Journal of Real-Time Image Processing (2020) 17:759–771
https://doi.org/10.1007/s11554-020-00984-x

SPECIAL ISSUE PAPER


# Fast deep neural networks for image processing using posits and ARM scalable vector extension

**Marco Cococcioni[1] · Federico Rossi[1] · Emanuele Ruffaldi[2] · Sergio Saponara[1]**

Received: 17 February 2020 / Accepted: 12 May 2020 / Published online: 18 May 2020


**Abstract**
With the advent of image processing and computer vision for automotive under real-time constraints, the need for fast and architecture-optimized arithmetic operations is crucial. Alternative and efficient representations for real numbers are starting to be explored, and among them, the recently introduced posit™ number system is highly promising. Furthermore, with the implementation of the architecture-specific mathematical library thoroughly targeting single-instruction multiple-data (SIMD) engines, the acceleration provided to deep neural networks framework is increasing. In this paper, we present the implementation of some core image processing operations exploiting the posit arithmetic and the ARM scalable vector extension SIMD engine. Moreover, we present applications of real-time image processing to the autonomous driving scenario, presenting benchmarks on the tinyDNN deep neural network (DNN) framework.

**Keywords** Deep neural networks (DNNs) · Posit arithmetic · Scalable vector extension · Auto-vectorization · Real-time image processing · Autonomous driving

## 1 Introduction

Nowadays, one of the most fruitful topics that exploit the pervasiveness of DNNs is image processing in the automotive industry. This field brings new problems and challenges to DNNs. On the one side, there is the need to reduce network architecture and computation complexity to better accomplish real-time tasks in resource-constrained devices. On the other side, there is the need to target-specific platform DNN accelerators [e.g. NVIDIA cuDNN for NVIDIA graphical processing units (GPUs)] to provide substantial speed-ups to neural network processing, in the both training and inference phases. On the complexity reduction side, one of the most explored and interesting fields is the alternative representation of real numbers, to reduce the number of bits used to represent the weights of the DNNs. Some ideas have already been proposed by industries such as Google (Brain Float—BFloat16—[1]), Intel (Flexpoint—FP16—[2, 3]) and Facebook AI Group [4]. Another promising representation that diverges from the floating-point standard is the posit number system [5–7]. This type has been proven to be a perfect drop-in replacement of 32-bit IEEE 754 floats in machine learning, using just 16 bits [8–13]. Moreover, it has been productively exploited in low-precision inference down to 8-bit posit representation, with very little degradation of network inference accuracy. Furthermore, as also explained in Sect. 2 and in [9], this number system can be exploited to build fast, approximated and efficient activation functions for neural networks like the sigmoid function by only using the already existent arithmetic logic unit (ALU) within the CPU. On the side of target-specific platform accelerators, the ubiquity of operations such as dot products, matrix multiplications and filter convolutions points out the need for optimized routines able to increase the throughput for these operations. While the spread of GPUs in this field is relevant, the use of such components may be precluded by both high implementation costs and low-power requirements. Microprocessor industries have

✉ Federico Rossi
federico.rossi@ing.unipi.it

Marco Cococcioni
marco.cococcioni@unipi.it

Emanuele Ruffaldi
emanuele.ruffaldi@mmicro.com

Sergio Saponara
sergio.saponara@unipi.it

[1] University of Pisa, Pisa, Italy

[2] MMI spa, Calci, Italy

already moved towards this direction providing a vectorized extension of their instruction set architecture (ISA). In particular, ARM has firstly proposed the NEON instruction set, then evolved and improved with the ARM scalable vector extension (SVE) [14, 15]. Furthermore, ARM has already developed a deep neural network library that supports its NEON vectorization backend [16], but at time of writing it lacks the SVE support. This extension, along with the ARM compiler, allows producing executable binaries exploiting the SVE instruction set in two dimensions. One dimension is the auto-vectorization approach, with the compiler autonomously producing vectorized instructions exploiting data parallelism in the code (e.g. loop unrolling). The other dimension is the explicit use of specific high-level instructions to instrument vectorization in an explicit way. This is possible thanks to the ARM C Language Extension (ACLE) for SVE [14]. Combining the reduction in information size with the vectorization is thus very interesting. If we halve the bits of a given representation without losing decimal accuracy, we can fit twice the elements in the same vector register, increasing the overall throughput. In this paper, we will develop a vectorized extension for the cppPosit C++ posit arithmetic library, following both the approaches. Then, this extension will be tested against common DNN and machine learning operations and in the tinyDNN C++ DNN library.

### 1.1 Organization of the paper

In Sect. 2, we present the posit format and its properties along with some interesting arithmetic operators developed only using integer arithmetic. In Sect. 4, we are going to summarize the main characteristic of the new ARM SVE architecture, pointing out the useful tools and approaches that we can exploit for the development of our vectorized backend. In Sect. 5, we are going to present the cppPosit library developed by the authors and to propose a vectorized extensions for it, providing implementations for common operations (such as dot product and convolution) and addressing the issues and challenges of posit in general. This work has been carried out within the H2020 European Processor Initiative (EPI). The obtained results provide interesting feedback to the EPI CPU designers, since they can evaluate upon a time the impact of their design choices.

In Sect. 6, we present the results obtained on the official ARM Instruction Emulator, trying to point out the difference in terms of processing time between the different versions and vectorization levels. Finally, in Sect. 7 we present the results achieved using the tinyDNN library (equipped with the cppPosit library) on very deep convolutional neural networks (using synthetic images). These benchmarks are interesting for real-time image processing applied to the automotive scenario. Single-operation benchmarks show the impact of our approach on the image processing building blocks (e.g. convolution is a very common operation in image processing and filtering). Furthermore, the tinyDNN benchmarks are focused on widely used neural networks in the autonomous driving world. In particular, the evaluated networks are employed as basic blocks both in automotive computer vision (e.g. object detection and semantic segmentation).

## 2 Posit arithmetic

As widely shown in [7, 8, 10, 17, 18], the posit format is a fixed-length alternative representation to float numbers. A posit can be configured in the total number of bits (*nbits*) and the number of exponent bits (*es*). It has up to four fields as in Fig. 1:

- Sign field (1 bit, ). Posits are 2's complement.
- Regime field (variable length: it is identified as the sequence of identical bits $r$ followed by the opposite bit $\bar{r}$, ).
- Exponent field (maximum length of *es*, ). This field can be shorter or even missing at all, for some representations, even when $es > 0$.
- Fraction field (variable length, ) can be missing too.

Given such a format, the value $x$ is represented by the signed integer $v$ representing the posit:

$$x = \begin{cases} 0, & \text{if } v = 0 \\ \text{NaN}, & \text{if } v = -2^{(nbits-1)} \\ sign(v) \times useed^k \cdot 2^e \cdot (1+f), & \text{otherwise} \end{cases} \quad (1)$$

where $useed = 2^{2^{es}}$ and $f = \phi \cdot 2^{-F}$ is the fractional part represented by the fraction field.

