NACU: A Non-Linear Arithmetic Unit for Neural Networks

NACU: A Non-Linear Arithmetic Unit for Neural Networks

Guido Baccelli
DET, Politecnico di Torino, Italy
guido.baccelli@studenti.polito.it

Dimitrios Stathis
ELE, KTH, Sweden
stathis@kth.se

Ahmed Hemani
ELE, KTH, Sweden
hemani@kth.se

Maurizio Martina
DET, Politecnico di Torino, Italy
maurizio.martina@polito.it

***Abstract*— Reconfigurable architectures targeting neural networks are an attractive option. They allow multiple neural networks of different types to be hosted on the same hardware, in parallel or sequence. Reconfigurability also grants the ability to morph into different micro-architectures to meet varying power-performance constraints. In this context, the need for a reconfigurable non-linear computational unit has not been widely researched. In this work, we present a formal and comprehensive method to select the optimal fixed-point representation to achieve the highest accuracy against the floating-point implementation benchmark. We also present a novel design of an optimised reconfigurable arithmetic unit for calculating non-linear functions. The unit can be dynamically configured to calculate the sigmoid, hyperbolic tangent, and exponential function using the same underlying hardware. We compare our work with the state-of-the-art and show that our unit can calculate all three functions without loss of accuracy.**

***Keywords*— Sigmoid, Hyperbolic tangent, Neural Networks, approximate, exponential, reconfigurable architecture**

I. Introduction

Today, artificial neural networks (ANNs), as a tool for machine learning, are a mega-trend. However, their power consumption is a challenge for power-constrained embedded systems, and the VLSI design community has been intensely researching efficient implementations that come close to ASICs. Coarse Grain Reconfigurable Architectures customised for *ANNs* provide ASIC comparable efficiency [1, 2] while retaining a degree of flexibility to morph into different *ANN* topologies like CNN or LSTM of varying dimensions. Such design frameworks require a versatile arithmetic unit that can morph into different non-linear functions like *hyperbolic tangent* (*tanh)*, *sigmoid (σ)*, exponential (*e),* etc. which serve as activation functions in different *ANNs*. Besides *ANNs*, these non-linear functions are also extensively used in biologically plausible integrate-and-fire neurons using differential equations and integrations, whose numerical solutions often involve these non-linearities.

Another major trend in research on efficient implementation of *ANNs* is approximate computing and fixed-point arithmetics [3] that try to exploit the robustness of *ANNs* by sacrificing the resolution to gain speed and lower power consumption. In this context, the state-of-the-art lacks a systematic method to do a trade-off between the implementation complexity of these non-linear functions and the accuracy that they can achieve.

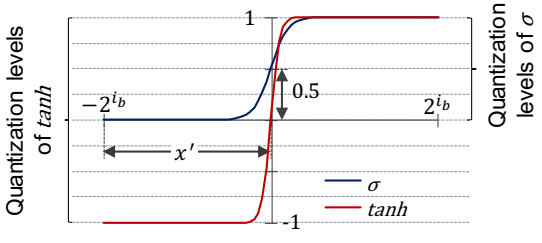

Fig. 1. Sigmoid and hyperbolic tangent function

The key contributions of this paper are solutions to these two needs. a) A common mathematical basis for the calculation of the *σ*, *tanh,* and *e* is used to design a versatile computational unit that can morph between these three functions. b) A formal method of selecting the fixed-point representation that maximises the accuracy of these non-linear functions.

II. Sigmoid and Hyperbolic Tangent Functions

The *σ* and *tanh* are continuous and differentiable functions, and their shape acts as a "softened" approximation of the threshold function. Their unique shape and their non-linearity are what makes them suitable for use in *NNs*. In this section, we mathematically relate the two functions that will then become the basis for a common micro-architectural hardware. Eqs. 1 and 2 define *σ* and *tanh*, respectively. Manipulating them allows us to express *tanh* in terms of *σ,* as shown in Eq. 3, which can be interpreted as *tanh* being a stretched and translated version of *σ*. The scaling factor of 2 is easily implemented in fixed-point representation by an arithmetic left shift.

