Efficient Data Encoding for Convolutional Neural Network application

HONG-PHUC TRINH and MARC DURANTON, CEA, LIST, Embedded Computing Laboratory
MICHEL PAINDAVOINE, LEAD, UMR University of Bourgogne - CNRS

This article presents an approximate data encoding scheme called *Significant Position Encoding (SPE)*. The encoding allows efficient implementation of the recall phase (forward propagation pass) of Convolutional Neural Networks (CNN)—a typical Feed-Forward Neural Network. This implementation uses only 7 bits data representation and achieves almost the same classification performance compared with the initial network: on MNIST handwriting recognition task, using this data encoding scheme losses only 0.03% in terms of recognition rate (99.27% vs. 99.3%). In terms of storage, we achieve a 12.5% gain compared with an 8 bits fixed-point implementation of the same CNN. Moreover, this data encoding allows efficient implementation of processing unit thanks to the simplicity of scalar product operation—the principal operation in a Feed-Forward Neural Network.

Categories and Subject Descriptors: B.2.4 [High-Speed Arithmetic]: Cost/performance; I.4.2 [Compression (Coding)]: Approximate Methods

General Terms: Data Representation, Performance

Additional Key Words and Phrases: Significant position encoding, canonical signed digit, convolutional neural network

ACM Reference Format:

Hong-Phuc Trinh, Marc Duranton, and Michel Paindavoine. 2014. Efficient data encoding for convolutional neural network application. *ACM Trans. Architect. Code Optim.* 11, 4, Article 49 (December 2014), 21 pages. DOI: <http://dx.doi.org/10.1145/2685394>

1. INTRODUCTION

Neural Networks (NNs) using real numbers for weights are universal approximators [Hornik et al. 1989] and could be used to solve hard problems such as classification or recognition from images [Krizhevsky et al. 2012; LeCun et al. 2010]. To fully profit from the inherent massive parallelism of neural processing, hardware NN implementations are essential. Therefore, a large variety of hardware NN implementation techniques has been introduced in the last two decades [Moerland and Fiesler 1997; Janardan and Indranil 2010]. Even though these implementations are largely different, all of them try to optimize three main criteria: area, precision, and processing speed.

Data representation is one important factor of a hardware implementation of NN. In the context of ASIC design, it influences directly the hardware area needed for weight storage, area, and time required for arithmetic operations in neural processing together with reliability and accuracy of the results. Floating-point representation with large data range could bring high reliability to the network and increase network capacity

Authors' addresses: H.-P. Trinh and M. Duranton, CEA, LIST, Embedded Computing Laboratory, F-91191 Gif-sur-Yvette, France; email: hong-phuc.trinh@cea.fr; M. Paindavoine, LEAD, UMR University of Bourgogne - CNRS, 21000 Dijon, France.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/12-ART49 \$15.00

DOI: <http://dx.doi.org/10.1145/2685394>

and classification/recognition performance. However, due to the complexity of circuits needed for its arithmetic operations, floating-point representation is not practical for fast and compact hardware NNs (e.g., for embedded or portable systems). Practical hardware implementations of NNs often constrained the design to integer or fixed-point representation in order to make use of simpler integer adder and multiplier [Mauduit et al. 1992; Bermak and Martinez 2003; Farabet et al. 2010]. However, even when using these limited precision representation, hardware implementation NNs still require a lot of area on the chip. In fact, a large number of circuits for addition and multiplication—two principal arithmetic operations using in NNs—are needed in the neuronal processing to fully profit from the inherent parallelism of NNs. Moreover, circuitry devoted to multiplication is far more resource-, power-, and time-consuming than the one devoted to addition. Many works were, therefore, tried to further simplify and speed up the costly multiplication process [Skrbek 1999; Lotrič and Bulić 2012; Gaines 1969].

This work is an attempt to optimize hardware implementation of Feed-Forward Neural Network (FNN) by introducing a multiply-efficient, approximate data encoding, which requires fewer bits for data representation but still has a precision high enough to not degrade the network capacity. The idea came from two important properties of NN classifiers: fault tolerance and robustness to noise [Sequin and Clay 1990; Temam 2012; Esmaeilzadeh et al. 2012; Belhadj et al. 2013]. These properties imply that within some degree of error in its data representation (synaptic weights and signals), the prediction of a NN classifier may still be correct. All data in the network could be approximated to be represented in a more efficient way. Moreover, arithmetic operations between these data should be simpler and easier to implement on ASIC. This approximate data encoding can be used in FNN classifiers such as Convolutional Neural Network (CNN) to achieve high-speed and reduced area implementation without significant degradation in terms of classification performance.

The rest of the article is organized as follows: Section 2 describes in detail the encoding proposal together with its implementing method in case of FNNs. Functional simulations and ASIC-oriented VHDL implementation will be carried out in Section 3, and conclusions and perspectives will be discussed in Section 4.

2. SIGNIFICANT POSITION ENCODING

2.1. Related Work

Practical hardware NN implementations critically demand that circuitry devoted to multiplication is significantly reduced. One solution to reduce it is to use approximate arithmetic. Skrbek [1999] presents an optimized implementation of multiplication, square root, logarithm, exponent and nonlinear activation function by using only linear approximation. For example, 2^x is approximated as $2^{int(x)}(1 + frac(x))$, where $int(x)$ stands for the integral part and $frac(x)$ is its fractional part. The FPGA-based implementation uses only shift registers and adders. However, multiplication is still quite complex:

$$x * y = sign(x) sign(y) \frac{EXP_2(LOG_2(2^n|x|) + LOG_2(2^n|y|))}{2^{2n}},$$

where n is the word length, $EXP_2(x)$ and $LOG_2(x)$ are the linear approximation version of 2^x , and $log_2(x)$.

Another solution is to use multiply-efficient data representation. In [Lotrič and Bulić 2012], in the design of an FNN, the authors replaced the exact multiplication with an approximate version by using an iterative logarithmic multiplier with some correction circuits. The proposed logarithmic multiplier needs fewer resources and consequently leads to designs with more concurrent units on the same chip. Another

attempt is to use bit-serial stochastic computing, a technique that represents real number by stream of random bits [Gaines 1969]. The numeric value is proportional to the density of 1 in the binary sequence. Complex computations such as multiplication could be done by simple bitwise operations on the stream. However, this technique requires a method of generating random bit streams. In terms of hardware, generating (pseudo-)random bits is fairly costly. Therefore, the advantage of simple computations is not so obvious. Nedjah and de Macedo Mourelle [2007] described and compared two FPGA prototype architectures designed to implement Feed-Forward fully connected artificial NNs, the first with traditional binary representation while the second uses stochastic representation. Stochastic representation reduces space requirements to a good extent though resulting networks are slightly slower compared to binary models.

In this work, we propose to combine two solutions—approximate arithmetic and multiply-efficient data representation—by introducing an approximate data encoding scheme that we call *Significant Position Encoding (SPE)*. The proposed data representation uses less than 56.25% number of bits per word compared with the initial 16-bit fixed-point representation (7 instead of 16). Moreover, it promises to be simpler than integer or fixed-point representation in terms of arithmetic operations, especially for the scalar product operation, the principal operation using in FNNs.