Figure 2 shows an example of posit format decoding:

### 2.1 Fast approximated operations on posits using only the ALU

In this subsection, we will refer to $x$ as the represented real value and to $v$, $Y$ as the integers representing the posit, with $v$ being the input of the operation and $Y$ being the output of the same operation. The represented real value can be obtained from its representation using Eq. (1).


**Fig. 1** Illustration of a posit⟨16, 2⟩

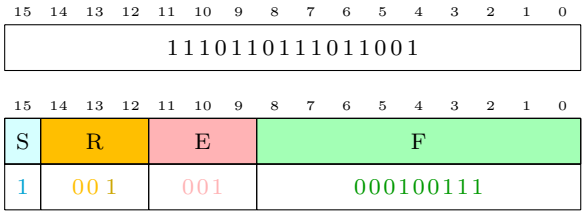



**Fig. 2** An example of a 16-bit posit with 3 bits for the exponent ($es = 3$). Given the sequence on top of the figure, after detecting it starts with 1, we have to compute the 2's complement of all the remaining bits (passing from 110-110-111011001 to 001-001-000100111). Then, we can proceed to decode the posit. The associated real value is therefore $-256^{-2} \cdot 2^{1} \cdot (1 + 39/512)$.. The final value is therefore $-1/65536 \cdot (1 + 39/512) = -0.00003284$

Please notice how, when $es = 0$, the formula in Eq. (1) can be further simplified as:

$$x = 2^k \cdot (1 + f) \tag{2}$$

where $k = -R$ for $x < 1$ and $k = R - 1$ for $x > 1$, and R is the regime length, as shown in Fig. 1.

This posit formulation allows implementing some arithmetic operators that, unlike IEEE 754 float numbers, can be evaluated by using only the ALU on the integer $v$ representing the posit bit string.

#### 2.1.1 The twice operation (2*x*)

When applying the twice operator, we consider three different cases for the posit value: $x \in [2, +\infty)$, $x \in [1, 2)$, $x \in [0, 1]$ (the same holds for negative values). We implement the twice operator in the different cases as follows (in the following expressions, $x \ll n$ represents the number $x$ left shifted of $n$ bits, $x \gg n$ represents the number $x$ arithmetically right shifted of $n$ bits, $x \gg^* n$ represents the number $x$ logically right shifted of $n$ bits, $|$ is the bitwise or operation and $\oplus$ is the bitwise exclusive or (xor) operation).

$$\begin{aligned} vabs &= abs(v) \\ s &= sign(v) \\ vs &= v \ll 1 \\ Y_t &= \begin{cases} vs \gg 1, & \text{if } x \geq 2 \\ vs \oplus twicemask, & \text{if } x \geq 1 \\ vs \ll 1, & \text{if } x < 1 \end{cases} \end{aligned}$$

where *twicemask* is obtained as follows:

$$twicemask = (1 \ll nbits - 2) | (1 \ll nbits - 3) \tag{3}$$

We obtain the final result as:

$$Y = (Y_t \gg^* 1) \oplus s - s.$$

A similar approach can be applied to the half ($x/2$) operation. We only need to change the transformation in the three different cases:

$$Y_t = \begin{cases} vs \ll 1, & \text{if } x \geq 2 \\ vs \oplus twicemask, & \text{if } x \geq 1 \\ vs \gg 1, & \text{if } x < 1 \end{cases}.$$

#### 2.1.2 The one's complement operator (1 − *x*)

The one's complement operator requires also that the posit to be in the range [0, 1] and can be implemented as follows:

$$Y = (1 \ll nbits - 2) - v.$$

#### 2.1.3 Fast reciprocate function (1/*x*)

We can implement a fast and approximated version of the reciprocate function as follows (where $\neg$ is a bitwise negation, $\oplus$ is the exclusive-or operator and *signmask* is a bit mask for the sign bit:

$$Y = (v \oplus \neg signmask).$$

The signmask can be obtained as follows:

$$\begin{aligned} msb &= 1 << (nbits - 1) \\ signmask &= ((msb | msb - 1) >> 1). \end{aligned}$$

Moreover, some interesting nonlinear activation functions in DNNs can be approximated with this format. Some of the most important approximated functions that can be implemented are the Sigmoid (see [7]), hyperbolic tangent and the extended linear unit function (see [9]), as also explained in Sect. 2.1.6.

#### 2.1.4 Fast sigmoid activation function

The sigmoid function $sigmoid(x) = 1/(1 + exp(-x))$ can be approximated as follows (where $v$ is the integer representing the posit and Y the integer representing the posit that approximates $sigmoid(x)$):

$$Y = ((1 \ll nbits - 1) + v + 2) \gg 2.$$

The approximated sigmoid function can be used as a building block for the other two functions, using linear combinations that exploit fast approximated operators of posit arithmetic seen before.

#### 2.1.5 Fast hyperbolic tangent

The hyperbolic tangent can be obtained from a linear combination of the sigmoid function using the double and the one's complement operators:

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

Finally, instead of using the exact sigmoid formula, we approximate the hyperbolic tangent using the fast approximated version of the sigmoid described in Sect. 2.1.4. In order to satisfy the one's complement requirements ($x \in [0, 1]$), we only consider negative $x$ values that result in the sigmoid output to be in [0, 1/2]. Then, we can exploit the hyperbolic tangent odd symmetry around 0 to obtain the values for the positive arguments.

#### 2.1.6 Fast extended linear unit

Similarly, the extended linear unit function for negative arguments can be implemented as a linear combination of the sigmoid using the said operators:

$$e^x - 1 = -2 \cdot \left[1 - \frac{1}{2 \cdot \text{sigmoid}(-x)}\right].$$

As in the previous case, if we substitute the exact sigmoid function with its fast approximated, we obtain the fast approximated version of the ELU. Figure 3 shows the accuracy comparison between the two. Figure 4 shows processing time comparison between latter two approximated functions. More mathematical details can be found in [17]. In the next section, we will see how to speed-up DNN training and inference using the SVE feature of modern ARM CPUs.

