# Low-precision logarithmic arithmetic for neural network accelerators

Maxime Christ, Florent de Dinechin, Frédéric Pétrot

# Low-precision logarithmic arithmetic for neural network accelerators

Maxime Christ†, Florent de Dinechin‡, Frédéric Pétrot†
†Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, France
maxime.christ@insa-lyon.fr florent.de-dinechin@insa-lyon.fr frederic.petrot@univ-grenoble-alpes.fr
‡Univ Lyon, INSA Lyon, Inria, CITI, France

***Abstract*—Resource requirements for hardware acceleration of neural networks inference is notoriously high, both in terms of computation and storage. One way to mitigate this issue is to quantize parameters and activations. This is usually done by scaling and centering the distributions of weights and activations, on a kernel per kernel basis, so that a low-precision binary integer representation can be used. This work studies low-precision logarithmic number system (LNS) as an efficient alternative.**

**Firstly, LNS has more dynamic than fixed-point for the same number of bits. Thus, when quantizing MNIST and CIFAR reference networks without retraining, the smallest format size achieving top-1 accuracy comparable to floating-point is 1 to 3 bits smaller with LNS than with fixed-point. In addition, it is shown that the zero bit of classical LNS is not needed in this context, and that the sign bit can be saved for activations.**

**Secondly, low-precision LNS enables efficient inference architectures where 1/ multiplications reduce to additions; 2/ the weighted inputs are converted to classical linear domain, but the tables needed for this conversion remain very small thanks to the low precision; and 3/ the conversion of the output activation back to LNS can be merged with an arbitrary activation function.**

**The proposed LNS neuron is detailed and its implementation on FPGA is shown to be smaller and faster than a fixed-point one for comparable accuracy.**

## I. Introduction

Commodity hardware such as multi-core processors and GPUs, although intrinsically very powerful, is not very power-efficient in neural network inference. This is why the last decade has seen a boom in the number of hardware accelerators for inference tasks, applicable to both cloud and embedded solutions [1]. These accelerators can be either computation engines directly placed into processor pipelines, such as ARM Scalable Vector Extensions [2] and Intel DL Boost [3], or independent coprocessors such as Google TPU [4], Qualcomm Cloud AI [5], or Kalray KaNN [6], to name a few. The chosen strategy is a trade-off between mapping tools complexity, target power efficiency, flexibility, ease of integration in existing frameworks, and so on.

These accelerators typically use 8-bit integers instead of 32-bit floating-point numbers for inference tasks. It has indeed been demonstrated that 8-bit values entail only very small inference accuracy loss for most application domains [7], hence the generalization of 8-bit quantization as a "one size fits all" solution. Furthermore, methods that efficiently map floating-point network parameters (weights and biases learned during training) onto the 256 available values do exist [8], and are already available in the most popular frameworks for Machine Learning (ML). The gain in memory bandwidth and computational effort offered by 8-bit quantization is huge.

Still, the search for higher power efficiency has lead researchers to study even smaller bit-widths, from both the algorithmic and architectural point of view. At the extreme, binary ($\{-1, 1\}$) and ternary ($\{-1, 0, 1\}$) networks have drawn some attention. Their neuron computations can be implemented as a handful of logic gates, and the memory requirements can hardly be smaller [9, 10]. Such networks can reach good accuracy [11, 12, 13], and research is still active in improving their training. However, to mitigate the accuracy loss due to such extreme quantizations, it is customary to increase the number of neurons and/or layers, at the cost of more parameters, hence a full retraining of the network.

This seems to leave some room for quantization lower than 8 bits, without changing the network structure, without retraining, and without accuracy loss.

However, for this to work, we need to drop fixed-point for a representation that offers a wider range for fewer bits. It the present study, we consider low-precision Logarithmic Number System (LNS) to that aim.

The paper is organized as follows. Section I gives some background on LNS with a focus on its former and current applications in ML. Using this representation and taking into account the constraints on the values induced by ML, Section III details the design and implementation of an "LNS neuron". Section IV explores the design space covered by this neuron, in particular the influence of its parameters on resource usage and top-1 accuracy. Finally Section V summarizes the results of this exploration, concluding that an LNS neuron is is more efficient than a fixed-point one for the same inference

TABLE I: Notations used in this article

| Symbol | Meaning |
|---|---|
| $b$ | the base of logarithms used, usually $b = 2$ |
| $X$, $W$, $B$, $P$ | real values of input, weight, bias and $WX$ product |
| ${}^{\mathrm{lg}}X$, ${}^{\mathrm{lg}}W$, ${}^{\mathrm{lg}}P$ | their LNS representation, e.g. ${}^{\mathrm{lg}}X = (z_X, s_X, L_X)$ |
| $L_X$, $L_W$, $L_P$ | signed fixed-point logarithms, $L_X \approx \log_b \lvert X \rvert$ |
| $s_X$ | a sign bit, $s_X = 1$ when $X < 0$ |
| $z_X$ | a "isZero" bit, $z_X = 1$ iff $X = 0$ |
| $m$ | Most Significant Bit (MSB) of the fixed-point logarithm |
| $\ell$ | Least Significant Bit (LSB) of the fixed-point logarithm |
| $\ell'$ | LSB of the fixed-point sum in linear domain |
| $N$ | number of terms to sum in the linear domain |

accuracy.

