Small reals representations for Deep Learning at the edge: a comparison

Marco Cococcioni[1][0000−0002−7020−1524], Federico Rossi[1][0000−0002−4906−6997], Emanuele Ruffaldi[2][0000−0001−6084−6938], and Sergio Saponara[1][0000−0001−6724−4219]

[1] Università di Pisa [name].[surname]@unipi.it
[2] MMI s.p.a emanuele.ruffaldi@mmimicro.com

Abstract. The pervasiveness of deep neural networks (DNNs) in edge devices enforces new requirements on information representation. Low precision formats from 16 bits down to 1 or 2 bits have been proposed in the last years. In this paper we aim to illustrate a general view of the possible approaches of optimizing neural networks for DNNs at the edge. In particular we focused on these key points: i) limited non-volatile storage ii) limited volatile memory iii) limited computational power. Furthermore we explored the state-of-the-art of alternative representations for real numbers comparing their performance in recognition and detection tasks, in terms of accuracy and inference time. Finally we present our results using posits in several neural networks and datasets, showing the small accuracy degradation between 32-bit floats and 16-bit (or even 8-bit) posits, comparing the results also against the bfloat family.

Keywords: Deep learning · edge computing · fog computing · fine tuning at the edge · alternative representation for real numbers · small reals · posits · bfloat · weights compression

1 Introduction

Recently, it has been shown that Machine Learning in general, and Deep Neural Networks (DNNs) in particular, tolerate low-precision representations for their parameters. This constitutes an opportunity for speeding up the computations, to reduce storage, and, more importantly, to reduce power consumption. The latter is of paramount importance at the edge and on embedded devices.

However, to allow the porting of trained DNNs on difference devices, there is the need to standardize low precision formats for machine learning.

The aim of this work is to grab the attention to this very important topic, with the hope that sooner or later a standard, like the well-known IEEE 754 one (see [1]), will be put in place.

This is a necessity strongly felt by practitioners and industry, even if academics and researchers seem to be less aware of its importance.

To make the picture of the situation more complex, we should also take into account the requirements of safety critical applications, where low-precision is

less encouraged, but can still be considered, provided that it does not hamper the safety of the system.

Safety critical applications at the edge not only put more stringent requirements on the binary representation for small reals in DNNs, but can also add constraints of reproducibility of the computations. This latter aspect can impact the design of the representation. As an example, consider the use of stochastic rounding: even if it has been proved to increase the effectiveness during the training of a DNN (especially when using 8-bit precision floating point numbers [2]), it undermines the reproducibility of the computations. Since we are confident that sooner or later a standard will be created, it is important to start to make comparison between the existing alternative ways to represents real numbers in deep neural networks, in particular when planned to be used at the edge. Before doing this, we provide a review of all the techniques proposed so far to reduce power consumption, such as quantization, network pruning, etc.

The paper is organized as follows: in Section 2 we reviewed the state of art of deploying Deep Neural Networks at the edge and the main trends of research activities in this field. In Section 2.1 we briefly described the network pruning technique and its applications in simplifying neural networks. In Section 2.2 we summarized the network quantization approach, also covering networks working with binary or ternary weights (we have called the latter cases as "drastic quantization"). In Section 2.3 we reviewed a family of low-precision format for DNNs, called small reals, that include all the types we analysed later on. In Section 3 we analysed the most promising alternatives to IEEE 32-bit floats: bfloat family in Section 3.1, flexpoint in Section 3.2 and Logarithmic Numbers in Section 3.3. In Section 4 we presented and deeply analysed the posit^{TM} format, highlighting some important properties. Furthermore, we showed the main contributions of this work, consisting of the integration of the cppPosit library and bfloats inside some interesting machine learning frameworks. In Section 5 we presented results on deploying neural networks on a low-power constrained device, the Raspberry Pi 3B and in Section 6 we analyse the obtained results and their impact, other than discussing future developments of the proposed approach. Finally, in Section 7 we draw a few conclusions.

2 Deploying DNNs at the edge: state of the art

In the last decade, a lot of research efforts in DNNs has been devoted to reduce the resources required to exploit neural networks with limited memory, storage or computing power (such as smartphones or network edge devices), as demonstrated by the success of TensorFlow Lite, the low-precision counterpart of Google TensorFlow library. Two research lines emerged, the first one focusing on the inference phase only, leading to reduced-precision representation for the neural network parameters, the second one aimed at additionally speeding up the training phase using low-precision numerical formats also for the gradients.

Concerning low-precision numerical formats currently used in DNNs, three main approaches can be distinguished:

1. use of low-precision floating-point formats;
2. use of low-precision fixed-point real numbers or integer numbers;
3. use of binary/ternary formats.