## 3 Posits and DNNs

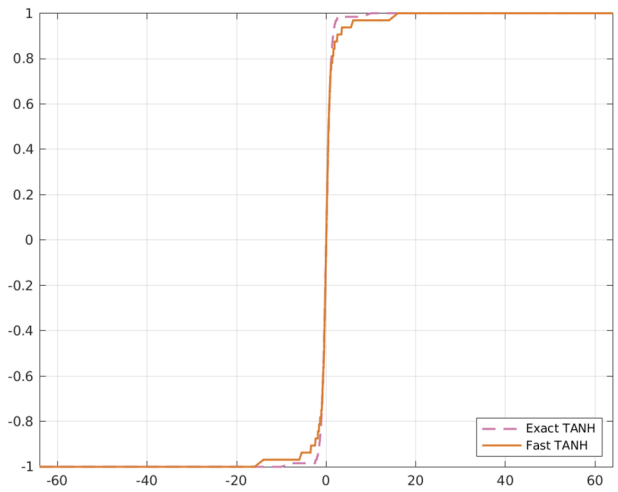
When considering posit numbers for DNNs, we need to take into account that the highest density of posit numbers is in the range $[-1, 1]$. This range indeed represents half of the posit projective circle. This can be exploited to design networks that are more proficient when used together with posit numbers. This can be addressed in different ways, as discussed in next subsections.

### 3.1 Activation functions

When choosing activation functions, we need to consider the output range of the functions. For example, if we consider the ReLU activation function, it discards all the negative numbers passed as argument flattening them to 0. Furthermore, the sigmoid function, limiting the output in [0, 1], discards the precious high-density region $[-1, 0]$. Instead, the hyperbolic tangent can fully exploit the region $[-1, 1]$. However, being modern deep neural network architectures very deep (the number of layers is huge), S-shaped functions like hyperbolic tangent suffer from vanishing gradients; thus, they are not acceptable in the training process. The ELU function and in general scaled extended linear units (SELUs [19]) manage to cover a higher range, typically parameterized by two real factors $\alpha$ and $\beta : [-\alpha \cdot \beta, +\infty]$.

### 3.2 Distribution of values

When stacking layers in a deep model, we need to care about the right-shifting of value distributions during forwarding passes. Adding a batch normalization layer [20] after some



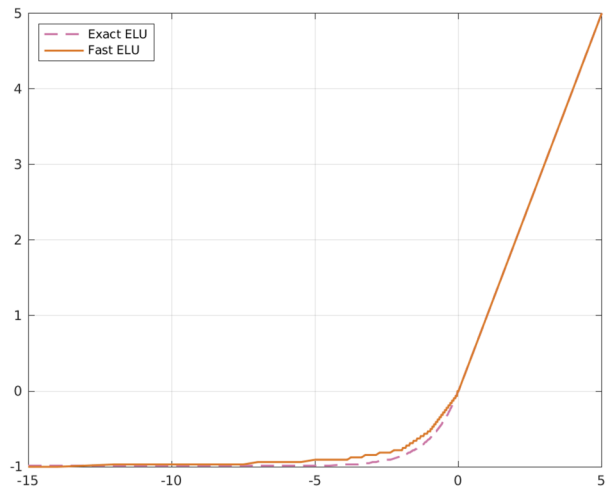
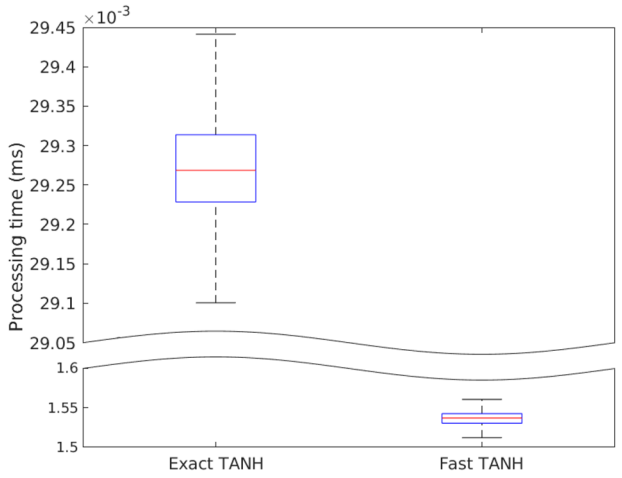




**Fig. 3** Accuracy comparison between exact (original formula applied to the posit format) and approximated versions of the hyperbolic tangent (TANH) and extended linear unit (ELU) using posit$\langle 8, 0\rangle$. The functions were computed on each point of the posit$\langle 8, 0\rangle$ range. The mean squared error between the TANH function versions is $2.8 \cdot 10^{-3}$, while the mean squared error for the ELU ones is $3.7 \cdot 10^{-3}$



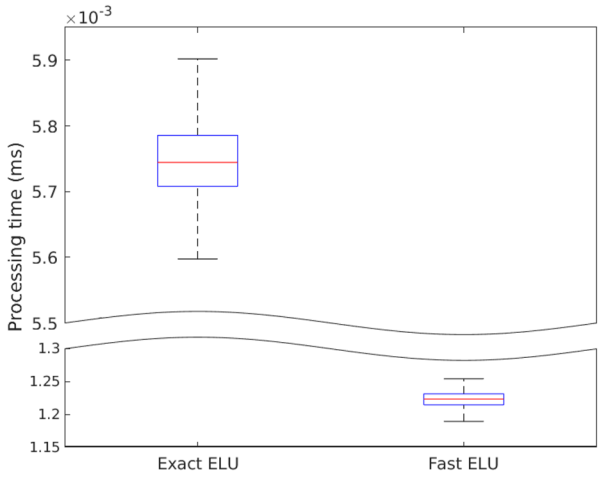




**Fig. 4** Processing time comparison between exact (original formula applied to the posit format) and fast approximated versions of the hyperbolic tangent (TANH) and extended linear unit (ELU). The reported results came from evaluations of the functions on each point of the posit$\langle 8, 0\rangle$ domain. As reported, the approximated ELU function is on average five times faster when compared to the exact version. On the other hand, the TANH function is more than 18 times faster on average

convolution and activation steps can manage to re-scale the values by subtracting the batch mean and dividing it by its standard deviation. This will result in a value distribution with a null mean value and unitary standard deviation, thus fitting the needs already explained.

### 3.3 Loss strategies

If we want to perform low-precision inference without losing too much accuracy (e.g. switching to posit$\langle 8, 0\rangle$ for inference), we may need to take into account the dynamic range of such types (e.g. posit$\langle 8, 0\rangle$ has a range $[-64, 64]$). This means that, during the training, we must penalize high network weights. This can be addressed by using different types of regularization. In [21] are shown recent trends in regularization for neural networks. For example, a weight decay approach (see [22]) with a decay rate of $\lambda$ adds the following L2 regularization term to the loss:

$$R(w) = \lambda \cdot \frac{1}{2} \cdot |w|_2^2.$$

This has been proven to reduce overfit and training error in [23]. In general, avoiding overfitting can help in maintaining low weight values. Therefore, the use of other layers designed to help with a generalization like dropout layer [24] can be useful as well.