## II. Background and State of the Art

### A. Logarithmic Number Systems (LNS)

LNS is best viewed as an alternative to floating-point where the mantissa of a floating-point number is replaced with fractional bits of the exponent [14]. A positive real $X$ is represented by its logarithm $L_X \approx \log_b X$ in some base $b$ (usually $b = 2$ but other bases can be used [15]). $L_X$ is itself represented in signed fixed point. $X = 1$ is represented by $L_X = 0$, negative $L_X$ represent numbers $X < 1$, positive $L_X$ represent $X > 1$. To represent negative values of $X$, an LNS system adds a sign bit $s_X$. Finally, since $\log_b(0) = -\infty$, the value $X = 0$ needs a special encoding in ${}^{\lg}X$. The mainstream approach is to use a "isZero" bit $z_X$, but an encoding as a special value of $L_X$ can also be used. All these notations are summarized in Table I.

LNS is mathematically simpler and more elegant than floating-point. For base-2 LNS ($b = 2$), if $L_X$ has $i$ integer bits and $f$ fractional bits, the LNS system provides accuracy and dynamic range comparable to that of binary floating-point with $i$ bits of exponent and $f$ bits of significand fraction.

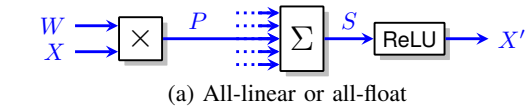
The main advantage of LNS arithmetic is that LNS multiplication of ${}^{\lg}X$ by ${}^{\lg}Y$ resumes to the fixed-point addition of $L_X$ and $L_Y$. Being a fixed-point addition, it is exact, without any rounding error (contrary to floating-point multiplication that entails some rounding). It can over/underflow, though.

The huge drawback of LNS is the cost of addition and subtraction in this system. Without detailing it, let us just state that the hardware complexity of an LNS adder scales poorly with the precision [14, 16]. This has prevented its adoption as a mainstream format for general-purpose computing. A common option is to use approximate LNS adder/subtracters [17, 18, 19, 20]. The alternative approach proposed in this work is to use very low precisions (less than 8 bits) for which accurate LNS arithmetic is cheap. Here of course we mean: accurate with respect to the format, not accurate in absolute terms since the format precision itself is low. Indeed, for such precisions it is possible to define arbitrary functions in extension, then trust the synthesis tools to optimize their implementation. Thus the elegance of the format is not nullified by the architectural complexity [17, 14, 21, 16, 22, 18] of implementing it in hardware.

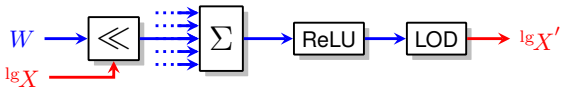
More specifically, this work targets FPGAs whose fine-grain structure is that of a 4- to 6-bits Look-Up Table (LUT), and the present work aims at soft-spot formats that efficiently match these architectural LUTs. For this, a contribution of this work is to depart from off-the-shelf LNS formats, shaving as many bits as possible to define a purely application-specific format for machine learning inference.

### B. LNS for machine learning

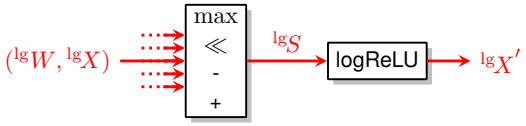
With the slide rule, logarithmic arithmetic predates electronic computers. It was studied for them very early [17, 23], with pioneering works [24, 21] demonstrating than 12-bit LNS could compete with floating-point in neural network training.



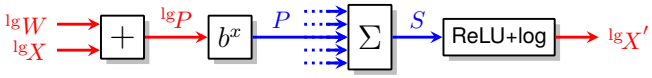

(a) All-linear or all-float




(b) Log activations, linear weights [18]




(c) Log activations, log weights, approx log sum [18]




(d) Proposed: log activations, log weights, accurate linear sum [25]

Fig. 1: Artificial neurons using log-encoded data (in red)

In the era of deep learning, the potential of low precision logarithmic encoding was again demonstrated [22] then put in practice [18] with an architecture (Figure 1b) that keeps the weights in the linear domain and implements multipliers as bit shifts. The final conversion to the logarithmic domain is approximated with a Leading One Detector (LOD). Another approach is to replace a standard multiplier with an approximate one based on LNS arithmetic [19, 20].

Figure 1 describes the various ways logarithmic encoding has been used in the recent literature. The approach we advocate (Figure 1d) is a simplified version of the "somewhat inaccurately named" (according to the author himself) Exact Log-Linear Multiply-Add (ELMA) approach [25]. The term "exact" emphasizes the fact that both the addition of the logarithms and the summation of the product terms are exact, since both are performed as fixed-point additions. However there are still rounding errors in the log/linear conversions (in the $b^x$ and ReLU+log boxes of Figure 1d).

### C. Originality and contributions

In the ELMA approach [25], a product ${}^{\lg}P$ may have integer and fractional bits: $L_P = I + F$ with $I$ a binary integer and $F \in [0, 1)$. In the present work, we make sure that 2 integer bits are enough (see Section III). In this case, we claim that with the wide LUTs of modern FPGAs, a plain tabulation of $b^x$ is more area-efficient than a table+shift approach: the shifter alone requires at least one FPGA LUT per output bit, just like the plain table. Plain tabulation also greatly improves the worst-case error, as in ELMA the $b^F$ rounding error may be amplified by the shift. Compared to previous work using just shifts [18] or a variant of Mitchell's approximation [17, 21, 18], plain tabulation even ensures correctly rounded logs and antilogs (i. e., as accurate as the format allows).

Finally, plain tabulation gives us the freedom to use the same architecture with $b \neq 2$, which has been shown to be beneficial [15], although the present study doesn't exploit this.