The details of our encoding are described in the following text, starting with initial ideas, the initial proposition's limits and how its efficiency could be improved with Canonical Signed Digit (CSD) representation. In the remainder of this section, we demonstrate the beneficial effect of our encoding for FNNs.

2.2. Initial Proposition

In this work, we are interested in the interval $(-1; 1)$ because it is an image of the entire real number $(-\infty; \infty)$ and the product of any two numbers in this interval is also included in this interval. The proposed data encoding will only be used to approximate numeric values in this interval.

Our scheme bases on two main ideas:

- (1) The truncation of a data representation in keeping a fixed number (3 for example) of most significant nonzero digits (digits 1 in binary representation) rather than a normal truncation where one keep a fixed number of most significant bits (including digits 0)
- (2) The use of a delta encoding where we store the differences (distances) between successive positions of non-zero digits rather than directly storing the positions themselves.

To clarify these ideas, consider two 16 fractional bits fixed-point numbers in their binary representation: 0.0010100010001011 (0.158370972) and 0.0101101010001011 (0.353683472). These two binary representations would be first truncated in keeping 3 most *significant 1* with 0.001010001 (0.158203125) and 0.01011 (0.34375). The positions (count from the radix point) of 3 nonzero digits in these two approximate numbers are: {3; 5; 9} and {2; 4; 5}. We can notice that the set of significant positions is always a strictly increasing set. By consequence, we can think of a delta encoding where distances between successive positions are used to represent the value. The two sets {3; 5; 9} and {2; 4; 5} would, therefore, be converted as $\{3 - 0; 5 - 3; 9 - 5\} = \{3; 2; 4\}$ and $\{2 - 0; 4 - 2; 5 - 4\} = \{2; 2; 1\}$ (the zero position is 0). Moreover, given the fact that distances between two consecutive positions are always greater than or equal to 1, we can automatically subtract these distances by 1—the default displacement between two consecutive positions. These two examples may now be represented with $\{3 - 1; 2 - 1; 4 - 1\} = \{2; 1; 3\}$ and $\{2 - 1; 2 - 1; 1 - 1\} = \{1; 1; 0\}$. Figure 1 clarifies the conversion process of the number 0.0010100010001011 (0.158370972).

Fig. 1. Conversion example of a 16 fractional bit fixed-point number into three significant positions SPE.

By consequence, instead of using binary form to express the values, we can use a set of small distances between nonzero positions $\{2; 1; 3\}$ and $\{1; 1; 0\}$ to express these two approximate values: 0.001010001 and 0.01011. To be more general, a set of N numbers $\{P_1; P_2; \dots; P_N\}$ is used to represent a binary number with N digits 1 at positions $\{P_1 + 1; P_1 + P_2 + 2; \dots; P_1 + P_2 + \dots + P_N + N\}$ and all other digits are 0. Note that the numeric value of a fixed-point number with N nonzero digits at positions $\{Q_1; Q_2; \dots; Q_N\}$ from the radix point is an addition of powers of two whose exponents are the values of these positions 2^{-Q_i} . Therefore, the mathematical formula of our scheme is:

$$\{P_1; P_2; \dots; P_N\} = \sum_{i=1}^N 2^{-(\sum_{j=1}^i P_j + i)}. \quad (1)$$

The advantage of this scheme lies in the fact that distances between successive nonzero positions are often quite small and as a result could be represented using just a few number of bits in a binary representation. Let us consider the two aforementioned examples: 0.0010100010001011 and 0.0101101010001011, which are approximated with two sets of small distances $\{2; 1; 3\}$ and $\{1; 1; 0\}$. All these distances are less than or equal to 3. We can, therefore, use 2-bit binary number to represent these distances. The two approximate sets might now be rewritten in binary form: $\{10; 01; 11\}$ and $\{01; 01; 00\}$.

2.3. First Improvement

A problem arises with this initial proposition: How can we represent numbers with distances between successive significant positions that are greater than the maximum number attainable by these limited bits? For example, the number 0.0100000101010011, which should be approximated with the set $\{1; 5; 1\}$, could not be represented using a set of three 2-bit binary distances (since 5 could not be represented using only 2 bits). Our solution for this problem is to represent these high value distances as additions of smaller values that are representable using limited number of bits (e.g., $5 = 3 + 2$ in previous example). To do this, we introduce the convention of *active* and *inactive* distance. If the value of the i^{th} distance equals to its maximum value MAX_{P_i} (all bits equal to 1 in its binary representation), this distance is *inactive*. Otherwise, when there is at least one 0 in its binary representation, the distance is *active*. When a distance is *inactive*, there will be no corresponding significant position at distance $MAX_{P_i} + 1$ from the previous position. The *inactive* distance's value MAX_{P_i} will be added to calculate value of next significant position, which corresponds to the nearest *active* position. The example 0.0100000101010011 would now be approximated with $\{01; 11; 10\}$, which represents the value $0.01000001 = \{1; 5\}$. Here, the second displacement, whose value equals to the maximal value attainable using 2 bits, 11, is *inactive*. It does not represent a significant position at distance $3 + 1$ from the first one, but it does indicate that we must add its value to the third distance, which is *active*, in order to determine the second significant position.

Moreover, using this *active*, *inactive* notion, we can represent numbers that have fewer than N significant positions in an exact way. Consider the number 0.0100 (0.25) that has only 1 significant position at distance 2 from the radix point. There is only 1 valid distance and could be represented using our scheme (with $N = 3$) by the set {01; 11; 11}. In this distance set, only the first one is *active*, the two others are *inactive*. It represents the exact value $2^{-(1+1)} = 0.25 = 0.0100$ (in binary). We could not represent the number without error like this way without using the *inactive* notion because there will be always N significant positions, and as a result, N nonzero digit (digit 1) in the approximate binary form.

By consequence, our approximate data encoding uses a set of N small natural numbers: $\{P_1; P_2; \dots; P_N\}$, where $0 \leq P_1 \leq MAX_{P_1}; 0 \leq P_2 \leq MAX_{P_2}; \dots; 0 \leq P_N \leq MAX_{P_N}$ to represent the real number:

$$\{P_1; P_2; \dots; P_N\} = \sum_{i=1}^N A_i \cdot 2^{-\sum_{j=1}^i (P_j + A_j)} \quad (2)$$

A_i defines whether the i^{th} distance is *active*: if all bits of P_i equal to 1, $A_i = 0$, otherwise, $A_i = 1$. $MAX_{P_i} = 2^{NB_{P_i}} - 1$ with NB_{P_i} the number of bits needed to represent P_i .