### 3.4 Data pre-processing

When considering low-precision inference, we also need to take into account the encoding of data fed to the neural network. For example, if we take an RGB dataset, we will find each pixel encoded in each channel as an integer in [0, 255]. If we feed this type of data to a posit$\langle 8, 0\rangle$ network, it will result in values above 64 to be clipped down to the maximum value. Moreover, we are not exploiting the negative axis. To address this problem, we may apply a re-scaling of the encoding before even training the network. Simply re-scaling the image in $[-1, 1]$ is not always a good solution, since it may result in an unacceptable loss of information. Another important point in the posit circle is the $useed = \pm 2^{2^{es}}$ point that is strictly connected to the dynamic range of a posit $\pm useed^{nbits-2}$. For example, re-scaling an image in the range $[-useed, useed]$ of posit$\langle 8, 0\rangle$ (thus having the pixel encoded in $[-2, 2]$) has been proven to be effective encoding during both training (with higher-precision types) and inference phase (with low-precision types). The formula to re-scale the value $p$ of each pixel is therefore:

$$n(p) = 2 \cdot useed \cdot \frac{p}{255} - useed.$$

In Appendix, we describe a MATLAB tool, helpful to support the user in choosing the best posit configuration, depending on the needs of the application at hand.

## 4 ARM SVE architecture

The ARM scalable vector extension (SVE [14]) is a vector extension for the ARM AArch64 architecture supported by the ARMv8 instruction set. The main difference between SVE and other single-instruction multiple-data (SIMD) engines (Intel AVX/SSE or ARM NEON) is that it does not specify any width for vector registers, but it provides some

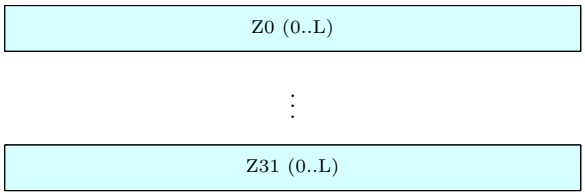



**Fig. 5** ARM SVE Z Data registers: 32 vector length agnostic register where $L = 128 \cdot k, k \in [1, 16]$

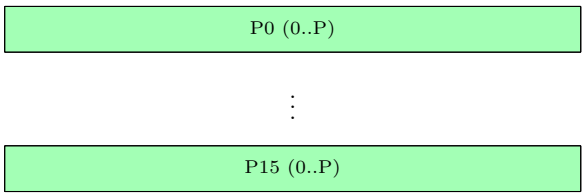



**Fig. 6** ARM SVE P predicate registers: 16 vector length agnostic register where $P = Z/8$

constraints for it. The vector register widths must be multiple of 128 up to 2048 bits. This approach, called vector length agnostic (VLA), allows us to implement only one vectorized version of our operations, exploiting both auto-vectorization and ARM ACLE (ARM C Language Extensions), without the need to target-specific hardware platforms (Figs. 5, 6).

SVE architecture introduces new kind of registers:

- *Z registers*: 32 registers with configurable width, from 128 to 2048 bits, as said above. These registers are meant to be data registers. SVE allows to interpret data in Z registers as 8 bits (bytes), 16 bits (half words), 32 bits (words) and 64 bits (double words). For instance, referring to posits, a 2048-bit Z register can hold up to 256 posit$\langle 8, X\rangle$ (in any exponent configuration).
- *P registers*: 15 predicate registers, with one bit to control each byte in a Z register (a 2048-bit Z register will be controlled by 256-bit P register). Each bit in the P register is interpreted as a Boolean. A predicate lane, made by 1 to 8 predicate bits, indicates whether the correspondent lane (when using a Z register) is active or not, depending on the least significant bit.

## 5 The C++ library cppPosit

For this work, we used the cppPosit C++ posit library developed in Pisa. This library exploits C++ templates to provide flexibility for posit configurations, ranging the total number of bits from 4 to 64. The main feature of cppPosit is the separation of the posit type in an interface frontend and a backend. The cppPosit frontend exposes all the possible implemented operations on posits, regardless of the underlying implementation. The backend implements the actual operations offered by the frontend, in one of the following flavours.

The supported backends are: i) fixed-point, ii) software floating-point (exploiting Berkeley SoftFloat library), iii) hardware floating-point unit (FPU) if present and iv) tabulated (or log-tabulated). The latter two deserves to be deeply analyzed. In fact, they become an important backend when a hardware posit processing unit (PPU) is not available and the number of bits is not required to be much large (like in DNNs).

### 5.1 Tabulated posits

When dealing with low-bit posits (e.g. 8-, 10-, 12-bit posits), we can think of pre-computing the arithmetic operators and some convenient functions in look-up tables to be used at run time for posit processing. Without optimization, these tables grow quadratically with the size of the posits. The main optimizations are applied exploiting addition and subtraction symmetry and antisymmetry properties to have half of the tables. The log-tabulated approach also optimizes the multiplication and division operations, by noticing that $\log(a \cdot b) = \log(a) + \log(b)$ (see also [4] for logarithmic numbers). In this way, we only need two single-operand tables for logarithm and exponentiation to perform both multiplication and division operations. These single-operand tables scale only linearly with the posit size, thus reducing the overall size of look-up tables. Note that the log-tabulated approach may result in some products or powers being off in the last bit.

### 5.2 Operational levels

The cppPosit library also classifies the posit operations into four different operational levels:

- L1: These operations only require bit manipulations of the signed integer $v$ representing the posit and thus can be executed just with the ALU support in a fast and efficient way. L1 operations are the most efficient one and are of crucial importance. Table 1 shows some of the L1 operations implemented in the library.
- L2: These operations require to decode the posit into the sign, regime, exponent and fraction with an additional unpacking step that slows down the computation. (This includes the use of count-leading-zeros (CLZ) operations.)
- L3: These operations also require the complete construction of the posit field, including the join between exponent and regime fields, with additional computation cost. Note that in the 0-bit exponent case L3 and L2 operations have the same complexity.

**Table 1** cppPosit most important implemented L1 operations, including common use activation functions such as sigmoid, hyperbolic tangent and extended linear unit

| Operation | Approximation | Requirements |
|---|---|---|
| $2 \cdot x$ | No | $es = 0$ |
| $x/2$ | No | $es = 0$ |
| $1/x$ | Yes | None |
| $1 - x$ | No | $es = 0, x \in [-1, 1]$ |
| FastSigmoid | Yes | $es = 0$ |
| FastTanh | Yes | $es = 0$ |
| FastELU | Yes | $es = 0$ |

– L4: These operations require the posit to be fully unpacked and reconstructed in the chosen backend.

As reported in Table 1, most of the L1 operations require to have 0 exponent bits, due to the emerging properties of posits with this particular configuration, as already explained in Sect. 2. Moreover, other functions such as the 1's complement require the posit to be in the unitary range.