These alternative representations can be limited to the weights, or to the weights and activations, or include all involved quantities (weights, activations and gradients). When following the first approach (i.e., low-precision floats), research and development efforts are converging toward a 16-bit floating point representation instead of the classical 32-bit one (which is called binary32) [1]. The same IEEE 754 standard [1] which has standardized binary32 has also standardized a 16-bit counterpart, called binary16, which reserves 5 bits to the exponent. However, most of the general purpose CPUs do not have full hardware support for binary16. In addition, it seems to be not particularly effective in deep learning. For these reasons, IBM has proposed a 16-bit floating point format having 6 bit for the exponent [3], called DLFloat (which stands for "deep learning float"), while Google has proposed the 8-bit alternative for the exponent [4], called bfloat16. This gap in the standard might be resolved soon, as there is a strong push from the machine learning community for suitable arithmetic formats. Another shortcoming of this approach is the lack of hardware support: as said above, most CPUs support 32- and 64-bit floats, but not 16- or 8-bit floats. Moreover, there are proposals to use a completely different representation for real numbers, like the posit format introduced in 2017 [5]. Although the posit format is promising for low-precision DNNs [6, 7, 8, 9], the not widespread availability of hardware support on CPUs still limits a large scale adoption (a list of hardware implementations of posits can be found at `https://en.wikipedia.org/wiki/Unum_(number_format)`).

The second approach (i.e., low-precision fixed-point numbers or integer numbers) is popular since it allows running DNNs even on entry-level CPUs microcontrollers not equipped with a Floating Point Unit (FPU), since the Arithmetic Logic Unit (ALU) is enough. On the one hand, fixed-point representations for real numbers are widely used (especially in financial applications and to improve the graphics in video games) even though C++ has not yet a standard library supporting them. DNN implementations using low-precision fixed-point for the both the weights and the activations are appearing [10]. Recently, a few papers discussed the specific issues of training DNNs with a fixed-point representation [11]. On the other hand, low-precision integer numbers are very interesting for time-sensitive applications, because operations between integer numbers have predictable computing times. However, the use of (low-precision integers), like 8-bit or less, usually requires a tailored training algorithm [12]. This approach is called *quantization*, for its obvious meaning.

The third approach takes the use of low-precision integer numbers in DNNs to the extreme, using ternary or even binary weights. Remarkable results have been obtained: DNNs with ternary weights (i.e., -1, 0 and 1) have been demonstrated to achieve the same classification accuracy as DNNs using binary32 weights [13]. DNNs with binary weights have been also devised, again with little or no degradation in the classification accuracy [14]. These results were confirmed on the

very challenging ImageNet dataset, considered as the most demanding open-source dataset for visual object recognition, with more than 20,000 different object categories [15]. The use of models with precision down to INT2 (i.e., 2-bit integer) has been demonstrated with a more than tolerable accuracy loss [16, 17]. As a result, NVIDIA has added the support down to binary numbers to its top-level GPUs to perform tensor operations [18]. Quantization can be applied either during the training phase or after it, just to perform the inference. However, DNN training using these numerical formats is more difficult compared to the two previously presented solutions as the gradient descent cannot be exploited, requiring the implementation of ad-hoc learning algorithms.

In [19] a series of challenges for DNN edge computing was presented. In particular the authors pointed out 4 main challenges to obtain a so called "TinyML": i) profiling the energy consumption is critical and the power consumption can vary wildly between different devices ii) edge systems often have very limited memory, two orders of magnitude smaller than usual smartphones, so optimizations are required iii) edge devices can be very different from each other, thus there is the need to normalize the benchmarks and the results obtained in those heterogeneous environments. iv) there is the need for software abstraction, even if this means losing a bit of low-level optimization that comes from hand-written and hand-tuned code.

2.1 Network pruning

When deploying trained model to edge devices we must balance the model accuracy performance with the inference processing time and resource utilisation. Indeed, the principal aim of network pruning is to reduce the computational cost of DNNs.

Typically DNNs are deployed with a large number of layers if several types, with most of them having their own weights and feature maps: traditionally, pruning is aimed to drastically reduce the amount of parameters in the network by removing some "redundant" connection between the layers. The idea is to delete such parameters whose removal will impact the less the training error. For example, we can delete very small-magnitude weights (when compared to the rest of the network). After deletion the network can continue its training, and so on, deleting weights at each step applying different deletion strategy. As a drawback the training process is significantly slowed down, since it requires a particular fine-tuning after each pruning step. The core idea expressed in [20] is to express an *optimal brain damage*, that is a theoretical measure of the *saliency* of weights in a network. In particular, a model of the training error functions is built and it is analytically associated with the effect of a perturbation of the parameters. From this expression the authors can express, with a series of transformations, the *saliency* s_k for each parameter k in the network.

