC for low-precision Posits and other floating/fixed-point types, resulting in 8-bit Posit matching or even overcoming 32-bit Floats.

TABLE III: Comparison of different activation functions when applied to neural network for traffic sign classification. Benchmarks were executed on a Intel 7th generation (Kaby Lake) `Intel i7-7560U` processor with 2 cores @ 2.4 GHz. [1]NCT: Normalized Computing Time (Posit computing times are normalized against the Posit16,0 computing times)

GTRSB									
Activation	FastELU			RELU			ELU		
	Acc. (%)	Time (ms)	NCT[1]	Acc. (%)	Time (ms)	NCT[1]	Acc. (%)	Time (s)	NCT[1]
Posit16,0	94.0	5.8	–	92.0	5.0	–	94.2	6.4	–
Posit14,0	94.0	4.6	0.79	92.0	4.3	0.86	94.2	5.2	0.81
Posit12,0	94.0	4.6	0.79	92.0	4.3	0.86	94.2	5.1	0.79
Posit10,0	94.0	4.6	0.79	92.0	4.2	0.84	94.2	5.0	0.78
Posit8,0	92.0	4.6	0.79	86.8	4.0	0.8	91.8	5.0	0.78

D. *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 deep neural networks during convolution operations and overflows can occur with high probability during the accumulation of term products. In order to avoid most of these overflows two solutions can be adopted:

1) Fused Dot Product: While a MAC technique computes the product result, rounds it, adds it to the accumulator and then round it again, a fused dot product (also known as fused multiply add) computes the entire expression at the maximum available precision, typically using an accumulator that has twice the bit of the single operands. In [26] the potentiality of Posits in overcoming rounding issues when using fused operations are 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 as well as storage requirements.

2) Exact Dot Product: The exact dot product (EDP) technique makes use of the concept of quires (very high bit-count scratch area) as accumulator, deferring rounding only at the very last operation, thus minimizing rounding errors. The concept of quires has been introduced by Ulrich Kulisch in [53], in order to minimize the number of transistor 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 back the result to the original low-precision type. In order to have the quire being able not to underflow or overflow during these operations we need to dimension it depending on the Posit configuration[1]. Suppose to have a $totbits$-bit Posit, the maximum possible value for the Posit will be $maxpos = u^{totbits-2}$, while the minimum possible value will be $minpos = \frac{1}{maxpos}$, where $u = 2^{2^{esbits}}$; each number is then an integer multiple of $minpos$. Suppose we need to perform the following dot-product $\{maxpos, minpos\} \cdot \{maxpos, minpos\}$, we'll need the quire to be able to accomodate the value $maxpos^2/minpos^2$. After some transformation, we can compute the maximum value to hold as

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

Moreover one bit has to be reserved for the sign and more bits to handle the sum (e.g. Gustafson chooses 30 more bits to guarantee the absence of overflows). Practically, for example, this means that with a 8-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 (4 64-bit registers) and for 32-bit ($esbits = 2$) Posit 512-bit (8 64-bit registers).

[1]https://posithub.org/docs/Posits4.pdf

E. *Kalray MPPA approach*

In order to address the challenges of high-performance embedded computing with time-predictability, Kalray has been refining a homogeneous manycore architecture called MPPA (Massively Parallel Processor Array) based on VLIW cores. On the 3rd-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 '/' to describe the two bandwidths of the multiplicand and the accumulator. Exploitation of INT8/32 operations relies on the TensorFlow Lite quantization support [55], while exploitation the Float16/32 artithmetic by standard frameworks is the same as for NVIDIA GPGPUs. However, unlike the NVIDIA tensor cores, the Kalray MPPA3 coprocessors perform exact dot-product inside the Float16.32 matrix multiply-accumulate operations, by applying Kulisch's principles on a 80+ϵ 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 MPPA3 coprocessors. Conversely, the Posit8,2 numbers include 8 values of magnitude 65536 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 hardware costs and application benefits of using Posit8,0 numbers as compressed format for Float32 network parameters is on-going. This evaluation should lead to the inclusion of new arithmetic instructions to expand Posit8,0 to Float16 in the MPPA IP delivered to the H2020 European Processor Initiative.