### 5.3 Vectorized extension

In this section, we introduce the vectorized extension of the cppPosit library, aimed to provide the vector version of the posit operations. Firstly, we need to take into account the differences between different operational levels. L1 and L2 operations are the easiest one to be vectorized; they only require bit manipulation of unsigned or signed integers plus additional encoding and decoding steps. Instead, L3 and L4 operations need to be brought back at the chosen backend, and then, in case of hardware floating point, we can use native SIMD vectorization if any.

In order to provide a more general and abstract interface to posit vectorized operations, the architecture has separate posit vector frontend and a specialized posit vector backend that, in the paper case, implements the vectorized operations using ARM ACLE for SVE (Fig. 7).

When implementing vectorized operations, we have a common template to follow:

– Prologue: We need to prepare the data to be fed to the SIMD engine. For posits and L1 operations, this means preparing a vector with the signed integer representing the posits. In the SVE case, this means loading into the $Z$ registers the posit holder type content (e.g. `int16_t` for posit$\langle 16, X\rangle$) using the `svld1(...)` intrinsic. For L3/4 operations, we need instead to unpack the posit to the underlying backend (fixed, floating or tabulated) and load the backend type into registers as well, performing full decoding of the posit type.

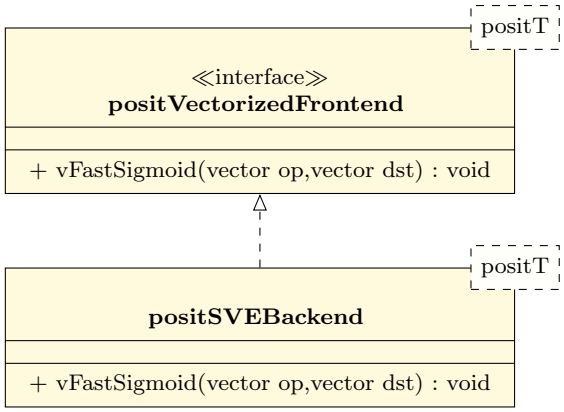



**Fig. 7** UML class diagram for an example SVEBackend that allows the vectorized computation of the fastSigmoid

– Body: The body contains all the arithmetic and logic functions needed to apply the considered operation. In the SVE case, this may contain the SVE intrinsics that operate on the $Z$ vector registers that contain the posit data. For instance, when implementing the fastSigmoid function, we will use the built-in intrinsics `svasr_x(...)` for the first right shift and `svadd_x(...)` for the sum. The first performs the same right shift on all the vector elements while the second performs the addition of the value $(1 \ll nbits - 2)$ to all the vector elements.
– Epilogue: We need to build back the posit into the result vector from the signed integer we have just manipulated in the function body. For SVE, this means invoking the `svst1(...)` intrinsic on the SVE result pointer obtained in the previous step. For L3/4 operations, we need instead to pack the posit up to the frontend, performing a full encoding of the posit type.

When vectorizing non-L1 operations that require the posit to be decoded in its components (sign, regime, exponent and fraction), we need to take into account two phases of the prologue. The first and simplest one is the posit conversion to the underlying signed integer holder type, that is, the cost of a pointer cast from the posit type to the holder one. This step has practically no cost. The second and hardest one is the vectorization of the posit decoding step since it involves many operations and branches on the bit string. After this decoding, the function body is the same as applying vectorization to the backend type (native ARM floats in our case).

The same behaviour holds for the epilogue as well. Predictably, both prologue and epilogue for non-L1 operations will introduce some kind of overhead in function computation, due to the conversion of the posit at the underlying backend. This means that, to see real effectiveness of this vectorized approach, we need to test this on large-sale data

and SVE vector sizes. This will be addressed more deeply in the next section..

## 6 Single-operation benchmarks results

In this section, we present benchmark results on isolated operations such as activation functions, dot products and DNN convolutions. All benchmarks are compiled in two different versions using the `armclang++ 19.3` compiler. One version is compiled enabling all compiler optimization using the `-Ofast` flag and targeting the `armv8-a+sve` architecture to enable SVE vectorized assembly instruction generation. The other version (*naive* from now on) is compiled without targeting any vectorization platform. In this way, we can compare the differences in execution time of single operations when using both the vectorized and the naive approach. At the time of writing, the only proposed hardware supporting SVE instruction set is the Fujitsu A64FX CPU [25] that employs 512-bit wide SVE vector registers. Unfortunately, it is not available to us at the moment. Therefore, both benchmarks presented in this paper are executed on the ARM SVE Instruction Emulator, with a different configuration of SVE vector lengths from 128 to 2048 bit. The emulator runs on a HiSilicon Hi1616 CPU with 32@2.4GHz ARM Cortex-A72 cores. (Only single-core performance is addressed for the single-operation benchmarks).

Table 2 shows activation function comparison between vectorized and naive approaches on different benchmarks. Each benchmark has been executed on 8192-bit vectors with different vectorization levels in the case of SVE. Each computation is repeated 1000 times, and the average is computed and reported. As we can see, every function benefits of a substantial speed-up thanks to vectorization, up to 18×.

Table 3 shows benchmark result for vectorized and naive approaches. Dot-product benchmarks have been executed on 8192-bit vectors with different vectorization levels in the case of SVE. Matrix multiplication benchmarks have been executed on $64 \times 64$ matrices. Each computation is repeated 1000 times, and the average is computed and reported. Convolution operations employ a $3 \times 3$ filter. More details are provided in the next subsections.

Note that the vectorized multiply and accumulate instruction (namely `svmla`) offered by the ARM SVE ISA allows to perform these operations as fused floating-point addition of products. As stated in the ARM SVE ACLE documentation [26], this instruction does not perform intermediate rounding step after the multiplication. (This is a very important behaviour.)

### 6.1 Dot product

As reported, vectorized dot product benefits of an impressive speed-up, even without vectorization of posit decoding. This can be straightforwardly explained: Both naive

**Table 2** Common activation function benchmark result comparison between vectorized and naive approaches

| posit version | vSig (ms) | | vTanh (ms) | | vELU (ms) | |
|---|---|---|---|---|---|---|
| | 8,0 | 16,0 | 8,0 | 16,0 | 8,0 | 16,0 |
| *Naive* | 3.41 | 3.08 | 5.76 | 7.24 | 8.12 | 8.54 |
| *SVE-128* | 0.59 | 1.51 | 1.32 | 2.65 | 1.29 | 2.60 |
| *SVE-256* | 0.73 | 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 |

*vSig* is the vectorized FastSigmoid. *vTanh* is the vectorized FastTanh. *vELU* is the vectorized ELU function

**Table 3** Common vector operation benchmark result comparison between vectorized and naive approaches

| Posit version | Dot (ms) | | vGeMM | | Conv | |
|---|---|---|---|---|---|---|
| | 8,0 | 16,0 | 8,0 | 16,0 | 8,0 | 16,0 |
| *Naive* | 14.60 | 14.95 | 3.7 | 3.23 | 80.67 | 80.84 |
| *SVE-128* | 3.20 | 3.11 | 0.58 | 1.12 | 24.02 | 37.99 |
| *SVE-256* | 3.31 | 3.21 | 0.72 | 1.04 | 11.66 | 21.49 |
| *SVE-512* | 2.42 | 2.28 | 0.41 | 0.61 | 6.85 | 14.03 |
| *SVE-1024* | 2.04 | 1.88 | 0.27 | 0.38 | 6.38 | 12.88 |
| *SVE-2048* | 1.82 | 1.62 | 0.20 | 0.26 | 3.65 | 8.81 |

Dot is the vectorized dot product. vGeMM is the vectorized general matrix–matrix multiplication. Conv is $3 \times 3$ convolution operation

and vectorized approaches need to convert posit to a chosen backend (native float in our case). Once converted, the vectorized approach fully exploits the floating-point unit and SIMD acceleration, while the naive one only exploits floating-point acceleration.