The paper iterative approach is explained hereafter:

1. The neural network is first trained until a result (good enough) is reached
2. Each parameter is associated with a *saliency*, s_k

3. The parameters are sorted according to s_k and low-saliency parameters are deleted. Go to (1).

Deleting a parameter means setting it to 0 and making it immutable from that moment on.

2.2 Network quantization

During the years, as the deep neural networks models grew in accuracy over the most famous datasets (e.g. ImageNet and others) the network complexity in terms of Floating Point operations (FLOPs) and model footprint increased. In particular the footprint of network models (e.g. AlexNet model size is around 233 MB) is particularly critical in low-power and edge devices that can be particularly constrained in non-volatile storage capacity. Typically quantization involves the compression of weights using small integers, like 8-bit integer types.

In [21] the authors presented an overview of quantization techniques on deep neural networks. In particular the authors were able to compress complex networks like AlexNet by a factor 35, using a combination of quantization and weight sharing, while inducing a very minimal increase in the recognition error.

Drastic quantization (binary and ternary networks) Another approach to quantization is pushing the compression further, aiming to binary or ternary weights representations. In [22] the authors presented an overview of several approaches to drastic quantization, using the Hybrid Binary Neural Network (HBN) model. This model is based on a combination of 32-bit integer layers and binary layers. Typically, the input layer and the prediction output layer have 32-bit integer weights, while the intermediate ones are implemented using binary weights. We report an example of HBN from Quantized Keras (QKeras), where the 95% of weights are binary:

1. 2-dimensional convolution with 32-bit weights
2. Batch Normalization with 32-bit weights
3. Quantization layer
4. 2-dimensional convolution with binary weights
5. Batch Normalization with 32-bit parameters
6. Quantization layer
7. Fully Connected layer with binary weights
8. Output layer with 32-bit predictions.

The authors showed how the choice of layers to be quantised (binarised in this case) is critical to reduce the network footprint and complexity without impacting the accuracy of the model.

2.3 Small reals

Since quantization employs vary small integers for numerical representation, we lose the possibility to fine-tune our models on the edge without changing any aspect of the training algorithm. The idea of using *small reals* is based on the need for continuity between the original network model representation and the edge one. In particular, we want to remain in the real number domain. There are several formats that can be classified as small reals, each of them having at most 16-bit representation:

1. binary16: the standard IEEE 16-bit representation with 5-bit exponent and 10-bit fraction
2. bfloat family (in detail in Section 3.1) with 16 or 8 bit representations
3. posit numbers (in detail in Section 4) with 16 or 8 bit representations (but also intermediate variants can be used, such as 6, 10, 12 or 14 bits if we accept the cost of memory misalignment).

In the next section we provide the state of the art for alternative representations of real numbers, with special emphasis on small ones (16-bit or less). Then, in section 4, we review the posit format, which is considered at the moment the main challenger to the IEEE 754 format.

3 Alternative Real Number representation: state of the art

3.1 The bfloat family

The bfloat family is an alternative representation to IEEE 32-bit floating point numbers. In particular, the aim of bfloat is to propose a format that has very common characteristics with the IEEE 32-bit format, with a reduction on the format length.

bfloat16 The first format proposed in this family was the bfloat16. We summarize hereafter its structure:

- 1-bit sign
- 8-bit exponent
- 7-bit fraction

It substantially differs from its predecessor 16-bit IEEE Floating Point (binary16) because it has the same number of exponent bits as the 32-bit IEEE Floating Point (binary32). This allows a very fast conversion between the two types, since it only involves a truncation on the fraction (or an appropriate rounding, depending on the cases). This format can be employed both for low-precision inference and for mixed precision training [23].

There is a light support for bfloat16 in the latest generations of Intel Xeon CPUs; in particular BF16 instructions were added to the AVX2 vector extension of the architecture.

bfloat8 The bfloat8 format represents a further reduction in bits. Indeed, the format employs 5-bit exponent (as binary16) and only 2-bit for the fraction. This choice makes the conversion from binary16 to bfloat8 very fast, being it just a matter of truncation. The same cannot be said from binary32: in this case the conversion is more complex. A particular implementation of bfloat8 (in [24]) enabled the use of stochastic rounding during mixed precision training on this format. This approach allowed bringing in more randomization into the training phase. Let us consider a number represented on a float with a higher number of bits, let us say k bits, and we want to find its representation on k' bits, with $k' < k$. Let $x = s \cdot 2^e \cdot (1+f)$ (sign, exponent and fraction respectively) be such a number. As an example, x might be a binary32 and x' a bfloat8. We may compute the probability $p = \frac{f-f'}{\epsilon}$ where f' is the truncation of the larger fraction into the smaller one and $\epsilon = 2^{-k}$. With probability p we round x to $y = s \cdot 2^e \cdot (1+f'+\epsilon)$, while with probability $1-p$ we round it to $y = s \cdot 2^e \cdot (1+f')$. With this approach the authors were able to train several neural networks model on common datasets (e.g. CIFAR10 and ImageNet) with 8-bit floating point numbers: they reported very little degradation in DNN accuracy while reducing the model size by a factor ~ 2.