Similarly, in the trained data, the activation-weight products are consistently smaller than 1, therefore their logarithm is always negative, and the sign bit of logarithms can be saved: the fixed-point number stored in our approach for $A \in \{W, X, P\}$ is actually $-\log_b A$. We also show later how we can dispense of the zero bit (and the MUXes that manage it) by making sure that some of the log values will be rounded to 0 for the summation by the $\boxed{b^x}$ box.

There are other complications in ELMA, such as tapered encoding and a few more architecture parameters. ELMA is also essentially a multiply-and-add operator that has to be used iteratively in an inference architecture. In our work the operator is a much coarser neuron, inputting in parallel $N$ activation/weight pairs. Depending on the available resources and external memory bandwidth, the number of neurons and their number of inputs can be chosen so as to optimize the operational intensity [26] for a given FPGA. This allows supporting networks much larger than the considered FPGA can hold, at the price of external memory accesses and somewhat complex scheduling schemes [27]. The value of $N$ can also be chosen to directly match the needs of the various layers of small to medium neural networks and fits a mid-range FPGA — $N = 784$ allows for a fully parallel MNIST, $N = 1152$ allows for an efficient implementation of most layers in a reference CIFAR-100 implementation. This massive parallelism is available in current FPGAs (including the internal bandwidth needed to store intermediate activations and buffer the weights read from high-speed external DRAM), in particular when all values in use fit on few bits.

This approach enables a more area-efficient summation (the $\boxed{\Sigma}$ box) thanks to the recent advances in bit array compression [28]. It is also more accurate thanks to a global error analysis.

Altogether, in ELMA the sizes of ${}^{\lg}W$ and ${}^{\lg}X$ are 8 bits, and the sum size is 38 bits, whereas the formats we use are respectively 4 and 9 bits.

We have mentioned accuracy twice. It should be clear to the reader that the purpose of computing accurately is to reduce the size of the data format needed. It also helps avoiding the need for retraining.

Indeed, we do not address training in this work: we take networks pre-trained in floating-point [29] and quantize them to our application-specific format, building upon the observation by Lee *et al.*[18] that "Retraining of weights is necessary to enable a good model in linear 4 bits, but unnecessary for $\log_2$ 4 bits (...)". Of course, quantization-aware training can only improve the result, but this is left for future work.

## III. Neuron design and implementation

### A. LNS and sum of products

In both convolutional or fully connected layer, the mathematical operation performed by the neuron is the same: a sum of products. The main difference between these layer types is the way they match inputs with weights, but in the end it always comes down to computing:

$$B + \sum_{i=1}^{N} X_i \times W_i$$

For a linear layer, $N$ is simply the number of neurons in the previous layer. For a convolutional layer, $N$ is the number of weights in the kernel for this layer multiplied by the number of input channels (e.g.: $28 \times 28 = 784$ in the first layer of a Multi-Layer Perceptron (MLP) for MNIST, up to $3 \times 3 \times 512 = 4608$ for a convolutional net on CIFAR). The scope of this work is focused on presenting a cheap hardware implementation of this operator (the so-called LNS neuron) that can be used in either layer type.

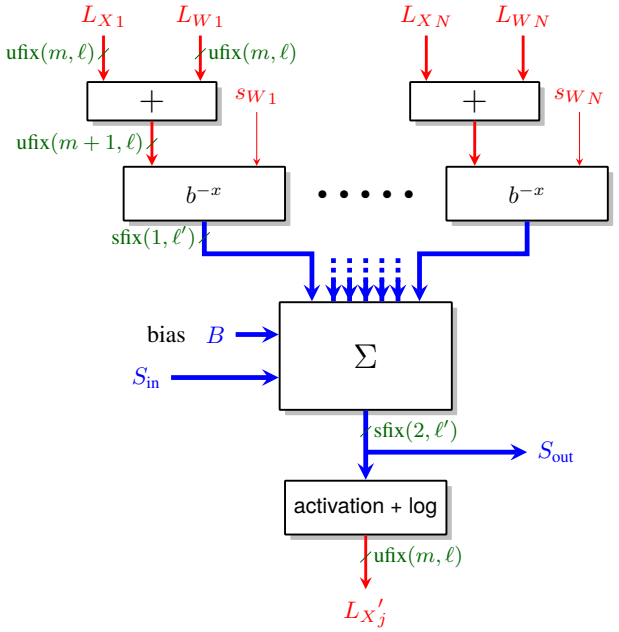



Fig. 2: The proposed low-precision LNS neuron

As mentioned before, focusing on FPGAs and low-bitwidth operands allows us to efficiently use the LUTs by tabulating complex functions. Three such functions are shown in figure 1d: $\boxed{b^x}$, $\boxed{\text{ReLU}}$, $\boxed{\log}$. They can be fairly costly to implement and the activation function (ReLU here) can change between networks (e.g.: sigmoid, tanh, ...). Tabulating them achieves a low implementation cost and flexibility and adaptability in the design. To minimize the cost of translating back and forth between log and linear domain, we fused the $\boxed{\log}$ block to the $\boxed{\text{ReLU}}$ since they happen consecutively: the function tabulated is $\lfloor \log_b (\text{ReLU}(x)) \rceil$.

Figure 2 shows an overview of a complete neuron architecture with its parameters. The next sections will explain the chosen encoding for each step of the process.