### 6.2 Matrix–matrix multiplication

Furthermore, matrix–matrix multiplication substantially benefits from vectorization operations. As a plus, as shown in [15], this operation has been realized avoiding the traditional sequence of dot products between rows and columns. The idea is to carry more than a single column of the second multiplication operand in vector registers, to be multiplied with the same element of the first one. Let $A \in \mathbb{R}^{M\times K}$ , $B \in \mathbb{R}^{K\times N}$ and $C = A \cdot B$ be the matrices involved in the operation. Values of $C$ can be obtained in batches of length equal to the vector register capability, say $L$, for the used representation as follows:

$$C_{i,[j:j+L-1]} = \sum_{i}^{K} A_{i,k} \cdot B_{k,[j:j+L-1]}. \tag{4}$$

### 6.3 Convolution

Computing the convolution (a sequence of matrix–vector multiplications) is the most demanding part in the forward pass of a convolutional deep neural network. Thus, speeding it up is crucial. Therefore, we considered a $3 \times 3$ convolution operation, where we obtained significant improvements from the vectorization approach, gaining a very impressive speed-up compared to the plain version. Our approach works for any size of the filter when the stride is equal to 1. For different types of convolution, the auto-vectorized version is preferable, still providing consistent speed-ups. The basic idea is to perform three different one-dimensional convolutions, one for each filter row, moving the filter along with the image matrix. For each filter stride, we convolve the filter rows with a batch of matrix row elements loaded in a vector register. In order to do this, we pre-fetch the nine filter elements in the vector registers. The pseudocode for the algorithm is shown in Algorithm 1. Note that the vector multiplication instructions are controlled by three different predicates, one for each column. The first predicate allows elements of the filter to be multiplied with elements inside the window $[j; j + L - 2]$, where $j$ is the current position of the filter in the image columns and $L$ is the SVE vector length. Similarly, the second predicate will allow multiplication only in $[j + 1; j + L - 1]$ and the third one in $[j + 2; j + L]$. The algorithm can be easily extended to $5 \times 5$ convolution, increasing the register pressure.

**Algorithm 1** 3×3 (stride = 1) Vectorized Convolution

**Input:** Image,Filter,Rows,Columns
**Output:** FeatureMap

```
/* Initialise by loading each of the 9 Filter weights in the SVE lanes */
// Loop on Image rows
for i = 0 to Rows − 2 do
    for j = 0 to Columns − 2 do
        /* Load the three rows targeted by current filter position (i,j), L is the SVE vector register length */
        firstRow = Image[i][j:j+L-1]
        secondRow = Image[i+1][j:j+L-1]
        thirdRow = Image[i+2][j:j+L-1]
        /* Perform multiplication with correspondent filter elements of same row. */
        p00 = firstRow × filterLane[0][0]
        ...
        p22 = thirdRow × filterLane[2][2]
        c0 = p00 + p10 + p20 // Reduce-sum columns
        c1 = p01 + p11 + p21
        c2 = p02 + p12 + p22
        res = c0 + c1 + c2      // Complete sum row-wise
        FeatureMap[i][:] = res  // Store the result row
    end for
end for
return FeatureMap
```

### 6.4 Pooling

Pooling kernels (i.e. average and max pooling) are important operations aimed to reduce the spatial information of network layers. In general, the spatial behaviour of these kernels is similar to convolution operations. In particular, the average pooling layer can be seen as a $f \times f$ convolution with all the filter elements fixed to $1/f^2$. Therefore, for average pooling we can make the same conclusion already applied to convolution operations, hence having similar results.

Typically, we want to reduce the layer output size by a factor $k$, equal to 2 or 3. To have a reduction of a factor 2, we need to employ a $2 \times 2$ kernel with stride equal to 2. In general, if we want to reduce the size by a factor $k$, we need to employ a $k \times k$ kernel with stride equal to $k$ (with appropriate padding). The implemented algorithm (see Algorithm 2) for vectorized max pooling is quite different from the convolution one, employing both intrinsic vectorization and auto-vectorization mechanisms. Consider now a typical $3 \times 3$ max-pooling with stride 3. The first step is to perform element-wise maximum between the three rows targeted by current filter top-left element row position $i$. Now we also need to perform the same operation column-wise. Therefore, to avoid expensive gather loads to fetch the matrix columns, we can perform the reduction with an additional, auto-vectorizable loop. We force the compiler to particularly vectorize a loop with an index step of 3 (that is, the separation step between groups of item involved in the maximum operation) using the following *pragma* directive:

**Table 4** Common pooling operation benchmark result comparison between vectorized and naive approaches

| Posit version | Max pooling (ms) | | Average pooling | |
|---|---|---|---|---|
| | 8,0 | 16,0 | 8,0 | 16,0 |
| *Naive* | 59.41 | 49.7 | 80.51 | 80.44 |
| *SVE-128* | 9.51 | 26.52 | 24.35 | 37.66 |
| *SVE-256* | 10.59 | 22.06 | 11.46 | 21.63 |
| *SVE-512* | 6.96 | 14.69 | 6.53 | 14.25 |
| *SVE-1024* | 5.12 | 11.84 | 6.68 | 12.48 |
| *SVE-2048* | 4.13 | 9.76 | 3.35 | 8.84 |

`#pragma clang loop interleave_count(3).`

This pre-processor directive aims to increase both the instruction-level parallelism inside the loop performing an unrolling operation and the data level parallelism of the vector performing data interleaving with a compiler-specified parameter 3.

For max pooling, we employed $225 \times 225$ images with a $3 \times 3$ pooling kernel with stride 3. Each computation was repeated 1000 times, and mean timing results are shown in Table 4. As reported therein, the max pooling operation incredibly benefits from the SVE vectorization, gaining a massive speed-up when compared to the naive version. As a plus, we analysed the Clang vectorization report to verify that the additional loop is interleaved by the compiler.