3.2 Flexpoint

The flexpoint format [25] is characterized by a *shared tensor exponent*. This exponent is used as a common exponent for all the reals in a given neural network layer or slice. This allows for example to have a 16-bit fixed-point representation in an entire DNN layer, with just additional 5-bits for the whole layer as an exponent. The exponent can be adjusted during the training, to match dynamic range variations that happen during the process. It should be noted that the idea behind flexpoint was already introduced earlier, but in different contexts, as "block floating point" representations (see, for example, [26]). Finally observe how the flexpoint approach, although interesting and powerful, cannot be used as a drop-in replacement to binary32: software changes are required to the DNN software libraries. This also makes cumbersome the reuse of pre-trained DNNs.

3.3 Logarithmic numbers

As reported in [27], the main problem with floating point operations in hardware is the transistors occupation for multiplication and division operations, which occupy the main part of the FPU, being significantly more complex than the circuitry for addition/subtraction. To address this issue, the Logarithmic Number System (LNS) was proposed decades ago in [28]. This system represents a number x a number as $y = 2^x$, in a pure logarithmic way. Following the logarithm properties this means that multiplication and division are just a matter of adding and subtracting logarithmic numbers (e.g. $y_1 \times y_2 = 2^{x_1+x_2}$).

However, this approach requires huge hardware lookup tables to compute the sum or difference of two logarithmic numbers [27]. This has been one of the main

bottlenecks for the format, since handling these tables can be more expensive than basic hardware multipliers.

Although this approach is really promising and can be combined with other formats, 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.

4 The posit format and innovative contributions of this work

Posit numbers [5] are a representation for real numbers that can be configured in two parameters, the number of bits *nbits* and the maximum number of exponent bits *esbits*.

The format can have at most 4 fields (3 when *esbit* is chosen equal to 0):

1. 1-bit sign
2. Variable length *regime*
3. Variable length (up to *esbits* if present) exponent[3]
4. Variable length fraction[4]

The novelty of the format is all in the regime field. This field is encoded with a run-length approach; indeed, its value depends on its length. To compute the length l of the regime we just need to look at the number of subsequent identical bits, interrupted by a bit of the opposite value (e.g. the bitstring `1110` has a length of 3, as well as the bitstring `0001`). To compute the actual value of the regime we need to take into account also the value of the single bit that is repeated in the sequence, let's call it b. The regime value is then:

$$k = \begin{cases} -l, \text{ if } b = 0 \\ l - 1, \text{ otherwise} \end{cases} \tag{1}$$

The strong point of the variable length format is inside the regime: when numbers are small (around the values ± 1), the regime length is low and the fraction length is high, thus giving the numbers in this area a high decimal accuracy. This makes perfectly sense when matched with Deep Neural Networks, where we can keep weights and activations across the layers near ± 1 exploiting weight decay and normalization techniques. Furthermore, if we look at the posit range, most of the values are in the range $[-1, 1]$; this means that, a neural network whose weights are entirely contained in this range will lose very little accuracy if represented using posit numbers [29].

Particular properties of the posit format emerge when configuring the format with 0 exponent bits. In detail:

[3] An different way to look at the exponent field is to consider it having a fixed length of *esbit*, where possible missing ones bits are implicitly considered equal to zero.

[4] The same consideration done for the exponent field also applies to the fraction, which could be regarded as a fixed-length field too, with implicit zeros.

1. In the range $[-1, 1]$ it is identical to a fixed point format
2. Simple operations such as doubling, halving and inverting can be computed without decoding, directly on the posit integer representation [29]
3. Several DNN activation functions can be computed decoding free (see next section)

4.1 Fast approximated activation functions

When we configure posit numbers with 0 exponent bits we can implement DNN activation functions using fast and approximated versions that can be computed directly on the integer representation, without decoding the posit.