2.4. Second Improvement: The Proposed Significant Position Encoding

2.4.1. Binary Signed Digit and Canonical Signed Digit. Binary Signed Digit (BSD) representation [Avizienis 1961] is a representation where each binary digit (0 or 1) is associated with a sign. A numerical value can have several BSD representations. For example, there are many different ways to represent the integer 15: $15 = 01111 = 10001 = 100\bar{1}1 = 10\bar{1}11 = \dots$, where $\bar{1}$ stands for -1 . There is a subgroup of BSD where representation is unique called *Canonical Signed Digit (CSD)*.

CSD is a ternary number system with the digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ stands for -1 [Coleman and Yurdakul 2001]. It is a sum-of-power-of-2 representation with the following properties:

- (1) No two consecutive digits of a CSD number are nonzero (it is in fact the nonadjacent form of BSD representation)
- (2) The CSD representation is unique
- (3) The CSD representation of a number has minimum number of nonzero digits (its Hamming weight is minimal) [Arno and Wheeler 1993]

For example, the CSD representation of 61 (in 8-bit binary 00111101) is 01000 $\bar{1}$ 01. Peled [1976] proved that the probability of occurrence of nonzero digit in a W -bit CSD number is:

$$P = \frac{1}{3} + \frac{1}{9W} \left(1 - \frac{1}{2^W} \right). \quad (3)$$

This means that in CSD representation, only about one third of all digits are nonzero compared with one-half in regular binary representation.

CSD representation has been widely used in reducing the complexity of multiple-constant multiplications (digital filter, convolution, linear transform, etc.) [Peled 1976; Khoo et al. 1996; Skaf and Boyd 2008]. In these works, the constant multipliers are encoded in CSD form, whereas the multiplicands are encoded in normal binary form. The number of SHIFT and ADD operations are reduced thanks to the limitation of nonzero digit of multiplier in CSD form and, therefore, contribute to a reduced area needed for hardware realization.

Fig. 2. Conversion example of a 16-bit CSD number into three significant positions SPE.

2.4.2. Significant Position Encoding. Until now, the proposed scheme tries to approximate a numeric value by an addition of N power-of-2 where exponents are positions of most significant nonzero digits in the number's binary representation. However, for binary numbers with several consecutive 1 such as 0.0111101110111011, the error of approximation could be quite large if only two or three of the most significant 1 are kept: the approximate value is 0.0111(0.4375) and approximation's error is 0.0000101110111011 (0.045822144). If, beside the addition operations, we also use subtraction operations, the binary number (0.0111101110111011) could be approximated by sum (addition and subtraction) of three powers of 2: $0.0111101110111011 = 2^{-1} - 2^{-6} - 2^{-10}$. In this case, approximation error is thus $2^{-14} - 2^{-16}$, much smaller than the initial error $2^{-5} + 2^{-7} + \dots$. By consequence, in order to minimize the truncation error, we propose to first convert the initial binary representation to a sum-of-power-of-2 representation before applying our scheme (truncation using fixed number of nonzero digits and differential encoding). Each position will, therefore, be associated with a corresponding sign, positive or negative, to indicate if we add or subtract the corresponding power-of-2 to the value.

The interesting properties of the CSD mentioned in previous subsection make it a prominent candidate as the data representation to be truncated in our scheme. The use of CSD as initial data representation improves the efficiency of our scheme in two aspects: First, this sum-of-power-of-2 representation minimizes our truncation error because it uses a minimum number of nonzero digits and these nonzero digits are nonadjacent; second, given the fact that in a CSD representation there is at least one zero between two consecutive nonzero digits, the distance between two consecutive positions is always greater than or equal to 2. Therefore, we can further optimize our scheme by subtracting 2 from all the distances instead of a subtraction by 1 like in the initial proposition (except the first one because it is not really a distance, we subtract it by 1—the offset distance from the radix point).

Let us consider an example: The number 0.00101̄0001̄0001001̄ (0.091903687) that has five nonzero digits at positions {3; 5; 9; 13; 16} (count from the radix point) whose signs are {+; -; -; +; -}. Figure 2 shows in detail the conversion process from this CSD value to our SPE representation. We first determined the three most significant positions {3; 5; 9} and their signs {+; -; -}. Then, distances between these successive positions are calculated: $\{3-0; 5-3; 9-5\} = \{3; 2; 4\}$. The set of distances after removing default displacement (2 instead of 1, except the first one) is now {2; 0; 2}. Combining with the set of signs, we now can represent the approximate value with {+2; -0; -2}, which corresponds to the value: 0.00101̄0001̄ (0.091796875).

From now on, our scheme uses small distances between successive positions and signs of N most significant nonzero digits in the CSD representation of a number to approximate its numeric value. Algorithm 1 describes a conversion process that takes into account numbers of bits used to represent each position (and, therefore, maximum numbers representable by each position $\{MAX_{P_i}\}$).

ALGORITHM 1: Conversion from a CSD representation to N-position SPE representation**Input:** CSD number; $\{MAX_{P_1}; MAX_{P_2}; \dots; MAX_{P_N}\}$ **Output:** $\{S_1P_1; S_2P_2; \dots; S_NP_N\}$ **begin** **if** *There is more than N non-zero digits in CSD number* **then** Integer $M \leftarrow N$ **else** Integer $M \leftarrow$ number of non-zero digits in CSD number **end** Determine sets of positions Q_i and signs S_{Q_i} of M most significant non-zero digits $\{S_{Q_1}Q_1; S_{Q_2}Q_2; \dots; S_{Q_M}Q_M\};$ Integer $i \leftarrow 1;$ Integer $j \leftarrow 1;$ $MAX_{pos} \leftarrow -1;$ **while** $i \leq N$ and $j \leq M$ **do** **if** $MAX_{pos} + MAX_{P_i} + 2 > Q_j$ **then** $P_i \leftarrow Q_j - MAX_{pos} - 2;$ $S_i \leftarrow S_{Q_j};$ $i \leftarrow i + 1;$ $j \leftarrow j + 1;$ **else** $P_i \leftarrow MAX_{P_i};$ $S_i \leftarrow S_{Q_1};$ $i \leftarrow i + 1;$ **end** $MAX_{pos} \leftarrow MAX_{pos} + P_i + 2;$ **end** **while** $i \leq N$ **do** $P_i \leftarrow MAX_{P_i};$ $S_i \leftarrow 0;$ $i \leftarrow i + 1;$ **end** **return** $\{S_1P_1; S_2P_2; \dots; S_NP_N\}$ **end**

Mathematically, our approximate data encoding uses a set of N small natural numbers: $\{P_1; P_2; \dots; P_N\}$, where $0 \leq P_1 \leq MAX_{P_1}; 0 \leq P_2 \leq MAX_{P_2}; \dots; 0 \leq P_N \leq MAX_{P_N}$ and N sign bits $\{S_1; S_2; \dots; S_N\}$ to represent the real number:

$$\{S_1P_1; S_2P_2; \dots; S_NP_N\} = \sum_{i=1}^N A_i \cdot (-1)^{S_i} \cdot 2^{1-\sum_{j=1}^i (P_j + 2A_j)} \quad (4)$$