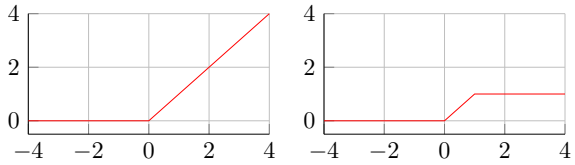



Fig. 3: ReLU (left) and ReLU1 (right) activation functions

### B. Distribution of weights and activations in deep learning

We consider networks which consist of a sequence of convolutional or fully connected layers, possibly with max-pooling layers between them. We ignore the latter as the encoding of activations is irrelevant in their operation (and they cost very little) and focus on the neural layers themselves (convolutional or fully connected). We evacuate the batch normalization layers, useful in training, as they can be removed for inference and their coefficients merged in the weights of the preceding layers (an approach suggested by the inventors of batch normalization [30] and used in other studies, e.g. [25]).

A few observations can be made on their trained weights and runtime activations.

Firstly, as already noted in [22], the output of a ReLU is always positive. As all the activations (except in the input layer) come from a ReLU unit, they are positive (Figure 3). The activations in the input layers can be rescaled and shifted arbitrarily, in particular to ensure they are positive, too. Therefore there is no need to store a sign bit in ${}^{\text{lg}}X$. Weights remain signed and need a sign bit (Figure 4).

We now show how to ensure that all the weights and activations remain smaller than 1 in magnitude, so that we can also spare the sign bits of their logarithmic encoding (the log of a value smaller than 1 is always negative). The constraints $|X_i| < 1$ and $|W_i| < 1$ would be relatively easy to integrate in the training, but we observe that it can also be ensured on trained networks by simple affine transformations of the inputs and weights, thanks to the fact that ReLU is piecewise affine.

Of course, any activation coming from a ReLU1 verifies $0 < X < 1$ by design (Figure 3). If the network was trained with ReLU instead (currently more common case), the batch normalization is designed to force the mean and variance of each layer's output equal to 0 and 1 respectively, so large values of $W_i$ are very unlikely (see Figure 5). Now assume that a layer ensures $X_i < 1$ (again this is easy to ensure for the input layer). The worst case run-time situation would be a neuron with all the activations equal to 1 for the positive $W_i$, whereas activations for negative $W_i$ would all be zero (or the opposite case). In other words, the worst-case activation in absolute value is $\max(-\sum_{W_i<0} W_i, \sum_{W_i>0} W_i)$.

This sum can be computed after training. In the networks we have studied, it does not exceed 25. It is thus possible to divide all the weights of the layer by a small power of two (in practice 8, 16 or 32) to ensure that the activations in the next layer will not exceed 1. In the logarithmic domain, this will consist in adding a small constant to all the $L_W$. The biases

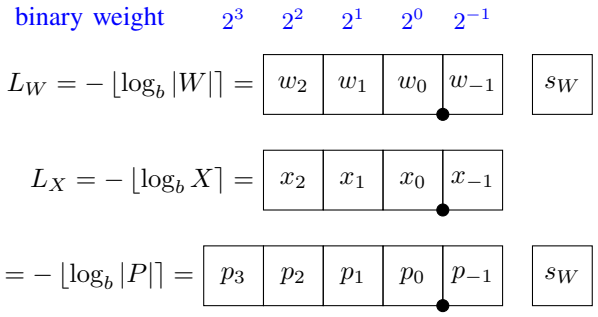



Fig. 4: Logarithmic representation for inputs and product (here for MSB $m = 2$ and LSB $\ell = -1$)

have to be updated as well. All this preprocessing is performed in floating-point and only changes the weights and biases. It has no runtime cost, and no impact on classification accuracy since negative sums remain negative and positive ones remain positive, so the ReLU threshold is unchanged.

Another option would be to cap all the activations to 1 by replacing the ReLU with a ReLU1 (see [31, Figure 3] for the effect of a ReLU1 on the activations of a layer without batch norm). This will require retraining, though.

### C. Ad-hoc LNS formats for the weights and activations

With these observations, we have only negative logarithms for the weights and activations, and we may store the negated logarithm as an unsigned fixed-point format. The proposed logarithmic format for weights and activations is shown in Figure 4. It still has two parameters, the MSB $m$ and the LSB $\ell$ of the parametric representation. $L_W$ is the correct rounding to the nearest of the exact logarithm of $W$ (after the previous affine transformations). The ReLU+log component will also tabulate correctly rounded values.

With these formats, it should be clear on this figure that the product is computed exactly in the logarithmic domain as $L_P = L_X + L_W$. The overflow bit is kept. Remember that we store negated logarithms: an overflow in this addition corresponds to the product of a very small $X$ by a very small $W$ that leads to an even smaller $P$. If this multiplication was performed in fixed point, it would be rounded to zero. If it was performed in floating-point, it would be rounded but the correct order of magnitude would be kept in the exponent of the result. In LNS, the encoding of the exact product of the input data costs only one bit more than these inputs.

It has been observed that weights can use smaller bitwidths than activations [32]. With the proposed approach, it will not save computation hardware (the addition hardware is the size of the wider format, and the smaller one must be padded with zero), but it may save memory and memory bandwidth.