Fast Approximated Sigmoid As pointed out in the original posit paper, the Sigmoid can be computed directly on the posit representation as follows (v is the integer representing the argument number, while Y is the integer representing the result number):

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

Fast Approximated Hyperbolic Tangent From the sigmoid function we can build other activation functions by manipulating the expression using the operations described in the previous section (doubling, halving, inverting and others). The hyperbolic tangent (tanh) can be implemented using the following expression (if substituting the sigmoid with its approximated version, we obtain the fast approximated tanh):

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

Fast approximated Extended Linear Unit The same approach can be followed with the Extended Linear Unit (ELU), by combining the fast approximated sigmoid function and the other approximated operations seen above:

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

In [30] the authors proposed a way to adopt posit numbers at the edge. They introduce a variant of posits called adaptive posit weight representation. When converting weights from 32-bit float representation they are also quantised to the adaptive posit format. This posit variant has a hyperparameter that control the dynamic range; it can be defined as a regime bias or as a maximum regime bit-width called rs. When $rs = 1$ the adaptive posit format is identical to a floating point with the same number of bits (the regime is non-existent in this case). When $rs = n - 1$ the adaptive posit format is a pure posit number. Thanks to this approach the authors were able to test their approach on different datasets and neural networks, without losing too much accuracy even with 5-bit adaptive posits. Furthermore they reported the maximum frequency (on

ASIC) obtained during conversions, peaking 1200 MHz with pure posit to float conversion with 5-bit posits. On the contrary, in this work we used standard 16-bit and 8-bit precision posits, and we have compared them with bfloat16 and bfloat8, respectively. The results of this comparison are reported in Section 5.

The aim of our approach is to compare different representations of real numbers on DNN fine tuning at the edge, *avoiding any change in the training algorithm*. In particular, we replace binary32 with bfloat16, bfloat8, posit16 and Posit8, and we report their classification accuracy on standard DNN classification benchmarks.

The added-value of this approach is that no software-hardware change are required, other than having an FPU supporting bfloat16/8 or posit16/8.

In particular, we are not requesting the support of the Stochastic Rounding, nor a different loss function or a tailored training algorithm.

In order to support posit numbers, in the past, we developed the cppPosit library [31]. To us, declaring a posit number is just simple as:

```
auto p8 = Posit<8,0,...>;
```

The greatest struggle in designing such a library was that we wanted a format that could be plug and play, so that we could just add it to any other machine learning library with just a type definition. To achieve this goal we focused on some core aspects of *modern* C++ (from 11 to 17):

- Type traits
- Extensive use of *constexpr* keyword to evaluate most of the branches at compile time, to gain as branchless portions of code as possible
- Extensive use of templates to generalize posit operations when compiling the code using `-Ofast -std=c++17`

When using novel types such as posit numbers, the lack of hardware is a critical aspect. We explicitly did not want to compute operations on posits (e.g. addition/multiplication and other) directly manipulating the posit bits. Instead, we only wrote the coding and decoding operations and the conversions to another type, called *backend*. The backend is a type that can leverage hardware acceleration to some extent. For example, two widely used backends in cppPosit are the fixed-point backend and the floating-point backend. Moreover, using a look-up table as a backend for such operations proved to be effective, but at greater memory cost.

Another obstacle to seamless integration of cppPosit with machine learning libraries was the interoperability with standard math library `<cmath>` or other linear algebra libraries (e.g. Eigen). Thanks to the extensive use of templates we easily integrated these two libraries within the cppPosit library, so that it could be easily used both with Eigen and the standard C++ math library.

This kind of interoperability out-of-the-box is not common in other posit libraries such as SoftPosit, that leverages a SoftFloat-like approach to arithmetic emulation. Furthermore, the cppPosit library is header only, therefore, its integration in a machine learning framework is simplified to just the inclusion of the main `posit.h` header.

Thanks to these design choices we integrated the posit library into the following machine learning framework:

- tinyDNN [32]: CPU-oriented DNN framework for small neural networks
- TensorFlow [33]: one of the most used DNN libraries, which offers a huge collection of datasets and pre-trained models.

A particular mention must be done to our posit-based TensorFlow implementation:

- The posit format was integrated *alongside* the other formats as a new *dtype* (a dtype is a data format in the TensorFlow name scheme)
- We needed to write a Python wrapper for cppPosit to accommodate the high-level Python interface.

As a result, we could load, store and convert pre-trained networks between posit format and the other format available in the TensorFlow library. In particular, we could manage to use 8-bit posits in TensorFlow (that typically does not allow 8-bit formats outside Tensorflow Lite) *without passing through network optimization and quantization* that are applied to the other 8-bit formats in TensorFlow. We achieved this by leveraging the posit encapsulation to mask the 8-bit type with a 16-bit memory alignment.