**Algorithm 2** 3×3 (stride = 3) Vectorized Max-Pooling

**Input:** Image
**Output:** PooledImage
1: */* Initialise by loading each of the 9 Filter weights in the SVE lanes*/*
2: *// Loop on Image rows*
3: **for** i = 0 to Rows-2 step 3 **do**
4: *Initialise maxRow element to store the maximum values in the SVE lanes*
5: **for** j = 0 to Columns - 2 **do**
6: */* Load the three rows targeted by current filter position (i,j), L is the SVE vector register length */*
7: firstRow = Image[i][j:j+L-1]
8: secondRow = Image[i+1][j:j+L-1]
9: thirdRow = Image[i+2][j:j+L-1]
10: */* Perform element-wise maximum between the 3 rows and store in maxRow. */*
11: maxRow = max(firstRow,secondRow,thirdRow)
12: **end for**
13: */* Auto-vectorized part: 3-by-3 maximum for the current maxRow */*
14: **for** k = 0 to Columns - 2 step 3 **do**
15: PooledImage[i/3][k/3] = max(maxRow[k:k+2])
16: **end for**
17: **end for**
18: **return** *PooledImage*

**Table 5** Image processing time (in seconds) for various very deep neural network models using posit⟨8, 0⟩

| Version | AlexNet | ResNet34 | VGG16 | VGG19 | ResNet152 |
|---|---|---|---|---|---|
| Naive | 40.06 | 146.07 | 590.68 | 675.32 | 779.7 |
| SVE128 | 2.76 | 10.07 | 40.74 | 46.57 | 53.77 |
| SVE256 | 2.64 | 9.61 | 38.88 | 44.45 | 51.32 |
| SVE512 | 2.54 | 8.93 | 36.12 | 41.30 | 47.68 |
| SVE1024 | 2.44 | 8.92 | 36.06 | 41.23 | 47.60 |
| SVE2048 | 2.34 | 8.90 | 35.97 | 41.13 | 47.48 |

For this benchmark random RGB $224 \times 224$, images are employed. As reported, the processing time with SVE vectorization experienced a dramatically improvement. Note that, however, 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 for evaluation at moment of writing and all benchmarks are executed on the ARM SVE instruction emulator, not on a real CPU

## 7 tinyDNN benchmarks results

In this section, we present benchmark results for the vectorized cppPosit library when used inside the tinyDNN neural network framework. For this benchmark, we used different very deep neural network models. In particular, we used some models proven to be successful in the ImageNet challenge [27]. AlexNet [23] consists of an eight-layer deep neural network, with internal convolutional layers reaching up to 192 convolution kernels, with an overall $60M$ parameters. ResNet architectures [28] are built stacking the so-called *residual blocks*, composed by two or, in deeper models, three convolutional layers. For each of them, the block input is summed to the output. This approach has been proven to improve classification performance in the ImageNet challenge. We tested our vectorized method on the 34-layer and 152-layer versions of the ResNet architecture. VGG16 and VVG19 models [29] are deep convolutional neural networks with a series of stacked convolutional layers (16 and 19 weight layers, respectively), where dimensionality reduction is operated by max-pooling layers.

Table 5 shows the inference results of the said benchmark networks. As reported the speed-up gained by the SVE-enabled version to the non-vectorized, one is impressive.

## 8 Conclusions

Since many of the current image processing applications use deep neural networks, it is crucial to speed-up at least the DNN forward-pass phase. In this paper, we presented an approach based on the use of a novel representation of real numbers (the posit format) and the speed-up of DNN operations using SIMD instructions. Our approach is interesting

for image processing applications where the GPU is not available, such as in most smart cameras applications or even in some assisted driving applications. More precisely, in this work we have presented a vectorized extension of posit arithmetic targeting the ARM SVE architecture, implementing some interesting core functions of machine learning and deep neural networks, where we have taken advantage of both explicit and auto-vectorization. We extended our cppPosit C++ software posit arithmetic library exploiting the knowledge on L1 operations and applying vectorization to the integer arithmetic behind them. This allowed us to obtain a substantial speed-up in the computation of fast approximated activation functions such as sigmoid, hyperbolic tangent and extended linear unit. Moreover, we proposed an approach for implementing machine learning vector and matrix operations with posit format, exploiting the underlying native vectorization for ARM floats, gaining again a solid speed-up in the computation of operations such as dot products and convolutions. Finally, we applied acquired knowledge to the tinyDNN C++ deep neural network library for the low-precision inference phase with 8-bit posits, reporting a relevant improvement in the mean sample inference time when switching from non-optimized version to optimized one. Future work includes porting more portions of the tinyDNN library to the ARM SVE architectures, to further extend the presence of vectorization within it.

**Acknowledgements** 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).

## Appendix: The posit designer tool

When choosing the posit configuration, we need to take into account multiple factors, such as target dynamic range and target decimal precision. We developed a MATLAB tool to analyse different alternative representations for real numbers, and we provide posit configurations that match the requirements for converting a given format into its closest posit alternative, evaluating range and resolution of them. The tool provides the following information:

1. Number-type statistics such as the total number of bits, maximum value and $\epsilon$ value (i.e. smallest step we can make from a number of that format). Figure 8 shows the output of this functionality. (In that figure, bin32_8 is a 32-bit float IEEE 754 with 8 bits for the exponent, i.e. a standard single-precision representation.)
2. Graphical evaluation of $\epsilon$ value against the max value (in a logarithmic scale).

| name | bits | maxvalue | epsvalue |
|---|---|---|---|
| 'bin32_8' | 32 | 3.4028e+38 | 1.1755e-38 |
| 'bin16_5' | 16 | 65504 | 6.1035e-05 |
| 'fixed16_8' | 16 | 128 | 0.0039062 |
| 'bin16_8' | 16 | 3.3895e+38 | 1.1755e-38 |
| 'posit8_0' | 8 | 128 | 0.015625 |
| 'posit16_1' | 16 | 1.0737e+09 | 3.7253e-09 |
| 'posit32_2' | 32 | 2.1268e+37 | 7.5232e-37 |

**Fig. 8** Posit designer tabulated statistics for different number types

| name | bits | maxvalue | epsvalue |
|---|---|---|---|
| 'posit16_3' | 16 | 1.3292e+36 | 1.9259e-34 |
| 'posit128_0' | 128 | 1.7014e+38 | 1.1755e-38 |
| 'posit121_0' | 121 | 1.3292e+36 | 1.5046e-36 |

**Fig. 9** Posit designer output for posit with 0 exponent bits covering the posit$\langle 16, 3\rangle$ configuration

**Table 6** Conversion table between training and inference types

| Training posits | Inference posits |
|---|---|
| posit$\langle 16, 1\rangle$ | posit$\langle 8, 2\rangle$ |
| posit$\langle 32, 2\rangle$ | posit$\langle 16, 3\rangle$ |
| posit$\langle 64, 3\rangle$ | posit$\langle 32, 4\rangle$ |

3. Next posit with 0 exponent bits that covers the dynamic range of a given number format. Figure 9 shows the output of this functionality.
4. posit to fixed type to build appropriate quire space for deferred rounding operations (such as exact multiply and accumulate).