$MAX_{P_i} = 2^{NB_{P_i}} - 1$ with NB_{P_i} the number of bits used to represent P_i . A_i defines whether the i^{th} distance is *active*: If all bits of P_i equal to 1, $A_i = 0$, otherwise, $A_i = 1$. When $A_i = 1$, the sign S_i before each position indicates if we add ($S_i = 0$) or subtract ($S_i = 1$) the corresponding power-of-2 of this position ($2^{1-\sum_{j=1}^i (P_j + 2A_j)}$) to the final value. If the first position is active ($A_1 = 1$), the first sign indicates the sign of the number because adding/subtracting the following powers-of-2 to this first and most significant power-of-2 does not change the value's sign. If the first position is not active while the second is, it is the second sign that indicates the sign of the number, and so on. For ease of comparison between two SPE numbers (cf. Section 2.7.1), we will always set the first sign as the sign of the number, which means, in case of the first position is not active, the first sign will be set equal to the sign of the most significant *active* position

(the lowest active position). From now on, the set $\{NB_{P_1}; NB_{P_2}; \dots; NB_{P_N}\}$ will be used as abbreviation of this SPE configuration, where NB_{P_i} is the number of bits needed to represent the i^{th} position and $MAX_{P_i} = 2^{NB_{P_i}-1}$.

2.5. Distribution and Properties of Significant Position Encoding

Let us consider a three-position SPE configuration $\{2; 3; 2\}$ where $P_1 \leq 3; P_2 \leq 7; P_3 \leq 3$ (P_1 and P_3 can be represented with 2 binary bits while P_2 can be represented with 3 binary bits). Its general formula:

$$\{S_1 P_1; S_2 P_2; S_3 P_3\} = (-1)^{S_1} A_1 \cdot 2^{-(P_1+1)} + (-1)^{S_2} A_2 \cdot 2^{-(P_1+P_2+3)} + (-1)^{S_3} A_3 \cdot 2^{-(P_1+P_2+P_3+5)} \quad (5)$$

Some special values are:

- Maximal positive value $\{0(+)00; 0(+)000; 0(+)00\}$, which equals $2^{-1} + 2^{-3} + 2^{-5}$ (0.10101)
- Minimal negative value $\{1(-)00; 1(-)000; 1(-)00\}$, which equals $-2^{-1} - 2^{-3} - 2^{-5}$ (-0.10101)
- Zero $\{S_1 11; S_2 111; S_3 11\}$, which equals $\pm 0 \pm 0 \pm 0$ (0.00000)
- Minimal positive value $\{0(+)11; 0(+)111; 0(+)10\}$ represents $0+0+2^{1-(3+2+7+2+3+2)} = 2^{-18}$

Note that this minimal positive value is not the step value (difference between two consecutive numbers) of this encoding. This step varies according to the numeric value. For example, the difference between two consecutive numbers $\{0(+)10; 0(+)110; 0(+)10\}$ and $\{0(+)10; 0(+)110; 0(+)11\}$: $2^{-3} + 2^{-11} + 2^{-15}$ and $2^{-3} + 2^{-11} + 0$ equals 2^{-15} . The minimal step is the difference between the minimal positive value and zero and equals the minimal positive value 2^{-18} .

This encoding can represent as many negative numbers as positive numbers. By changing the sign before all positions (or just all active positions) of an SPE number, we obtain its negative. Approximation errors of numbers represented using this encoding depend on the number's value and vary according to SPE configuration. However, maximum relative error depends only on number of positions used: In 1-position configuration (powers-of-2), the maximum relative error is $2^{-1} = 50\%$; in 2-position configuration, the maximum relative error is $2^{-1}/2^2 = 2^{-3} = 12.5\%$, where 2^2 is the minimum rapport between values represented by first and second position (because the distance between two consecutive positions is greater than or equal to 2); in 3-position configuration, the maximum relative error is $2^{-1}/2^4 = 2^{-5} = 3.125\%$. In the initial proposed scheme, where binary representation is used as data representation to be truncated, the maximum relative error in 2-position configuration is about 25% and the maximum relative error in 3-position configuration is about 12.5%. These observations confirm that using CSD as initial data representation allows reducing the approximation error of our scheme compared with the initial proposition.

Figure 3 shows the distribution of numbers in open interval $(-1; 1)$ that can be represented using 2- or 3-position SPE. We see that this representation is nonuniform, not dense, and centered around 0 and other powers-of-2 ($\pm 0.5; \pm 0.25; \pm 0.125; \dots$). In general, numbers whose value is around zero or powers-of-2 could be well approximated, whereas truncation error of values in the middle between two consecutive powers-of-2 is quite large. Simple configuration like $\{3; 2\}$ (two positions, the first with 3 bits and the second with 2 bits) could not well approximate values outside interval $(-0.25; 0.25)$ with maximum relative error but could be used to approximate values lying inside this interval very well (cf. Figure 3(b)). More complex configurations (three positions, each

Fig. 3. Distribution of numbers representable by some SPEs.

with 2 or 3 bits, cf. Figures 3(h), 3(i), 3(j)) are more dense and could represent all values inside interval $[-0.65625; 0.65625]$ quite well.

2.6. Feed-Forward Neural Network

A Feed-Forward Neural Network (FNN) is an artificial NN where connections between neurons—the processing unit—do not create a directed cycle. In these networks, neurons are generally arranged in layers. A neuron in a certain layer receives information from neurons in previous layers through weighted connections (synapses). It computes the weighted sum of its inputs $v_j = \sum_i x_i w_{ij}$ and passes this value through an activation function to obtain its output $x_{out} = f(v_j)$ (cf. Figure 4). Information moves forward from the input layers, through intermediate layers, and to the output layers. By consequence, for a forward pass, arithmetic operations needed in this kind of network are scalar product (calculate the weighted sum v) and activation function. There are three principal types of activation functions:

(1) Threshold function:

$$f(v) = \begin{cases} 1 & \text{if } v \geq \alpha \\ 0 & \text{if } v < \alpha \end{cases}$$

Fig. 4. Artificial neuron model.

(2) Piecewise-linear function

$$f(v) = \begin{cases} 1 & \text{if } v \geq \alpha \\ v & \text{if } \alpha > v \geq -\alpha \\ 0 & \text{if } v < -\alpha \end{cases}$$

(3) Sigmoid or hyperbolic tangent function

$$f(v) = \frac{1}{1 + e^{-x}}$$

$$f(v) = a * \tanh(b * v)$$

The two first types could be implemented easily in hardware with simple comparator module. However, straightforward implementation of the third type is not practical in hardware because we need to implement exponential and division operations, which are too time- and area-consuming. Three possible approaches to approximate this function are to use lookup table, piecewise linear function, and piecewise nonlinear function.