5 Comparison results

In this section we present some results on deploying neural networks on a constrained resource device. We used a Raspberry Pi 3B, equipped with a Cortex-A53 (ARMv8) CPU running at 1.4GHz and 1GByte of LPDDR2 SDRAM. We tested neural network models that were trained in a much more powerful system using 32-bit floating point numbers. Then we converted such models to different numerical formats to evaluate the accuracy degradation of such representation. Furthermore, we reported the sample inference time of the models on the Raspberry board.

We used the following network models:

- LeNet-5 like convolutional neural network ([34]),
- EfficientNet deep convolutional neural network ([35])
- Single Shot Detector (with 300×300 input images) ([36])

We used the following evaluation datasets:

- MNIST: hand-written digits recognition benchmark [37], 32×32 grey scale images
- GTSRB: German Traffic Sign Recognition benchmark [38], 32×32 RGB images
- CIFAR10: general purpose image recognition benchmark [39], 32×32 RGB images

- ImagenetV2: additional test-set that uses the same Imagenet classes but with new images [40], 224×224 RGB images
- Pascal VOC 2007: object detection dataset [41], 300×300 RGB images

In Table 1 we reported the accuracy results of the first three small datasets (MNIST, GTSRB, CIFAR10) with the LeNet-5 like neural network. Since we hand-trained on these three datasets, we were able to add a normalization on our data pipeline, in order to represent the images on the range $[-1, 1]$, enabling us to perform inference using low-bit posits and bfloat.

In Table 2 we reported the accuracy results of the big datasets (Imagenet, PASCAL VOC) with the very deep neural networks (EfficientNet and SSD300). Since we were using a pre-trained model, we could not control the image encoding; indeed, the images in these two model were expected to be encoded in $[0, 255]$. This prevented us to use 8-bit posits and 8-bit bfloat due to numerical ranges.

Table 1: Inference (test-set) accuracy on small, edge convolutional neural network trained with binary32, on different small datasets.

	LeNet-5 like CNN		
	MNIST	GTRSB	CIFAR10
binary32	98.86%	91.9%	83.5%
posit16,1	98.83%	91.8%	83.5%
posit16,0	98.50%	90.5%	83%
bfloat16	98.86%	91.9%	82%
posit8,0	98.34%	90.4%	78%
bfloat8	69.57%	80.45%	67.5%

Table 2: Inference accuracy test on very deep neural networks with big datasets (again, pre-trained using binary32).

	EfficientNetB0 + ImagenetV2	SSD300 + VOC 2007
	(accuracy)	(mean avg. precision)
binary32	81.9%	80.39%
posit16,2	79.7%	78.49%
bfloat16	78.9%	73.29%

Table 3: Sample inference time (frames per second in brackets) on different neural network models and input size. The times were evaluated on a Raspberry Pi3 Model B. Concerning posit16,x and posit8,x, we used $x = 0, 1, 2$ exponent bits, without observing changes in the speed.

	LeNet-5		EfficientNet	SSD300
Input Size:	$32 \times 32 \times 1$	$32 \times 32 \times 3$	$224 \times 224 \times 3$	$300 \times 300 \times 3$
posit16,x	9.2ms (108.5 fps)	23.9ms (41.72 fps)	17.05s (0.05 fps)	730s (0.0010 fps)
bfloat16	4.8ms (208.97 fps)	9.7ms (103.37 fps)	12.73s (0.08 fps)	472s (0.0020 fps)
posit8,x	9.1ms (110.38 fps)	21ms (46.94 fps)	15.89s (0,06 fps)	714s (0.0013 fps)
bfloat8	5.7ms (173.03 fps)	11ms (86.11 fps)	11.49s (0.09 fps)	528s (0.0018 fps)

6 Analysis of the Results and Future Developments

In Table 1 we can see how different formats perform in a scenario with small networks and simple datasets. As reported, all the 16-bit alternatives we analysed matched the baseline accuracy of the IEEE 32-bit floating point format. If we halve the bits again, with the 8-bit formats, we can see how 8-bit posits widely outperform bfloat8 numbers. This result show how 8-bit posits benefits from the non-fixed fraction bits, having the possibility to expand them at the expense of the regime when numbers are small. On the other hand, having only 2-bit of fraction in bfloat8 can be an issue when we plug directly the novel format without fine-tuning; indeed, if we could fine-tune the networks for a few epochs using only bfloat8 we could benefit from the stochastic gradient approach. However, without bfloat8 proper hardware support, this approach is still not feasible due to emulation overhead. We could think of applying the same idea to posit numbers, adding the support for such characteristic to a possible Posit Processing Unit (PPU).

From Table 2 we can see the behaviour of the 16-bit formats, when employed in more complex neural networks (EfficientNet has around 800 layers) and more challenging DNN tasks. As reported, the two 16-bit formats struggle to match the baseline accuracy, with the posit format losing 2 percentage point in both cases and the bfloat16 losing respectively, 3 and 7 percentage points.