### D. Ad-hoc linear format for the exact sum

Some rounding will still be inflicted to our exact product, however, and this takes place in the $b^x$ box. Its output is again a fixed-point format with an MSB and an LSB. Since the product $P$ is smaller than 1, the MSB is trivially 0. Actually,

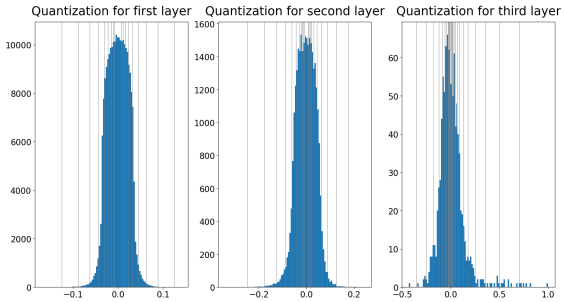
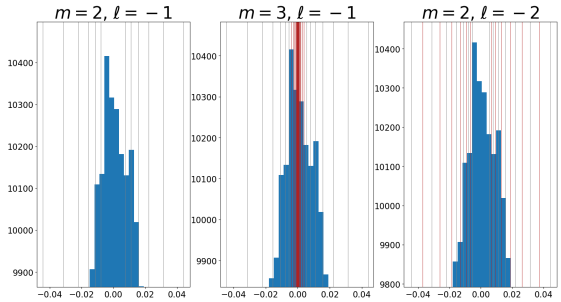



Fig. 5: Quantization indices for $m = 2$ and $\ell = -1$.




(a) zoom on fig. 5, left (b) adding 1 to MSB $m$ (c) subtracting 1 to LSB $\ell$

Fig. 6: Influence of parameters on quantization.

since the network has been preprocessed to guarantee that the output activations is smaller than 1 (in absolute value), the MSB of the sum itself is a sign bit at position 1 (weight $2^1$).

The LSB, however, noted $\ell'$ in Figure 2, is kept open as a third parameter of the proposed approach.

### E. Encoding of zeroes

LNS formats in the literature usually carry a zero bit, since $\log_b(0)$ is not defined, all the more as 0 is an important value in neural networks, widespread across weights and activations. However, a good choice of the $\ell'$ parameter will allow us to discard that extra bit from our LNS encoding, for both ${}^{\lg}X$ and ${}^{\lg}W$. Let us take an example to illustrate this: we will pick $m = 2$ and $\ell = -1$, just as shown in Figure 4. The largest value that can be encoded in $L_X$ is then $(111.1)_2 = (7.5)_{10}$. In base $b = 2$, it is the encoding of $x = 2^{-7.5} = (0{,}000\,000\,010\,110...)_2$. The first significant bit happens at position -8. It means that for $\ell' \geq -7$, the $\boxed{b^x}$ function will round $2^{-(111.1)_2}$ to 0. Referring to Figure 2, we can also tell that if $L_X = (111.1)_2$ or $L_W = (111.1)_2$, then $L_P \geq (111.1)_2$ ensuring that it will be rounded to 0 as well.

In a nutshell, the largest LNS value is in practice an encoding of 0, since when it is used, it will entail that the linear-domain product $P$ is zero. Obviously, the higher $\ell'$, the bigger the rounding errors in the $b^x$ conversion, negatively affecting the classification accuracy. However, picking $\ell'$ lower than or equal to the previous threshold also degrades the accuracy, since every 0 in the network now gets rounded to a small value, which leads to accumulation and classification errors, while costing more to implement.

### F. Discussion on the impact of the format parameters

Figure 5 shows a histogram of the weights by layer of a (300, 100, 10)-MLP trained against MNIST. We can see that the $|W| < 1$ assumption is well met. The graph hints a normal distribution centered on 0. The vertical bars indicate where the LNS quantization would happen. We will later use the histogram of the first layer in other illustrations as it has the highest weight count and so the best visual representation of that distribution. We keep the parameters used in the examples so far because they are a good fit for this application.

Notice that since $X = 2^{-L_X}$, the quantization gets tighter around 0, but there is a gap with no value between $2^{-(2^{m+1}-2^\ell)}$ (Figure 6 (left) is a zoom on the center).

Let us now take a closer look on the impact of $m$ and $\ell$ on this quantization. Adding a bit of representation obviously doubles the number of available quantized values, but in a not so obvious way because of the LNS. Figure 6 highlights in red the additional values added to the quantization when adding one bit to:

- $m$ (center): all the new values are located *inside* the gap previously mentioned. (Because $L_X = -\lfloor \log_b X \rceil =$, so bigger $L_X$ means smaller $X$)
- $\ell$ (right): similar to a standard linear quantization, it adds a value in between existing ones, with no influence on the central gap.

## IV. Experiments

### A. Setup

This work primarily relies on 2 frameworks:

- `flopoco`, an open-source VHDL generator written in C++, to produce the hardware design;
- `pytorch`, an open-source Python machine-learning framework with a high-level interface.

A neuron operator was added to `flopoco`. The classification accuracy of this neuron was evaluated with `pytorch`. A problem was that these frameworks are not fully available in the same language (`pytorch` offers some C++ helpers but only a portion of the interface is available). With the help of the `pytorch`'s cpp extension, we wrote a wrapper to execute C++ code in a python environment. This way, the same C++ emulation function may be used by `pytorch` to compute the neuron output in a bit-accurate simulation, and by `flopoco` to build test benches for the VHDL neuron.