Preliminary results obtained comparing the use of Float32, Float16 and Posit8,E (with a E from 0 to 3) for data storage (while computation is still done in Float32) during the inference phase using network models for both classification task (e.g. SqueezeNet, Alexnet, VGG16, VGG19, GoogleNet and Custom CNN on MNIST and CIFAR100) and detection tasks (e.g. YOLOv3) show that Posit8,1 or Posit8,2 offer the best performance, with an accuracy loss below 1% vs. Float32, but with a data compression of factor 4. This will lead to reduced complexity for data transfer and storage that are dominating DNN applications.

To be noted that: i) the networks where pre-trained using Float32, and ii) the used datasets in the reported results had thousand of images. Indeed, the ILSVRC2012 (ImageNet Large Scale Visual Recognition Challenge 2012) dataset has been used for classification and the VOC2012 (Visual Object Classes Challenge 2012) dataset for detection.

F. Vectorization of Posit operations (tested on random images)

While in the absence of proper hardware support for Posits (i.e. Posit Processing Unit) we can still accelerate DNN core functions and operators using already existing hardware accelerators. This is the case of ARM Scalable Vector Extension (SVE) SIMD engine. We have also ported our cppPosit library to provide vectorized version of Posit functions exploiting ARM SVE library. When talking about vectorized functions, L1 Operations are the easiest ones to vectorize. In fact, since they only rely on integer arithmetic and logic, we can effortlessly exploit native ARM SVE vectorization of integer operations. Benchmarks were executed on a HiSilicon Hi1616 CPUwith 32@2.4GHz ARM Cortex-A72, using the ARM SVE Instruction Emulator. Table IV shows some timing results between vectorized and non-vectorized approaches. Furthermore, we have provided an interface between the Posit floating point backend and the ARM SVE types in order to vectorize L3/4 operations as well. This allowed to implement Posit-accelerated version of convolution and pooling operations. Table V shows an example of timing results with 3×3 convolution and max-pooling operations. Finally, Table VI shows vectorization performance in terms of processing time on low-precision inference on Posit8,0. Performance have been obtained on the tinyDNN library, on various very deep neural netwoks. 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 hardware is not available at moment of writing and all benchmarks are executed inside the ARM SVE instruction emulator.

TABLE IV: L1 operations performance processing time (in milliseconds) comparison between non-vectorized (naive) and vectorized (SVE-X) approaches. Each timing result comes from function computation on a vector of 8192 items

	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	2.60
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	0.50	0.35	0.47

TABLE V: 3×3 Convolution and Pooling processing time (in milliseconds) comparison on two common Posit configuration with 225×225 random images. The Naive approach is the non-vectorized one. The other approaches are with incremental SVE-vector registers.

	Max 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

VII. DNN Signal Processing performance: accuracy and complexity

In [52] and [57] Carmichael et al. show an architecture using Posits in deep neural networks called Deep Positron using exact multiply and accumulate technique (EMAC) on 8-bit low precision formats. The architecture has been tested on the MNIST, Fashion-MNIST and datasets reporting no drop in accuracy with regards to Float32. Another approach to deep learning with low-bit numbers has been tested in [39], using logarithmic numbers with a *ResNet-50* architecture on *Imagenet*, resulting in a drop of 0.90% percentage point when shifting from Float32 to logarithmic representation. We have integrated the cppPosit library in a deep neural network C++ library called tiny-DNN [58], that is capable of support various different computing arithmetic

TABLE VI: Image processing time (in seconds) for various very deep neural network models using Posit8,0. For this benchmark random RGB 224×224 images are employed.

Version	**Alexnet**	**Resnet34**	**VGG16**	**VGG19**	**Resnet150**
	Time(s)	**Time(s)**	**Time(s)**	**Time(s)**	**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	41.30	47.68
SVE-1024	2.44	8.92	36.06	41.23	47.60
SVE-2048	2.34	8.90	35.97	41.13	47.48

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 German Traffic Road Sign Recognition Benchmark (GTRSB) using the fused dot product (FDP) technique. For MNIST dataset 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 $16, 14, 12, 10$ bits we registered no drop in accuracy from Float32 to the Posit type.