2.7. Feed-Forward Neural Network Arithmetic Operations Using SPE

Szabó et al. [2000] presented a bit-serial/parallel NN implementation method for pre-trained networks. This method utilizes the advantages of CSD encoding and bit-level pattern coincidences to efficiently implement a matrix-vector multiplier, therefore, significantly reducing the hardware cost. However, only synaptic weights are represented with CSD encoding; other data such as neuronal outputs are still binary number. The resulting architecture performs full-precision computation and, therefore, may not profit from the robustness to error of an NN. In this work, we use the proposed SPE as the only data representation for all operations and perform an approximate computation with these data.

One advantage of the proposed SPE lies in the utilization of small displacements between positions to approximate numeric value. Therefore, arithmetic operations could be efficiently implemented using small fan-in logic. For example, the product of two SPE numbers could be obtained directly by simple additions of their positions (because they are sum-of-power-of-2). For instance, Equation (6) gives the product of two 2-position SPE numbers:

$$\begin{aligned} \{\pm P_1; \pm P_2\} * \{\pm Q_1; \pm Q_2\} = & \pm A_{P_1} A_{Q_1} \cdot 2^{-(P_1+Q_1+2)} \pm A_{P_1} A_{Q_2} \cdot 2^{-(P_1+Q_1+Q_2+4)} \\ & \pm A_{P_2} A_{Q_1} \cdot 2^{-(P_1+Q_1+P_2+4)} \pm A_{P_2} A_{Q_2} \cdot 2^{-(P_1+Q_1+P_2+Q_2+6)} \end{aligned} \quad (6)$$

Note that in our encoding, P_i and Q_i are small number that can be represented by 1, 2, or 3 bits. By consequence, we need at most four 3-bit adders in order to perform this multiplication.

In the context of this work, we are interested in the forward propagation of a pre-trained NN, where comparison (max operator), weighted sum of inputs of a neuron and activation function are the only operations needed [Hornik et al. 1989; LeCun et al. 2010]. We will consider these operations in detail. The most simple 2-position SPE will be used as data representation in this consideration for ease of convenience. Operations between other SPEs could be done in a similar manner.

2.7.1. Comparison. A comparison between two SPE numbers could be done efficiently in a serial manner: First, we compare the first signs and positions of two numbers. If they are different, the relationship between two numbers is determined and the comparison will be stopped; if not, we continue to compare next signs and positions and so on.

To give more detail, let us compare two 2-position SPE numbers: $A = \{S_{P_1}P_1; S_{P_2}P_2\}$ and $B = \{S_{Q_1}Q_1; S_{Q_2}Q_2\}$. Because S_1 always indicates the sign of the number (cf. Section 2.4.2), if $S_{P_1} < S_{Q_1}$, we are sure that A is positive ($S_{P_1} = 0$) and B is negative ($S_{Q_1} = 1$); therefore, $A > B$. If $S_{P_1} = S_{Q_1}$, we compare now P_1 and Q_1 . If $P_1 < Q_1$ (including the case that all bits of Q_1 equal to 1, which means Q_1 is inactive), then if $S_{P_1} = S_{Q_1} = 0 : A > B$, if $S_{P_1} = S_{Q_1} = 1 : A < B$. If $P_1 = Q_1$, we repeat this procedure with second positions and so on.

The complexity of this comparison is, therefore, comparable with a comparison between two normal signed binary numbers created by concatenating the signs and positions together: $S_1P_1S_2P_2$.

2.7.2. Convolution or Scalar Product Module. The weighted sum is the scalar product (or a convolution) between synaptic weights and inputs of a neuron: $\sum_{i=0}^n x_i \cdot W_i$. In our scheme, the product of two SPE numbers is normally not an SPE number (because of the sparsity of this representation, cf. Figure 3). Therefore, we must reapproximate the result into SPE form. By consequence, if we calculate separately each product $x_i \cdot W_i$, convert this value to SPE representation, and then calculate the sum of these n products, the final error will be very large because it is accumulated after many operations. Moreover, the addition of two SPE numbers is not straightforward and is quite difficult to implement. Its implementation might be more complex than the addition of two binary numbers. To avoid this problem, we propose different methods to calculate a scalar product. Consider $\{\pm P_1; \pm P_2; \dots; \pm P_{M_1}\}$ the set of activated positions of x_i in SPE format, $\{\pm Q_1; \pm Q_2; \dots; \pm Q_{M_2}\}$ the set of activated positions of W_i . The product $\{\pm P_1; \pm P_2; \dots; \pm P_{M_1}\} * \{\pm Q_1; \pm Q_2; \dots; \pm Q_{M_2}\}$ has value:

$$\sum_{i=0}^{M_1} (-1)^{sign_i} 2^{-exp_i} * \sum_{j=0}^{M_2} (-1)^{sign_j} 2^{-exp_j} = \sum_{i=0}^{M_1} \sum_{j=0}^{M_2} (-1)^{sign_i + sign_j} 2^{-(exp_i + exp_j)}. \quad (7)$$

First approach: The scalar product, which is in fact the sum of these sum-of-power-of-2, could be done by using counters to count the number of each power-of-2 that participate in the final sum. A counter must be able to count up to n , where n is the size of input synaptic weight vector. Moreover, depending on the sign of each operand ($sign_i \cdot sign_j$), the counter will be incremented or decremented. If the precision (the minimal step) of this SPE is 2^{-Pre} , the number of counters needed for a calculation without error is 2^{Pre} . The precision of this method depends directly on number of counters used. Therefore, this method could give a result much more precise than the result of a fixed-point scalar product. Using this method, error does not depend on the synaptic weight vector length and appears only at the final step—when we calculate the final sum before converting it to SPE representation. While in fixed-point

calculation, an error appears in each calculation of product of synaptic weight and input. The longer the input vector is, the larger the error.

Second approach: The calculation is a sum-of-power-of-2 where the exponent of each operand is a sum of two different initial exponents. There are at most $M_1 \cdot M_2$ exponents to be calculated, each by one or some additions of P_i and Q_i . These exponents are in fact positions of nonzero digits (1 and -1) in the CSD representation of the product. We can, therefore, obtain 2 binary numbers: positive (all positions with digit 1) and negative (all positions with digit $\bar{1}$ or -1). Normal binary adders are then used to accumulate these positive and negative values over all products. At the end, the final result is obtained by subtracting the positive value to the negative value. This result is in binary representation and could be converted to SPE representation or used directly as input of the next operation (input of activation function—tangent hyperbolic—in case of NN application).

2.7.3. Activation Function. FNNs usually use threshold, piecewise linear function, and hyperbolic tangent/sigmoid function as activation functions. The threshold or piecewise linear activation function with SPE as data representation could be implemented easily in hardware by using just a quite simple comparator module. However, straightforward implementation of hyperbolic tangent/sigmoid functions in hardware is not easy because of their exponential nature. The most common method to implement these hyperbolic/sigmoid functions in hardware is to use piecewise linear approximation or Look-Up Tables (LUTs). Both methods could be implemented efficiently using SPE: In piecewise linear approximation, we must do some addition/multiplication with constants, and these operations can be implemented easily with the SPE number; LUTs with an SPE number as input cost less than LUTs with fixed-point input because an SPE number requires a smaller number of bits to represent. In this work, we use tangent hyperbolic as neuron's activation function and use the LUT method for its implementation.