Furthermore, we derived a general formula that allows us to convert any posit$\langle X, Y\rangle$ to any posit$\langle Z, W\rangle$ (with $X > Z$) without losing the dynamic range coverage:

$$W \geq \log_2\left(\frac{X-2}{Z-2}\right) + Y. \tag{5}$$

This may be useful when trying to reduce the number of bits during of neural network weights after we trained it. Table 6 shows an example of application of this formula.

## References

1. Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., Mansell, D.: Bfloat16 processing for neural networks. In: 2019

IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 88–91 (2019)
2. Köster, U., Webb, T., Wang, X., Nassar, M., Bansal, A.K., Constable, W., Elibol, O., Gray, S., Hall, S., Hornof, L., et al.: Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In: Advances in Neural Information Processing Systems, pp. 1742–1752 (2017)
3. Popescu, V., Nassar, M., Wang, X., Tumer, E., Webb, T.: Flexpoint: Predictive numerics for deep learning. In: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), pp. 1–4 (2018)
4. Johnson, J.: Rethinking floating point for deep learning. CoRR (2018). [Online]. Available: arxiv:1811.01721
5. Gustafson, J.L.: The End of Error: Unum Computing. Chapman and Hall/CRC, Boca Raton (2015)
6. Gustafson, J.L.: A radical approach to computation with real numbers. Supercomput. Front. Innov. **3**(2), 38–53 (2016)
7. Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: posit arithmetic. Supercomput. Front. Innov. **4**(2), 71–86 (2017)
8. Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S.: Novel arithmetics to accelerate machine learning classifiers in autonomous driving applications. In: 26th IEEE International Conference on Electronics Circuits and Systems (ICECS'19) (2019)
9. Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S.: A fast approximation of the hyperbolic tangent when using posit numbers and its application to deep neural networks. In: International Workshop on Applications in Electronics Pervading Industry, Environment and Society (ApplePies'19) (2019). [Online]. https://doi.org/10.1007/978-3-030-37277-4_25
10. Cococcioni, M., Ruffaldi, E., Saponara, S.: Exploiting posit arithmetic for deep neural networks in autonomous driving applications. IEEE Automotive (2018)
11. Carmichael, Z., Langroudi, H.F., Khazanov, C., Lillie, J., Gustafson, J.L., Kudithipudi, D.: Deep positron: A deep neural network using the posit number system. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1421–1426 (2019)
12. Fatemi Langroudi, S.H., Carmichael, Z., Gustafson, J., Kudithipudi, D.: PositNN framework: Tapered precision deep learning inference for the edge, pp. 53–59 (2019)
13. Lu, J., Fang, C., Xu, M., Lin, J., Wang, Z.: Evaluations on deep neural networks training using posit number system. IEEE Trans. Comput. 1 (2020)
14. A sneak peek into SVE and VLA programming (2018). https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/a-sneak-peek-into-sve-and-vla-programming
15. ARM Scalable Vector Extension and application to Machine Learning (2019). https://developer.arm.com/-/media/developer/products/software-tools/hpc/White%20papers/arm-scalable-vector-extensions-and-application-to-machine-learning.pdf?revision=510ee340-fce1-4fd8-bad6-bade674620a5
16. ARM NN (2019). https://developer.arm.com/ip-products/processors/machine-learning/arm-nn
17. Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S.: Fast approximations of activation functions in deep neural networks when using posit arithmetic. Sensors **20**(5) (2020)
18. Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S., de Dinechin, B.: Novel arithmetics in deep neural networks signal processing for autonomous driving: challenges and opportunities. IEEE Signal Process. Mag. (2020). https://doi.org/10.1109/MSP.2020.2988436
19. Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S.: Self-normalizing neural networks. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds) Advances in Neural Information Processing Systems 30, pp. 971–980. Curran Associates, Inc. (2017). [Online]. Available: http://papers.nips.cc/paper/6698-self-normalizing-neural-networks.pdf
20. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ser. ICML'15. JMLR.org, pp. 448–456 (2015). [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045167
21. Kukačka, J., Golkov, V., Cremers, D.: Regularization for deep learning: a taxonomy (2017)
22. Plaut, D.C., et al.: Experiments on learning by back propagation (1986)
23. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017). https://doi.org/10.1145/3065386
24. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014)
25. Post-K Supercomputer with Fujitsu's Original CPU, A64FX Powered by ARM ISA (2019). https://www.fujitsu.com/global/Images/post-k_supercomputer_with_fujitsu's_original_cpu_a64fx_powered_by_arm_isa.pdf
26. ARM C Language Extensions for SVE. https://static.docs.arm.com/100987/0000/acle_sve_100987_0000_00_en.pdf
27. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. (IJCV) **115**(3), 211–252 (2015)
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
29. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). arXiv preprint arXiv:1409.1556



**Marco Cococcioni** received the Laurea degree in 2000 and the diploma degree in 2001 in Computer Engineering from University of Pisa and Scuola Superiore S. Anna, respectively, both with magna cum laude. In 2004, he earned the PhD degree in Computer Engineering at the University of Pisa. In 2010–2011, he spent 2 years as Senior Visiting Scientist at the NATO Undersea Research Centre (now CMRE) in La Spezia, Italy. For his collaboration with CMRE, he obtained the NATO Scientific Achievement Award in 2014. Since 2016, he is an Associate Professor at the Department of Information Engineering of the University of Pisa. His main research interests include artificial intelligence, decision support systems, fuzzy logic, multi-objective evolutionary optimization, genetic fuzzy systems, pattern recognition, neural networks and operations research. He has been the general chair of three IEEE international conferences. In addition, he has co-organized six special sessions at international conferences and has been program committee member of 50+ international conferences. 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 Application. Prof. Cococcioni has co-authored 90 contributions to international journals and conferences, and he is a Senior Member of both IEEE and ACM (Association for Computing Machinery).

**Federico Rossi** is a PhD 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.

**Sergio Saponara** (SM'13) is Full Professor of Electronics at University of Pisa, where he got Master degree cum laude and PhD degree. In 2012, he was a Marie Curie Research Fellow in IMEC. He is an IEEE Distinguished Lecturer and co-founder of special interest group on IoT of both IEEE CAS and SP societies. He is the director of I-CAS Lab, of CrossLab Industrial IoT, of the Summer School Enabling Technologies for IoT. He is associate editor of several IEEE and Springer Journals. He co-authored more than 300 scientific publications and 18 patents. He is the leader of many funded projects by EU and by companies such as Intel, Magneti Marelli, Ericsson and PPC.