VIII. Benchmark Datasets and Examples of Achievable Results

We have considered different standard datasets, like the one shown in Figure 5, and standard CNN architectures, like the one shown in Figure 6. In particular, for the MNIST and GTSRB benchmarks we trained customized CNN variants of that reported in Figure 6, including Posit-related optimizations to convolutional and activation layers. For the Fashion-MNIST benchmark we used a pre-trained model with starting accuracy of 95%. For CIFAR-10 we used VGG16 pre-trained model [59]. All the networks were initially trained using Float32 and then tested on the corresponding test sets, converting each Float32 trained model using different Posit configurations. Furthermore, in order to provide a fair timing-accuracy tradeoff comparison, the Float32 model has been tested exploiting the SoftFloat library for software-emulated floating point numbers.

A. MNIST, Fashion-MNIST and CIFAR-10

Table VII 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 more complex drop-in replacement for the MNIST dataset, providing more general classes to be recognized (such as fashion products). Furthermore, CIFAR-10 consists in an even more complex task, bringing 3-channel images in the dataset.

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 with 10 and 8 bits can replace Float32 with some drop in accuracy. The same holds for the Fashion-MNIST dataset.

Note how the processing time (on an Intel 7th generation (Kaby Lake) i7 processor) for single image inference of VGG16 model on a CIFAR-10 sample is expressed in seconds, highlighting the infeasibility of these model on traditional CPU architectures. However we are moving towards GPU-enabled DNN libraries as described in Section IX. For comparison, an entire training epoch of $60k$ CIFAR-10 samples on a Resnet-50 architecture only takes around 30 seconds on a dual GPU (Tesla T4) configuration, thus only 0.5 milliseconds for forward and backward passes (including weight update).

To be noted also that, to make the comparison fair, we compare in Tables VI and VII the *software* implementation of Posits (using our developed cppPosit library) with a *software* implementation of Floats (the SoftFloat library). From Tables VI and VII we can observe that moving from SoftFloat32 to Posit8,0 we get (roughly) the same classification accuracy on all the considered datasets, but with a reduction in computing time of about a factor 3.

B. Automotive Benchmarks: The traffic sign recognition problem

Fig. 5: GTRSB dataset example

In this subsection we report the results obtained on a classification benchmark related to assisted/autonomous driving. Benchmarks were executed on an Intel 7th generation (Kaby Lake) `Intel i7-7560U` processor with 2 cores @ 2.4 GHz. German Traffic Road Sign Recognition Benchmark is a baseline benchmark for road sign recognition, being very interesting as automotive task. Table VIII shows that also in this case Posits from 12 to 16 bits and even 10 bits can be a perfect replacement for Float32 while Posit8,0 performs good with a little drop in accuracy.

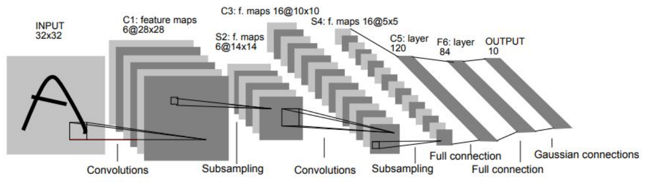

Fig. 6: LeNet5 architecture as described in [49]. Some customisations have been added to the network in order to better fit our goals: the activation function has been changed to FastTanh (as described before) for the MNIST dataset and to a fast approximation of ELU for the GTSRB dataset. The input size of the first layer has been extended to hold the $64 \times 64 \times 3$ color images of the GTSRB datasets.

Fig. 7: Cityscapes Dataset example of semantic segmentation of a road in Stuttgart.

We have also started an activity to assess the performance of Posits using the Yolo (You Only Look Once) approach [60, 61] and on Apollo [62] (http://apollo.auto/) heterogeneous framework and the results achieved confirm

TABLE VII: Accuracy and processing time obtained on MNIST, Fashion-MNIST and CIFAR-10 datasets. Processing time is evaluated as the mean per-sample inference time on the testset of the relative dataset.
[1]NCT: Normalized Computing Time (Posit computing times are normalized against SoftFloat32 computing times)