In Table 3 we can see the sample inference time of the various networks, with different input sizes. When analysing these results we need to take in mind that we are completely emulating the behaviour of the different formats since we clearly do not have the proper hardware support and acceleration. Indeed, we rely on a floating point backend to perform the computation while weights and images are stored using the emulated format. This means that, for each multiplication or addition, we will convert the number to the floating point backend and then we will convert it back after computation to the original emulated format. This results show how bfloat family largely benefits from the strict similarity with IEEE floats; indeed, the conversion between a bfloat16 number and a binary32 one, is just a left shift of 16 positions (and vice versa) while the conversion

between a posit and a binary32 numbers is way more complex, involving more operations.

If we combine both results from the tables we can envision a scenario where we use a 16-bit bfloat16 to perform mixed precision inference/training on 16-bit while we stick to posit8 for low-precision inference, having a clear advantage over bfloat8 in our tests.

6.1 Future developments

When analysing bfloat8 we saw that it could benefit from a few epochs of fine-tuning using the stochastic rounding proposed by the authors. The most common framework that employs 8-bit formats, such as Tensorflow/Lite , widely use the quantization technique and network pruning to simplify networks for deployment in edge devices. This approach can introduce some issues: i) loss of performance in terms of accuracy ii) no guarantee of meeting target platform requirements iii) no guarantee on inference time or frames processed per second. An idea could be optimizing the network adding a multi-objective genetic algorithm that takes into account some parameters as constraints to match the target platform: i) maximum number of hidden layers, and ii) maximum number of active neurons per layer. With such constraints, we will be able to control both the time complexity for the training, the RAM request, and the inference latency (which, on his turn, impacts the frame per second that can be processed, in computer vision applications).

Future works may involve exploiting posit numbers for a family of microcontrollers that are equipped with an FPU (e.g. STM32 or Cortex F4) to be used as back-end unit for the computation.

7 Conclusions

In this paper we reviewed several techniques to optimize neural network for deployment to the edge. We have highlighted the quest for a new standard for computations with small reals at the edge. In particular we analysed the behaviour of two very promising formats, the bfloat family and the posit format. We presented some results concerning the use of the posit representation and compared them to results with bfloat numbers. From the results we saw that 16-bit posits and bfloat can match the baseline IEEE 32-bit float accuracy in several DNN task. Furthermore, we saw how 8-bit posit can outperform 8-bit bfloat in simple DNN tasks. Despite the good results obtained so far using posits, we think that there is still much to explore in order to fully exploit the potential of this novel format. In particular we expect to obtain more interesting results with the proper hardware support for both posit numbers and bfloat, which would allow the native training of really deep neural networks, or the fine tuning at the edge.

Acknowledgments

Work partially supported by H2020 projects (EPI grant no. 826647, https://www.european-processor-initiative.eu/ and TEXTAROSSA grant no. 956831, https://textarossa.eu/) and partially by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

References

[1] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[2] N. Mellempudi et al. *Mixed Precision Training With 8-bit Floating Point.* 2019. arXiv: 1905.12334 [cs.LG].

[3] A. Agrawal et al. "DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference". In: *2019 IEEE 26th Symp. on Computer Arithmetic (ARITH'19).* 2019, pp. 92–95. DOI: 10.1109/ARITH.2019.00023.

[4] N. Burgess et al. "Bfloat16 Processing for Neural Networks". In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH).* June 2019, pp. 88–91. DOI: 10.1109/ARITH.2019.00022.

[5] J. L. Gustafson and I. T. Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic". In: *Supercomputing Frontiers and Innovations* 4.2 (2017), pp. 71–86.

[6] M. Cococcioni, E. Ruffaldi, and S. Saponara. "Exploiting Posit arithmetic for Deep Neural Networks in Autonomous Driving Applications". In: *In Proc. of the 2018 IEEE Int. Conf. of Electrical and Electronic Technologies for Automotive (Automotive '18).* 2018, pp. 1–6. DOI: 10.23919/EETA.2018.8493233.

[7] M. Cococcioni et al. "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.* Ed. by Sergio Saponara and Alessandro De Gloria. Vol. 627, Lecture Notes in Electronic Engineering. Springer Int. Pub., 2020, pp. 213–221.

[8] M. Cococcioni et al. "Novel Arithmetics in Deep Neural Networks Signal Processing for Autonomous Driving: Challenges and Opportunities". In: *IEEE Signal Processing Magazine* 38.1 (2021), pp. 97–110. URL: 10.1109/MSP.2020.2988436.