3. IMPLEMENTATION EXPERIMENTS

3.1. Convolutional Neural Network

Convolutional Neural Networks (CNNs) were initially introduced by LeCun [1998] for the handwriting recognition task. It is an FNN that combines three ideas of network structure to ensure a certain degree of translational and distortional invariance: local receptive fields, weight sharing, and spatial subsampling. The network is divided into different layers of different types: a convolutional layer, a subsampling layer, and a fully connected layer. Each layer is composed of one or several planes. These planes are composed of many cells (or neurons) and are called *feature maps*. Each cell of a feature map receives connections from a local neighborhood of one or several feature maps of the previous layer: This is the idea of local receptive fields. In a convolutional layer, all the cells of a feature map have in common a convolution kernel. Thus, it allows the extraction of the same local characteristic of the previous representation. A spatial subsampling layer reduces the size of the feature maps and allows obtaining some degree of invariance. A CNN is generally composed of two parts: feature extraction using convolutional and subsampling layers, and classifier (normally of type MLP—Multi-Layer Perceptron), taking as input the results of the last subsampling layer (see Figure 5).

CNNs are often used in image recognition systems. They allow achieving state-of-the-art performance on many vision tasks, from a quite simple task like classification of handwritten digits on the MNIST database¹ [Wan et al. 2013] to a more complex

¹MNIST handwritten digits database, <http://yann.lecun.com/exdb/mnist/>.

Fig. 5. A typical CNN architecture.

Fig. 6. CNN architecture used for the MNIST handwritten recognition task.

task like in the German Traffic Sign Recognition Benchmark (GTSRB)² [Ciresan et al. 2012]. Recently, Krizhevsky et al. [2012] achieved a breakthrough performance on the ImageNet dataset,³ bringing down the state-of-the-art error rate from 26.1% to 15.3%.

3.2. Software Implementation

Simulations have demonstrated the feasibility and efficiency of our proposed encoding. We have used C++ and VHDL (behavior) implementations of the forward propagation of CNNs using SPE as a data type for the MNIST handwritten recognition task and GRSTB. The topology of our network for the MNIST handwritten task is a simple CNN based on proposition of Simard et al. [2003], which uses two combined convolutional-subsampling layers (a convolution kernel size of 5 and a subsampling of 2) as feature extraction (cf. Figure 6).

For GTSRB, we use the feature extractor, as shown in Figure 7, and a classifier consists of 2 Fully-Connected Layers: fc1 with 200 neurons and fc2 with 43 neurons (which is the number of classes of traffic signs in this task). Synaptic weights are learned *offline* with high precision (floating point) using the C++ version. All signals and weights are then converted (approximated) to SPE representation. All operations in the network are done using data in SPE form. Except for the initial *offline* conversion, no more conversion modules are needed. As mentioned earlier, our encoding can represent numbers in range $[-0.65625; 0.65625]$, with a distribution quite dense around powers-of-2 values like 2^{-1} , 2^{-2} , 2^{-3} ... and especially dense around 0. To fully take advantage of our encoding, we scaled our CNNs such that the values of not only synaptic weights but also neuronal signals are small, centered at 0 and belong to the segment $[-0.5; 0.5]$. As shown in Figure 8, the synaptic weights of a trained CNN have a normal distribution with mean

²<http://benchmark.ini.rub.de/?section=gtsrb&subsection=results>.

³The 1000-class ImageNet benchmark, <http://www.image-net.org/challenges/LSVRC/2012/results.html>.

Fig. 7. CNN architecture used for GTSRB.

Fig. 8. Distribution of synaptic weights of a trained CNN.

0 and very small standard deviation. Almost all synaptic weights lie in the segment $[-0.2; 0.2]$ and, therefore, can be well approximated by our encoding.

Figure 9 shows number of misclassifications of various SPE-encoded networks on the MNIST testing set. The initial value 70 (red columns) corresponds to the number of misclassifications of the initial trained CNN using floating-point precision (70 errors/10,000 samples or a recognition rate of 99.3%). The state-of-the-art result on MNIST handwriting recognition task attains 23 errors/10,000 samples (recognition rate of 99.77%), but it uses a committee of 35 networks, and each of them has a larger capacity (and number of parameters) than ours [Ciresan et al. 2012]. This figure points out that the classification performance of networks using 3-position SPE does not degrade much. Except the case of $\{2; 1; 1\}$ SPE topology where number distribution is too sparse in comparison to the rest (cf. Figure 3(d)), and performance degradation in all other cases does not exceed 12% in terms of number of misclassification and does not exceed 0.08% in terms of recognition rate. Even when a simpler topology (2-position SPE $\{3; 2\}$) is used, we have only 73 errors (recognition rate of 99.27% compared with the initial recognition rate 99.3%). It means that when using only a data type consisting of 7 bits ($\{3; 2\}$ SPE configuration: 2 bits in 2 signs, 3 bits in the first position, and 2 bits in the second position), our network can achieve almost the same classification

Fig. 9. Number of misclassifications of several SPE-encoded networks on the MNIST testing set (10,000 samples).

Fig. 10. Number of misclassifications of several SPE-encoded networks on the MNIST training set (60,000 samples).

performance as a network using a floating point: We loss only 0.03% in terms of recognition rate.

We have tested our SPE networks on several differently trained networks (with different classification results), and even on the MNIST training set where the trained network attains a very good recognition rate (more than 99.9%). We observed almost the same behavior in all these cases: In using our data type of 7 bits (the {3; 2} SPE configuration), performance is only slightly degraded (cf. Figure 10).

To see the broad applicability of this scheme, we have also implemented and tested several SPE-encoded CNNs for a more sophisticated task: the GTSRB task. Similar behavior in the dependence of classification results on SPE configurations is also observed and presented in Figure 11. Among all SPE configurations tested, the {3; 2} configuration seems to be the best: It uses only 7 bits and allows achieving a recognition rate comparable with the initial recognition rate.

For a fairer comparison, we have implemented a fixed-point version of our trained network and tested its classification performance in function of number of fractional bits used (see Figures 12 and 13). A lower number of fractional bits fixed-point network give quite bad results: 898 errors (6 bits) and 5,373 errors (5 bits) in 10,000 MNIST samples; 1,282 errors (6 bits) and 9,031 errors (5 bits) in 12,630 GTSRB samples. In using a fixed-point data type of less than 8 bits, the classification error becomes quite large compared with the initial error of the trained network. The performance of the network is then largely degraded. Therefore, to obtain a comparable classification

Fig. 11. Number of misclassifications of several SPE-encoded networks on the GTSRB testing set (12,630 samples).

Fig. 12. Number of misclassifications of fixed-point encoded networks on the MNIST testing set in function of fractional bits number.

Fig. 13. Number of misclassifications of fixed-point encoded networks on the GTSRB testing set in function of fractional bits number.