Type	MNIST			Fashion-MNIST			CIFAR-10		
	Acc. (%)	Time (ms)	NCT[1]	Acc. (%)	Time (ms)	NCT[1]	Acc. (%)	Time (s)	NCT[1]
SoftFloat32	99.4%	8.8	–	95.0%	41.9	–	93.75%	7.75	–
Posit16,0	99.4%	5.2	0.59	95.0%	13.6	0.32	93.75%	2.55	0.32
Posit14,0	99.4%	4.6	0.52	95.0%	13.5	0.32	93.75%	2.49	0.32
Posit12,0	99.4%	4.6	0.52	95.0%	13.5	0.32	93.75%	2.44	0.31
Posit10,0	99.3%	4.6	0.52	95.0%	13.4	0.32	93.75%	2.40	0.30
Posit8,0	98.5%	3.8	0.43	94.0%	13.4	0.32	85.0%	2.34	0.30

TABLE VIII: Accuracy-processing time trade-off obtained on the German Traffic Road Sign Benchmark dataset.
[1]NCT: Normalized Computing Time (Posit computing times are normalized against SoftFloat32 computing times)

Type	GTRSB		
	Acc. (%)	Time (ms)	NCT[1]
SoftFloat32	94.0%	15.86	–
Posit16,0	94.0%	6.37	0.40
Posit14,0	94.0%	5.21	0.32
Posit12,0	94.0%	5.08	0.32
Posit10,0	94.0%	5.0	0.31
Posit8,0	93.8%	4.0	0.25

what already obtained above with GTRSB, MNIST and Fashion-MNIST datasets. Moreover we started an activity to asses Posit performances in semantic segmentation tasks (such pixel-level or instance-level classification, [33, 34]) on famous datasets like CityScapes, see Figure 7. The results we are obtaining are in line with those obtained on, MNIST and fashion-MNIST and GTRS benchmark datasets.

C. *k-Nearest Neighbours results*

The k-Nearest Neighbours (k-NN) algorithm is ubiquitous in pattern recognition problems. It can be used to segment images, or 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 algorithms finds the K nearest neighbours of a given point, from those in a given dataset. We have compared the performance of the k-NN when using Posits and Floats and, again, we have found that the accuracy of a Posit16,0 is very close to that of Float32 (see Fig. 8), and that a Posit8,0 outperforms a Float16. These results have been obtained on a single dataset, but scaling it multiple times in order to reduce the dynamic range of the input data (thus allowing 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 application and thus this means that implementing Posit-based HW accelerators will be beneficial for a number of different applications.

D. *Next Experiments*

We are working towards the implementation of other fast approximated functions (e.g. ELU). We are currently porting our cppPosit-based tinyDNN library on the ARM

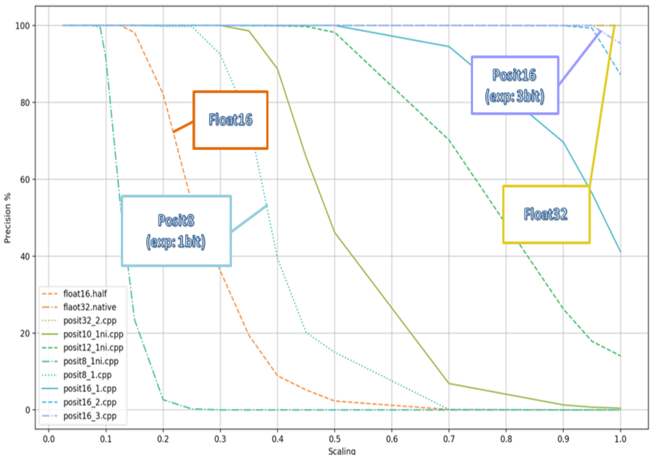

Fig. 8: Performance of the k-NN using different data types, on a single dataset using different values for the scaling factor.

instruction emulator used within the H2020 EPI project (European Processor Initiative, [64]), to exploit the Scalable Vector Extension (SVE2) as much as possible (providing a vectorization backend for the cppPosit library).

We are also planning to test our software on available simulators like GEM5, SESAM and MUSA, in order to provide useful feedbacks to the ongoing EPI processor co-design process.

IX. Conclusions and Roadmaps

In this work we have reviewed the state-of-the-art of DNN signal processing for autonomous driving application 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 IEEE-754 standard, and we have assessed its potentialities in autonomous driving applications. Implementations with both SW-libraries or HW-SW embedded systems, from academia and industry, have been discussed. The achieved results when combining Posit arithmetic with DNN are promising in terms of trade-off between accuracy and processing time. From this and related works, it is clear that the current challenges are i) the development of real-time and low-power accelerators for performing DNN inference at the edge, ii) the development of methods for DNN verification and validation, with the high coverage rates required by the standards for safety-critical applications and iii) moving towards a GPU-enabled DNN library, such as Tensorflow, in order to build, train and evaluate even more complex models, once integrated with our cppPosit library. Furthermore we plan to test our approach on GPU-enabled ARM devices such as the NVIDIA Jetson boards and on mobile devices that do not employ GPUs or even without the FPU.