The first experiment was to implement a regular MLP to evaluate the proposed design on the MNIST benchmark (digit classification). This MLP consisted in 3 fully connected (FC) layers of 300, 100 and 10 neurons with no bias. The activation function for all hidden layers was ReLU1. The network was trained in standard 32 bits floating-point precision with `pytorch` against the MNIST dataset, split into a 60 000

TABLE II: VGG-like convolutional network

| layer index | layer type |
|---|---|
| (1) | $LNSConv(3, 128)$ + $ReLU1()$ |
| (2) | $LNSConv(128, 128)$ + $ReLU1()$ |
| (3) | $MaxPool2d(2, 2)$ |
| (4) | $LNSConv(128, 256)$ + $ReLU1()$ |
| (5) | $LNSConv(256, 256)$ + $ReLU1()$ |
| (6) | $MaxPool2d(2, 2)$ |
| (7) | $LNSConv(256, 512)$ + $ReLU1()$ |
| (8) | $LNSConv(512, 512)$ + $ReLU1()$ |
| (9) | $MaxPool2d(2, 2)$ |
| (10) | $LNSConv(512, 1024)$ + $ReLU1()$ |
| (11) | $MaxPool2d(2, 2)$ |
| (12) | $LNSLinear(1024, 10)$ |

samples training set and a 10 000 samples validation set. The learned parameters and input values were then converted to our custom LNS. We ran the bit-accurate simulation of the proposed LNS neuron against the validation set to evaluate the classification accuracy with `pytorch`. We also generated the VHDL code for the largest neurons of the network (in the first layer with $28 \times 28 = 784$ input pairs), and its implementation cost was obtained with Vivado 2021.2 for the target Kintex-7 7k70tfbv484-3 FPGA.

In a second experiment, we implemented a standard convolutional network for the CIFAR-10 image classification benchmark. This standard VGG-like network has been used in other works as baseline for quantization [33], and pretrained weights for this architecture were found in the Pytorch Playground [29]. It consists in a succession of convolution layers with $3 \times 3$ kernels, batch norm and max-pool layers, and a final linear classifier. We merged the activation function in the neuron as previously described, and also merged the batch-norm layers into the weights (as suggested in [30]). The modified architecture used here is detailed in table II.

There was one small issue: the original training used the ReLU activation function, which does not provide the hard cap at 1 that we expect for activations. To solve this without retraining, we statically computed the maximum activation $a_{\max}$ achievable in the entire network with its set of weights. Assuming that the inputs are bounded by 1, $a_{\max}$ is simply the maximum between the sum of the positive values of the weights and the sum of the negative values of the weights. Then we modified the floating-point weights and biases to be sure the output activations would be scaled down by $a_{\max}$. In the input layer, it required scaling both weights and bias:

$$\frac{a}{a_{\max}} = \frac{B}{a_{\max}} + \sum_i \frac{W_i}{a_{\max}} \times X_i \quad . \tag{1}$$

In the hidden and output layers, only the bias requires scaling, since their inputs were already scaled down:

$$\frac{a}{a_{\max}} = \frac{B}{a_{\max}} + \sum_i W_i \times \frac{X_i}{a_{\max}} \quad . \tag{2}$$

All this was performed on the pretrained weights, before conversion to LNS.

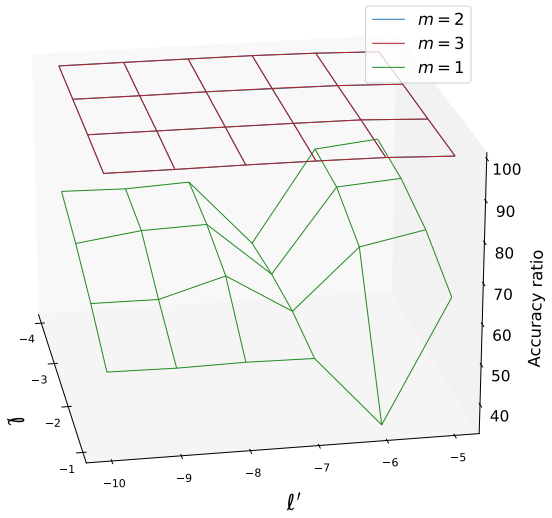



Fig. 7: Influence of parameters on MNIST accuracy

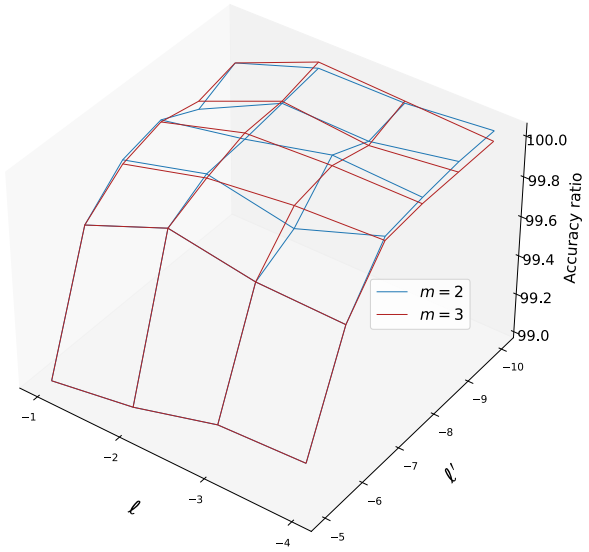



Fig. 8: Zoom on $m = 2$ and $m = 3$ (notice the vertical scale)

### B. *Classification accuracy*

Figure 7 and 8 plot the classification accuracy achieved by the MLP on MNIST for several parameter configurations. The accuracy ratio metric displayed on the graph is the ratio of top-1 accuracies between the quantized version of the network, and the original floating point network.