Fig. 14. Schematic of the convolution module using SPE.

result, a fixed-point data type of at least 8 bits should be used. This conclusion is in accordance with many works in the field: NN forward propagation generally requires a data representation of at least 8 bits [Janardan and Indranil 2010]. It should be also noted that NN hardware implementations usually use 16-bit fixed-point representation (for synaptic weights and/or neuronal outputs) to ensure a proper operation of the learning process [Cloutier and Simard 1994; Chen et al. 2014].

By consequence, the implementation cost of our SPE-encoded network should be compared with that of an 8-bit fixed-point network.

3.3. VHDL Coding for ASIC Implementation

To see whether our SPE encoding has better efficiency compared with fixed-point implementations in hardware, we have implemented an ASIC-oriented RTL designs using VHDL hardware description language of two convolution modules using SPE and fixed point as data representation. The Synopsys Design Compiler is used for simulations and synthesis with a 40nm TSMC N40LP technology node. The two implementations use a serial approach to calculate the value: $\sum_{i=1}^{25} x_i \cdot W_i$: W_i and x_i are loaded serially and are inputs of a multiplier module, whose aim is to calculate the binary product of these two inputs W_i and x_i . This binary product is then accumulated over time (by using a register and a normal binary adder). The main difference between these two modules is thus the multiplier module. Figure 14 shows the schematic of the top module and the multiplier module in our {3; 2} SPE implementation.

The *MULT_SPE* module allows calculating the binary product of two {3; 2} SPE-encoded numbers: x_i and W_i . It uses a 3-bit adder, a 6-bit input, a 9-bit output LUT, and a shifter. Module's outputs consist of a sign and an unsigned 9-bit binary number, which is the absolute value of the product. These outputs will be used in a simplified version of a scalar module (cf. Section 2.7.2), where we use the signed product directly rather than divide it by two parts: positive and negative.

In fact, the product of x_i and W_i could be rewritten as:

$$\begin{aligned} x_i * W_i &= \{(-1)^{S_1} 2^{-(P_1+1)} + (-1)^{S_2} 2^{-(P_1+P_2+3)}\} * \{(-1)^{S'_1} 2^{-(P'_1+1)} + (-1)^{S'_2} 2^{-(P'_1+P'_2+3)}\} \\ &= (-1)^{S_1} 2^{-(P_1+1)} \{1 + (-1)^{S_{21}} 2^{-(P_2+2)}\} * (-1)^{S'_1} 2^{-(P'_1+1)} \{1 + (-1)^{S'_{21}} 2^{-(P'_2+2)}\} \quad (8) \\ &= (-1)^{S_1+S'_1} 2^{-(P_1+P'_1+2)} F(S_{21}, P_2, S'_{21}, P'_2), \end{aligned}$$

where $S_{21} = S_2 \text{ xor } S_1$; $S'_{21} = S'_2 \text{ xor } S'_1$ and $F(S_{21}, P_2, S'_{21}, P'_2)$ is a function of $S_{21}, P_2, S'_{21}, P'_2$. The value of $F(S_{21}, P_2, S'_{21}, P'_2)$ is in fact the product's absolute value

Table I. Comparison between Convolution Modules Using 8- and 16-bit Fixed-Point Representations and 7-bit SPE Representation

Unit	Fixed Point 16 bits	Fixed Point 8 bits	SPE	Gain SPE/fix8 (%)
Power (mW) @250MHz	0.1484	0.0738	0.0639	13.41
Area (μm^2) @ 250MHz	2,314	1,007	809	19.66
Maximum frequency (Hz)	650	830	1,000	20.48

Fig. 15. Evolution of power consumption (a) and circuit area (b) in function of clock period in two different convolution modules using SPE and 8-bit fixed-point data representation.

shifted by $P_1 + P'_1 + 2$. It could be obtained in hardware by a 6-bit input LUT: 2 bits for 2 signs S_{21} and S'_{21} , 2 bits for P_2 and 2 bits for P'_2 .

Synthesis result. Synthesis and timing analysis of the two SPE and fixed-point convolution modules has been performed using the Synopsys Design Compiler with 40nm TSMC N40LP technology node. The Process-Voltage-Temperature corner used is process 1.0, voltage 0.99V, and temperature 125°C. The SPE convolution module uses {3;2} SPE configuration, which is a 7-bit data representation. Both 8-bit and 16-bit fixed-point data representation are used in the other module and will be used as a comparison target (cf. Section 3.2). The synthesis result is presented in Table I and Figure 15: Table I shows a comparison between different convolution modules in terms of power consumption, circuit area (at 250MHz), and maximum frequency, whereas Figure 15 shows the evolution of their power consumption and circuit area in function of clock period.

Here, the maximum frequency or minimum clock period is measured using Design Compiler Timing Analysis. This measure depends on the critical path or the path with the longest delay between any two registers clocked by the system clock. Maximum frequency corresponds to the minimal clock period at which there is no violation in timing throughout the circuit.

Figure 15 shows that our SPE implementation always surpasses the 8-bit fixed-point implementation in terms of power consumption and circuit area for all functional frequencies. Its maximum functional frequency is also about 20% greater than that of 8-bit fixed-point convolution module. Moreover, this SPE encoding use only 7 bits, we, therefore, have a gain of 12.5% in terms of storage, which is not negligible in NNs, seeing that synaptic weight storage is the most area-consuming part in an NN. A closer look in the hierarchy shows that it is the *MULT_SPE* module who brings these gains:

Table II. Comparison between Convolution Modules Using 7-bit Fixed-Point Representation and 7-bit SPE Representation

Unit	Fixed Point 7 bits	SPE	Gain SPE/ <i>fix7</i> (%)
Power (<i>mW</i>) @250MHz	0.0696	0.0639	8.19
Area (μm^2) @ 250MHz	907	809	9.79
Maximum frequency (Hz)	900	1000	10.0

At 250MHz, the *MULT.SPE* module consumes 10% less energy and 52.6% less area ($200\mu m^2$ versus $385\mu m^2$ for optimized 8-bit fixed-point multiplication module).

To see whether these gains really come from the efficiency of our encoding in terms of arithmetic operations or just come from the fact that we use less bits in our scheme, we have also realized a comparison of two modules using the same bit-width: 7-bit fixed-point and 7-bit SPE ($\{3; 2\}$ configuration). The synthesis result (see Table II) shows that even with the same bit-width, we have always gains about 10% in all three criteria: power, surface, and speed.

3.4. Discussion and Future Works

In this work, we have presented an efficient data encoding that could be used in an FNN. Its major advantages lie in the reduction of area required for data storage (gain of 12.5% compared with an 8-bit fixed-point version) and the efficiency of scalar product—the principal operation in a NN. In terms of data storage, 12.5% seems to not much. However, we must note that in NNs, especially in Deep NNs, hundreds of thousand neurons and millions of synaptic weights are used [Krizhevsky et al. 2012]. Therefore, this gain could be quite significant. In terms of scalar product operation, our SPE module surpasses a normal optimized 8-bit fixed-point scalar product module in all three criteria: area, speed, and energy. Its gains are 19.7%, 20.5%, and 13.4%, respectively.