Acknowledgment

This work is partially funded by H2020 European Processor Initiative (grant agreement No 826647) and partially by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

References

[1] S. Saponara and B. Neri, "Radar sensor signal acquisition and multi-dimensional FFT processing for surveillance applications in transport systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 66, no. 4, pp. 604–615, 2017.

[2] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1038–1051, 2019.

[3] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, 2018. [Online]. Available: https://doi.org/10.4271/J3016_201806

[4] J. Royo-Alvarez et al., Ed., *From Signal Processing to Machine Learning*. John Wiley & Sons, Ltd, 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 Processing Magazine*, vol. 35, pp. 180–177, 2018.

[6] 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 Processing Magazine*, vol. 34, no. 6, pp. 85–95, 2017.

[8] A. Arnab, S. Zheng, S. Jayasumana, B. Romera-Paredes, M. Larsson, A. Kirillov, B. Savchynskyy, C. Rother, F. Kahl, and P. H. S. Torr, "Conditional random fields meet deep neural networks for semantic segmentation: Combining probabilistic graphical models with deep learning for structured prediction," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 37–52, 2018.

[9] S. F. Dodge and L. J. Karam, "Quality robust mixtures of deep neural networks," *IEEE Transactions on Image Processing*, vol. 27, no. 11, pp. 5553–5562, 2018.

[10] P. Nousi, A. Tefas, and I. Pitas, "Deep convolutional feature histograms for visual object tracking," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 8375–8379.

[11] A. Voulodimos, N. D. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," in *Comp. Int. and Neurosc.*, 2018.

[12] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[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 Processing Magazine*, vol. 35, no. 1, pp. 84–100, 2018.

[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 Processing Magazine*, vol. 35, no. 1, pp. 126–136, Jan 2018.

[16] P. Nousi, E. Patsiouras, A. Tefas, and I. Pitas, "Convolutional neural networks for visual information analysis with limited computing resources," in *25th IEEE International Conference on Image Processing (ICIP'18)*, 2018, pp. 321–325.

[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 Technical Paper 2019-01-0118, 2019.

[18] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in 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 *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018, pp. 1–4.

[20] G. Tagliavini, A. Marongiu, and L. Benini, "Flexfloat: A software library for transprecision computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 145–156, 2020.

[21] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Toms, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1105–1110.

[22] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating deep convolutional networks using low-precision and sparsity," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 2861–2865.

[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 *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 1393–1397.

[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," *CoRR*, vol. abs/1609.07061, 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 *2018 International Conference of Electrical and Electronic Technologies for Automotive*, 2018, pp. 1–6.

[26] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[27] P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha, *Autonomous Vehicles: State of the Art, Future Trends, and Challenges*. Cham: Springer International Publishing, 2019, pp. 347–367. [Online]. Available: https://doi.org/10.1007/978-3-030-12157-0_16

[28] A. Woo, B. Fidan, and W. W. Melek, *Localization for Autonomous Driving*. John Wiley & Sons, Ltd, 2019, ch. 29, pp. 1051–1087.

[29] Y. Wenhui and Y. Fan, "Lidar image classification based on convolutional neural networks," in *2017 International Conference on Computer Network, Electronic and Automation (ICCNEA)*, 2017, pp. 221–225.

[30] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, vol. 32, pp. 323 – 332, 2012, selected Papers from IJCNN 2011.

[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: Springer International Publishing, 2018, pp. 604–613.

[32] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[33] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *ECCV*, 2018.

[34] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," in *ICCV*, 2019.

[35] M. Courbariaux, Y. Bengio, 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. 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," *The Journal of Machine Learning Research*, 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 *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 1737–1746. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045303

[38] P. Lindstrom, "Universal coding of the reals using bisection," in *Proceedings of the Conference for Next Generation Arithmetic 2019 (CoNGA'19)*, 2019, pp. 7:1–7:10.

[39] J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 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 *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*, 2003, pp. 253–261.