We explored three different values for $m$. As $m = 1$ yields significantly lower accuracy, it was not considered further. This poor accuracy is understandable considering the associated representation space. Indeed, with $\ell$ being negative, the maximum value for $L_X$ or $L_W$ becomes $2^2 - 2^\ell \approx 4$, hence $X \approx 2^{-4} = 0.0625$. It means that the proposed encoding rounds every weight and activation in the $]-0.0625, 0.0625[$

TABLE III: Synthesis results for accuracy-equivalent parallel neurons. $(m, \ell) = (2, -1)$ means that weights fit on 5 bits and activations fit on 4 bits (see Fig.4).

| benchmark | parameters $(m, \ell)$, $(1, \ell')$ | top-1 accuracy abs. | top-1 accuracy ratio | cost in LUTs | latency in ns |
|---|---|---|---|---|---|
| MNIST | float32 | 98.03 | 100 | - | - |
| MNIST | (2, -1), (1, -6) | 97.64 | 99.6 | 12491 | 10.3 |
| MNIST | (2, -1), (1, -7) | 97.83 | 99.8 | 13790 | 10.9 |
| MNIST | 6-bit linear | 97.93 | 99.9 | 36658 | 10.2 |
| CIFAR10 | float32 | 93.74 | 100 | - | - |
| CIFAR10 | 6-bit linear | 90.83 | 96.9 | 51910 | 13.0 |
| CIFAR10 | (3, -1), (1, -11) | 91,40 | 97.5 | 30632 | 12.8 |
| CIFAR10 | (2, -2), (1, -10) | 92.33 | 98.5 | 28652 | 12.4 |
| CIFAR10 | 8-bit linear | 93.55 | 99.8 | 83522 | 13.4 |

interval to 0. This is a poor match to the weight distribution shown in figure 5, especially in the first layer.

Figure 8 shows that $m = 3$ is not much better in terms of accuracy than $m = 2$, however its hardware cost is significantly higher (see Figure 9).

### C. Hardware implementation

Figure 9 shows how the area evolves with the neuron parameters, using a simple model that estimates the cost of a $2^p \times q$ table (as used for the $b^x$ and ReLU+log blocks) as $2^{p-6} \times q$ LUT, and counts 0.55 LUT per bit added in the summation (see [28] for the adder tree implementation details). This model is quite accurate, for instance Vivado synthesized the ((2, -1), (1, -6)) LNS neuron in 12,491 LUTs where our estimations were 12,890 LUTs. Further comparisons showed that the estimation is about 3% larger than the synthesis. Since the synthesis is computationally demanding, the plot shows the estimated area.

These figures can help us chose the best parameter configuration for our application. In our case for MNIST, we wanted to reduce the number of bits required for the weights and activations as much as possible to help with the memory bandwidth issue, while keeping an acceptable accuracy. This lead us to pick $m = 2$, $\ell = -1$ and $\ell' = -6$ for our previous examples.

Table III provides some actual synthesis results on Kintex7 using Vivado. We also merged here some of the best classification accuracies we could achieve and their configuration. The hardware cost of the lines referred to as "x-bit linear" relies on a standard integer multiplier that feeds its output to the same summation operator than the one used in our LNS neuron. Their associated classification accuracy is obtained with [29], and the number presented here is the top-1 accuracy ratio. This way we can compare to accuracy equivalent linear quantization. The results clearly shows that LNS neurons can achieve a similar accuracy with fewer bits, and a much cheaper hardware cost.

Since this work does not yet provide a complete neural network accelerator, the purpose of this table is merely to show that the proposed neuron fits the target FPGA.

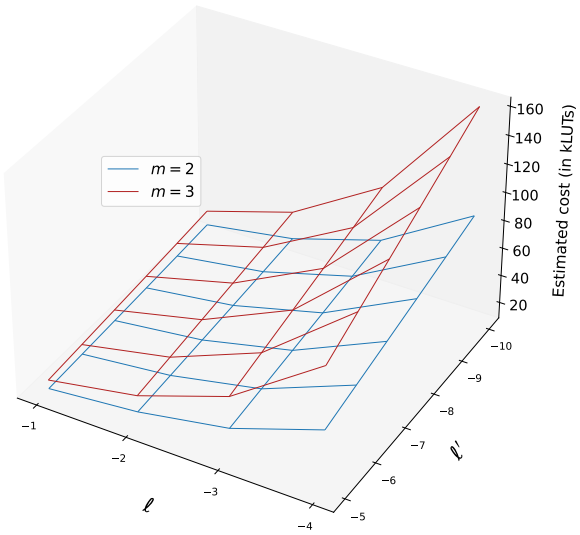



Fig. 9: Parameters influence on hardware cost of a neuron for $N = 784$ (the smallest Virtex-7 FPGA contains more than 360,000 LUTs).

## V. Conclusion and future work

This work proposes a low-bitwidth LNS representation specifically designed for ML inference. With a simple weight conversion without retraining, the accuracy loss compared to the float32 reference can be negligible, using smaller bitwidths than when using fixed-point formats. This enables an efficient FPGA implementation: thanks to the small number of bits, the difficult computations can be tabulated and densely packed into FPGA lookup tables. Compared to a fixed-point neuron with comparable classification accuracy, the proposed LNS neuron features lower resource usage.

As a next step, we plan to build a complete accelerator and to study low-bitwidth LNS-aware training. A simple retraining approach giving encouraging results for a small number of bits has been proposed by [18]. Several other authors have proposed full training strategies [22, 20, 34, 35], but for bitcounts over 8. [36] proposes an approach using 4 bits, but fixes the representation, while our neuron is parametric, which will offer more flexibility. For such a study, the proposed approach also has the potential to use a different base $b$ and an arbitrary activation function, provided that the training ensures that weights and activations remain below 1.

## VI. Acknowledgements

The authors would like to acknowledge the financial support of the French Agence Nationale de la Recherche (ANR) through the MIAI@Grenoble Alpes ANR-19-P3IA-0003 grant and the Imprenum project.

## References


[1] Y. LeCun. "Deep learning hardware: Past, present, and future". In: *IEEE International Solid-State Circuits Conference*. IEEE. 2019, pp. 12–19.