[9] M. Cococcioni et al. "Fast deep neural networks for image processing using posits and ARM scalable vector extension". In: *Journal of Real-Time Image Processing* (2020), pp. 1–13. ISSN: 1861-8200. URL: 10.1007/s11554-020-00984-x.

[10] D. Lin, S. Talathi, and S. Annapureddy. "Fixed Point Quantization of Deep Convolutional Networks". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2849–2858.

[11] X. Chen et al. "FxpNet: Training deep convolutional neural network in fixed-point representation". In: *International Joint Conference on Neural Networks (IJCNN 2017)*. 2017.

[12] Shuchang Z et al. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. 2018. arXiv: 1606.06160 [cs.NE].

[13] H. Alemdar et al. "Ternary neural networks for resource-efficient AI applications". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 2547–2554. DOI: 10.1109/IJCNN.2017.7966166.

[14] Haotong Q. et al. "Binary neural networks: A survey". In: *Pattern Recognition* 105 (2020), p. 107281. ISSN: 0031-3203.

[15] O. Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[16] J. L. McKinstry et al. "Discovering low-precision networks close to full-precision networks for efficient embedded inference". In: *arXiv preprint arXiv:1809.04191* (2018).

[17] J. Su et al. "Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2018, pp. 29–42.

[18] J. Choquette et al. "NVIDIA A100 Tensor Core GPU: Performance and innovation". In: *IEEE Micro* 41.2 (2021), pp. 29–35.

[19] C. R. Banbury et al. "Benchmarking TinyML Systems: Challenges and Direction". In: *arXiv e-prints*, arXiv:2003.04821 (Mar. 2020), arXiv:2003.04821. arXiv: 2003.04821 [cs.PF].

[20] Y. Le Cun, J. S. Denker, and S. A. Solla. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 598–605. ISBN: 1558601007.

[21] S. Han, H. Mao, and W. J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV].

[22] D. Pau et al. "Comparing Industry Frameworks with Deeply Quantized Neural Networks on Microcontrollers". In: *2021 IEEE International Conference on Consumer Electronics (ICCE)*. 2021, pp. 1–6. DOI: 10.1109/ICCE50685.2021.9427638.

[23] D. Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. arXiv: 1905.12322 [cs.LG].

[24] N. Wang et al. "Training Deep Neural Networks with 8-Bit Floating Point Numbers". In: *Proceedings of the 32nd International Conference on Neu-*

ral Information Processing Systems. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 7686–7695.

[25] U. Köster et al. "Flexpoint: An adaptive numerical format for efficient training of deep neural networks". In: *In Proc. of teh 31st Conference on Neural Information Processing Systems (NIPS'17)*. 2017, pp. 1742–1752.

[26] A. Oppenheim. "Realization of digital filters using block-floating-point arithmetic". In: *IEEE Transactions on Audio and Electroacoustics* 18.2 (1970), pp. 130–136. DOI: 10.1109/TAU.1970.1162085.

[27] J. Johnson. "Rethinking floating point for deep learning". In: *CoRR* abs/1811.01721 (2018). arXiv: 1811.01721. URL: http://arxiv.org/abs/1811.01721.

[28] M. G. Arnold, J. Garcia, and M. J. Schulte. "The interval logarithmic number system". In: *In Proc. of the 16th IEEE Symp. on Computer Arithmetic (ARITH'03)*. 2003, pp. 253–261. DOI: 10.1109/ARITH.2003.1207686.

[29] M. Cococcioni et al. "Fast Approximations of Activation Functions in Deep Neural Networks when using Posit Arithmetic". In: *Sensors* 20.5 (2020). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/20/5/1515.

[30] H. F. Langroudi et al. "Adaptive Posit: Parameter Aware Numerical Format for Deep Learning Inference on the Edge". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2020.

[31] E. Ruffaldi. *cppPosit*. https://github.com/eruffaldi/cppPosit.

[32] E. Riba and P. Nyan. *tinyDNN*. https://github.com/tiny-dnn/tiny-dnn.

[33] *TensorFlow*. https://www.tensorflow.org/. 2009.

[34] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[35] M. Tan and Q. V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG].

[36] W. Liu et al. "SSD: Single Shot MultiBox Detector". In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

[37] Y. LeCun and C. Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[38] J. Stallkamp et al. "The German Traffic Sign Recognition Benchmark: A multi-class classification competition". In: *In Proc. of the IEEE Int. Joint Conf. on Neural Networks (IJCNN'11)*. 2011, pp. 1453–1460.

[39] A. Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[40] B. Recht et al. *Do ImageNet Classifiers Generalize to ImageNet?* 2019. arXiv: 1902.10811 [cs.CV].

[41] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results*. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.