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

[42] E. Ruffaldi, "cppPosit," 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*. Cham: Springer International Publishing, 2020, pp. 213–221.

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

[45] A. Podobas and S. Matsuoka, "Hardware implementation of posits and their application in FPGAs," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 138–145.

[46] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 334–341.

[47] M. K. Jaiswal and H. K.-. So, "Pacogen: A hardware posit arithmetic core generator," *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[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 *Proceedings of the Conference for Next Generation Arithmetic 2019 (CoNGA'19)*, 2019, pp. 5:1–5:10.

[49] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[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, 2020.

[51] "Cephes mathematical function library," 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 *Proceedings of the Conference for Next Generation Arithmetic 2019 (CoNGA'19)*, 2019, pp. 3:1–3:9.

[53] U. Kulisch, "An axiomatic approach to rounded computations," *Numerische Mathematik*, vol. 18, no. 1, pp. 1–17, 1971. [Online]. Available: https://doi.org/10.1007/BF01398455

[54] B. D. de Dinechin, "Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore Processor," in *56th ACM/IEEE Design Automation Conference (DAC'19)*, 2019, p. 154.

[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 *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, 2018, pp. 2704–2713.

[56] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 106–113.

[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 & Test in Europe Conference & Exhibition (DATE'19)*, 2019, pp. 1421–1426.

[58] https://github.com/tiny-dnn/tiny-dnn.

[59] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[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 *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 305–317.

[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 Singapore, 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 *56th ACM/IEEE Design Automation Conference (DAC'19)*, 2019, pp. 1–6.

[63] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Novel arithmetics to accelerate machine learning classifiers in autonomous driving applications," in *in Proc. of the 26th IEEE International Conference on Electronics Circuits and Systems (ICECS'19)*, 2019.

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

Marco Cococcioni (IEEE Senior Member) is an Associate Professor at the Department of Information Engineering of the University of Pisa since 2016. He has been the general chair of three IEEE conferences and program committee member of 50+ international conferences in the area of computational intelligence. He is in the editorial board of four journals indexed by Scopus. He is member of three IEEE task forces: Genetic Fuzzy Systems, Computational Intelligence in Security and Defense, and Intelligent System Applications. Prof. Cococcioni has co-authored more than 90 scientific contributions and he is a Senior Member of both IEEE and ACM.

Federico Rossi is a Ph.D student of the Information Engineering Department at University of Pisa. In 2019 he received his Master Degree in Computer Engineering magna cum laude. He is currently involved in the European Processor Initiative (EPI) project. His research topics include alternative real number representations and their applications to Deep Neural Networks for the automotive environment.

Emanuele Ruffaldi is senior software engineer at MMI S.p.A. (IT) working on robotic assisted microsurgery. Formerly he has been Assistant Professor at Scuola Superiore Sant'Anna in the Perceptual Robotics laboratory, Pisa, Italy. His research interests are in the field of machine learning for HRI and embedded artificial intelligence. He is Senior IEEE Member and has served IEEE as Publicity Chair for the Haptics TC. He co-authored more than 100 scientific publications and 1 patent.

Sergio Saponara is Full Professor at University of Pisa, responsible of Automotive Electronics, Hardware Security and Embedded System activities as well as President of BSc and MSc degrees in Electronic Engineering. He is also director of a summer school on Industrial IoT and a specialization course on Automotive powertrain electrification. He is responsible for University of Pisa of the European Processor Initiative (EPI) project. He is IEEE Distinguished Lecturer and Associate Editor of many peer-reviewed journals. He co-authored more than 300 scientific publications and about 20 patents.

Benoît Dupont de Dinechin is CTO of Kalray. He is the main architect of the Kalray VLIW cores, and the co-architect of the MPPA processors. Benot defined the Kalray software roadmap and still contributes to its implementation. Before joining Kalray, Benoît was leading R&D for the Software Tools division at STMicrelectronics, becoming a Fellow in 2008. Prior to STMicrelectronics, Benoît developed the software pipeliner of the Cray T3E production compilers at the Cray Research park, MN, USA. Benot earned an engineering degree from ISAE-SUPAERO (Toulouse), and a doctoral degree in computer systems from the University Pierre et Marie Curie (Paris). He completed his post-doctoral studies at the McGill University, Canada.