Another advantage that we have not mentioned in previous sections is the possibility to use the proposed encoding directly from a pretrained FNN (using floating-point). We do not need to retrain the network to adapt the new encoding. The proposed encoding is accurate enough (cf. Section 2.4.2) to not degrade the network performance when using the encoded parameters directly converted from initial floating-point representation. This characteristic allows saving a lot of work in the adaptation phase compared with other weight discretization techniques. In fact, other works in reducing numerical precision of data representation in NNs [Moerland and Fiesler 1997; Szabó et al. 2000] need either a hardware-friendly learning algorithms (perturbation learning [Jabri and Flower 1992], cascade error projection learning [Duong and Stubberud 2000], local learning [Chen et al. 2004]), or an adaptation phase where the learned parameters are adjusted to suit the new data encoding.

In terms of future work, a complete implementation of the encoded CNN would be done. Other applications in which the proposed encoding could be used will also be studied.

4. CONCLUSION

In this article, we have proposed an approximate data encoding called *Significant Position Encoding (SPE)*. The encoding uses a set of small distances between successive positions of nonzero digits in a CSD representation to express a numeric value. It is multiply-efficient and allows efficient implementation of the forward propagation pass of CNNs. In this implementation, we use only 7 bits of data representation and achieve almost the same classification performance as the initial pretrained network:

On the MNIST handwriting recognition task, using this data-encoding scheme loses only 0.03% in terms of recognition rate (99.27% vs. 99.3%). Therefore, in terms of storage, we achieve a 12.5% gain compared with an 8-bit fixed-point implementation of the same CNN. Moreover, with this data encoding, we have also achieved significant gains in terms of surface, energy and speed of arithmetic operations used in the NN. These gains are 13.4% in energy, 19.7% in surface, and 20.5% in maximum frequency.

REFERENCES

- S. Arno and F. S. Wheeler. 1993. Signed digit representations of minimal hamming weight. *IEEE Trans. Comput.* 42, 8 (Aug. 1993), 1007–1010. DOI: <http://dx.doi.org/10.1109/12.238495>
- A. Avizienis. 1961. Signed-digit number representations for fast parallel arithmetic. *IRE Trans. Electron. Comput.* 3 (1961), 389–400.
- B. Belhadj, A. Joubert, Z. Li, R. Hélot, and O. Temam. 2013. Continuous real-world inputs can open up alternative accelerator designs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 1–12.
- A. Bermak and D. Martinez. 2003. A compact 3D VLSI classifier using bagging threshold network ensembles. *IEEE Trans. Neural Networks* 14, 5 (2003), 1097–1109.
- T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 269–284.
- Z. Chen, S. Haykin, and S. Becker. 2004. Theory of monte carlo sampling-based alopex algorithms for neural networks. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*. 17–21.
- D. Ciresan, U. Meier, and J. Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12)*. IEEE, 3642–3649.
- J. Cloutier and P. Y. Simard. 1994. Hardware implementation of the backpropagation without multiplication. In *Proceedings of the 4th International Conference on Microelectronics for Neural Networks and Fuzzy Systems*. IEEE, 46–55.
- J. O. Coleman and A. Yurdakul. 2001. Fractions in the canonical-signed-digit number system. In *Proceedings of the Conference on Information Sciences and Systems*.
- T. A. Duong and A. R. Stubberud. 2000. Convergence analysis of cascade error projection—an efficient learning algorithm for hardware implementation. *Int. J. Neural Syst.* 10, 03 (2000), 199–210.
- H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 449–460.
- C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. 2010. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*. IEEE, 257–260.
- B. R. Gaines. 1969. Stochastic computing systems. In *Advances in Information Systems Science*. Springer, 37–172.
- K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2, 5 (1989), 359–366.
- M. Jabri and B. Flower. 1992. Weight perturbation: An optimal architecture and learning technique for analog VLSI feedforward and recurrent multilayer networks. *IEEE Trans. Neural Networks* 3, 1 (1992), 154–157.
- M. Janardan and S. Indranil. 2010. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing* 74, 13 (2010), 239–255. DOI: <http://dx.doi.org/10.1016/j.neucom.2010.03.021>
- K. Y. Khoo, A. Kwentus, and A. N. Willson Jr. 1996. A programmable FIR digital filter using CSD coefficients. *IEEE J. Solid-State Circuits* 31, 6 (June 1996), 869–874. DOI: <http://dx.doi.org/10.1109/4.509877>
- A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*, Vol. 1. 4.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324. DOI: <http://dx.doi.org/10.1109/5.726791>

- Y. LeCun, K. Kavukcuoglu, and C. Farabet. 2010. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*. IEEE, 253–256.
- U. Lotrič and P. Bulić. 2012. Applicability of approximate multipliers in hardware neural networks. *Neurocomputing* 96 (2012), 57–65.
- N. Mauduit, M. Duranton, J. Gobert, and J. A. Sirat. 1992. Lneuro 1.0: A piece of hardware LEGO for building neural network systems. *IEEE Trans. Neural Networks* 3, 3 (1992), 414–422.
- P. Moerland and E. Fiesler. 1997. Neural network adaptations to hardware implementations. *Handbook of Neural Computation* 1 (1997), 2.
- N. Nedjah and L. de Macedo Mourelle. 2007. Reconfigurable hardware for neural networks: Binary versus stochastic. *Neural Comput. Appl.* 16, 3 (2007), 249–255.
- A. Peled. 1976. On the hardware implementation of digital signal processors. *IEEE Trans. Acoust. Speech Signal Process.* 24, 1 (Feb. 1976), 76–86. DOI: <http://dx.doi.org/10.1109/TASSP.1976.1162774>
- C. H. Sequin and R. D. Clay. 1990. Fault tolerance in artificial neural networks. In *Proceedings of the 1990 International Joint Conference on Neural Networks (IJCNN'90)*. IEEE, 703–708.
- P. Y. Simard, D. Steinkraus, and J. C. Platt. 2003. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the 7th International Conference on Document Analysis and Recognition*. 958–963. DOI: <http://dx.doi.org/10.1109/ICDAR.2003.1227801>
- J. Skaf and S. P. Boyd. 2008. Filter design with low complexity coefficients. *IEEE Trans. Signal Process.* 56, 7 (July 2008), 3162–3169. DOI: <http://dx.doi.org/10.1109/TSP.2008.919386>
- M. Skrbek. 1999. Fast neural network implementation. *Neural Network World* 9, 5 (1999), 375–391.
- T. Szabó, L. Antoni, G. Horváth, and B. Fehér. 2000. A full-parallel digital implementation for pre-trained NNs. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN'00)*, Vol. 2. IEEE, 49–54.
- O. Temam. 2012. A defect-tolerant accelerator for emerging high-performance applications. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 356–367.
- L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus. 2013. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*. 1058–1066.

Received June 2014; revised October 2014; accepted October 2014