[2] N. Stephens et al. "The ARM scalable vector extension". In: *IEEE micro* 37.2 (2017), pp. 26–39.

[3] M. Arafa et al. "Cascade Lake: Next Generation Intel Xeon Scalable Processor". In: *IEEE Micro* 39.2 (2019), pp. 29–36.

[4] T. Norrie et al. "The Design Process for Google's Training Chips: TPUv2 and TPUv3". In: *IEEE Micro* 41.2 (2021), pp. 56–63.

[5] K. Chatha. "Qualcomm Cloud Al 100: 12TOPS/W Scalable, High Performance and Low Latency Deep Learning Inference Accelerator". In: *IEEE Hot Chips 33 Symposium*. IEEE. 2021, pp. 1–19.

[6] B. Dupont de Dinechin. "A Qualitative Approach to Many-core Architecture". In: *Multi-Processor System-on-Chip 1: Architectures*. John Wiley & Sons, 2021. Chap. 2, pp. 27–52.

[7] R. Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". arXiv:1806.08342. 2018.

[8] B. Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[9] Y. Umuroglu et al. "FINN: A framework for fast, scalable binarized neural network inference". In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 65–74.

[10] A. Prost-Boucle et al. "Scalable high-performance architecture for convolutional ternary neural networks on FPGA". In: *27th International Conference on Field Programmable Logic and Applications*. IEEE. 2017, pp. 1–7.

[11] S. Zhou et al. "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients". arXiv:1606.06160. 2016.

[12] A. Bulat and G. Tzimiropoulos. "XNOR-Net++: Improved binary neural networks". arXiv:1909.13863. 2019.

[13] H. Alemdar et al. "Ternary Neural Networks for Resource-Efficient AI Applications". In: *International Joint Conference on Neural Networks*. IEEE. 2017, pp. 2547–2554.

[14] J. Detrey and F. de Dinechin. "A tool for unbiased comparison between logarithmic and floating-point arithmetic". In: *Journal of VLSI Signal Processing* 49.1 (2007), pp. 161–175.

[15] S. A. Alam, J. Garland, and D. Gregg. "Low-precision Logarithmic Number Systems: Beyond Base-2". In: *ACM Transactions on Architecture and Code Optimization* 18.4 (2021), pp. 1–25.

[16] P. D. Vouzis, C. Collange, and M. G. Arnold. "A Novel Cotransformation for LNS Subtraction". In: *Journal of Signal Processing Systems* 58.1 (2010), pp. 29–40.

[17] J. N. Mitchell. "Computer Multiplication and Division Using Binary Logarithms". In: *IRE Transactions on Electronic Computers* EC-11.4 (1962), pp. 512–517.

[18] E. H. Lee et al. "LogNet: Energy-efficient neural networks using logarithmic computation". In: *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2017, pp. 5900–5904.

[19] M. S. Kim et al. "Low-power implementation of Mitchell's approximate logarithmic multiplication for convolutional neural networks". In: *23rd Asia and South Pacific Design Automation Conference*. IEEE. 2018, pp. 617–622.

[20] M. Arnold, E. Chester, and C. Johnson. "Training neural nets using only an approximate tableless LNS ALU". In: *31st International Conference on Application-specific Systems, Architectures and Processors*. IEEE. 2020, pp. 69–72.

[21] M. Arnold et al. "On the cost effectiveness of logarithmic arithmetic for backpropagation training on SIMD processors". In: *International Conference on Neural Networks*. Vol. 2. 1997, pp. 933–936.

[22] D. Miyashita, E. H. Lee, and B. Murmann. "Convolutional neural networks using logarithmic data representation". arXiv:1603.01025. 2016.

[23] E. Swartzlander and A. Alexopoulos. "The Sign/Logarithm Number System". In: *IEEE Transactions on Computers* C-24.12 (1975), pp. 1238–1242.

[24] M. Arnold et al. "Implementing back propagation neural nets with logarithmic arithmetic". In: *International AMSE Conference on Neural Networks*. 1991.

[25] J. Johnson. "Rethinking floating point for deep learning". In: *Neural Information Processing Systems (NIPS)*. 2018.

[26] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[27] X. Zhang et al. "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs". In: *IEEE/ACM International Conference on Computer-Aided Design*. 2018, pp. 1–8.

[28] M. Kumm and J. Kappauf. "Advanced Compressor Tree Synthesis for FPGAs". In: *IEEE Transactions on Computers* 67.8 (2018), pp. 1078–1091.

[29] A. Chen and G. Smith. *Pytorch playground*. https://github.com/aaron-xichen/pytorch-playground. accessed 2022/02/28.

[30] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *32nd International Conference on Machine Learning*. 2015, pp. 448–456.

[31] A. Krizhevsky. *Convolutional Deep Belief Networks on CIFAR-10*. 2012.

[32] A. Mishra et al. "WRPN: Wide reduced-precision networks". arXiv:1709.01134. 2017.

[33] M. Courbariaux, Y. Bengio, and J.-P. David. "Binaryconnect: Training deep neural networks with binary weights during propagations". In: *Advances in neural information processing systems* 28 (2015).

[34] A. Sanyal, P. A. Beerel, and K. M. Chugg. "Neural network training with approximate logarithmic computations". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2020, pp. 3122–3126.

[35] J. Zhao et al. "LNS-Madam: Low-Precision Training in Logarithmic Number System using Multiplicative Weight Update". arXiv:2106.13914. 2021.

[36] B. Chmiel et al. "Logarithmic Unbiased Quantization: Practical 4-bit Training in Deep Learning". arXiv:2112.10769. 2021.