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad (1) \qquad tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \quad (2)$$

$$tanh(x) = 2\sigma(2x) - 1 \quad (3)$$

The gradient of *tanh* is steeper compared to *σ,* as shown in Fig. 1. Smaller gradient implies that a smaller lookup table (LUT) would be required for *σ* compared to *tanh*. This is because a smaller gradient requires fewer levels of quantisation to achieve the same accuracy, for the same range of input *x*, see Fig 1. This is the reason for modelling *σ* as a LUT and calculating *tanh* values from it.

$$\sigma(-x) = 1 - \sigma(x) \quad (4) \qquad \tanh(-x) = -\tanh(x) \quad (5)$$

The *σ* and *tanh* functions are centrosymmetric, as shown in Eqs. 4 and 5 and depicted in Fig. 1. This property allows a full-function to be realised with a LUT that models only the positive (or the negative) input range of the function. This halves the size of the LUT required for *σ* and allows modelling of the full range of *σ* and *tanh* using Eqs. 3, 4 and 5.

III. Fixed-Point Format

Fixed point representation is the most widely used number representation in embedded systems because of the simpler arithmetic units leading to more energy-efficient designs. The $Q(i_b).(f_b)$ notation is the standard way to specify fixed-point numbers, where i_b represents the integer bits excluding the sign bit, and f_b represents the fractional bits. The total number of bits is $N = 1 + i_b + f_b$; the extra '1' is for the sign bit. The

dynamic range of the input and the desired accuracy decide N and i_b, implying f_b. The objective is to use the smallest N to lower power consumption.

When deciding the fixed-point format for σ, the dimension for input x is driven by the need to cover the desired dynamic range of the function, as shown in Fig. 1. Since the input range exceeds ±1, the i_b for the input variable x must be greater than 1. Whereas, the dynamic range of the output of σ is bounded [0,1], and accuracy is the primary concern which is decided by f_b. Based on these arguments, we next formalise how i_b for the input variable x and f_b for σ can be dimensioned to maximise the accuracy of output and cover the desired input range. Let In_{max} be the largest positive number that the fixed-point format for input needs to represent, Eq. 6 gives the maximum value that the σ can reach.

$$\sigma(In_{max}) = \frac{1}{1 + e^{-In_{max}}}, \quad In_{max} = 2^{i_{b_{in}}} - 2^{-f_{b_{in}}} \tag{6}$$

In_{max} should be large enough that $e^{-In_{max}}$ approaches zero, resulting in σ approaching 1. In_{max} is also inherently related to the accuracy of σ represented by f_{b_out} because any change in σ beyond $\sigma(In_{max})$ should be so small that it is interpreted as zero. This can be formalised as $e^{-In_{max}} < 2^{-f_{b\_out}}$. If this condition is satisfied, the value of σ will saturate to 1. For any value beyond $In_{max,}$ the change in σ will be so small that it will be interpreted as zero change, thus satisfying the desired accuracy. This condition formalises the relationship between the desired input range and the resolution or accuracy of the output in Eq. 7.

$$\begin{aligned}
& e^{-In_{max}} < 2^{-f_{b_out}} \\
& \Longrightarrow -In_{max} < (ln\,(2)) \cdot \left(-f_{b_out}\right) \\
& \Longrightarrow \left(2^{i_{b_{in}}} - 2^{-f_{b_{in}}}\right) > ln\,(2) \cdot f_{b_out} \\
& because\ -f_b = i_b - N + 1 \\
& \Longrightarrow 2^{i_{b_{in}}}(1 - 2^{1-N_{in}}) > ln(2) \cdot f_{b_out} \\
& because\ f_b = N - i_b - 1 \\
& \Longrightarrow 2^{i_{b_{in}}} > ln(2) \cdot \frac{\left(N_{out} - i_{b_out} - 1\right)}{(1 - 2^{(1-N_{in})})}
\end{aligned} \tag{7}$$

Only fixed-point formats that satisfy Eq. 7 can fulfil the desired input range and output accuracy. The equation cannot be expressed in closed form, so it has to be solved case by case. This result provides a lower bound for input and output integer bits. In most cases it's beneficial to have the same input and output format and implies $i_{b\_in}=i_{b\_out}=i_b$, $f_{b\_in}=f_{b\_out}=f_b$ and $N_{in}=N_{out}=N$. The final choice on fixed-point format must be based on the general features of the target application, such as the maximum value a number can reach within the application and the desired precision. Here we present the limits of the σ fixed-point implementation for the general case, which will allow us to represent the full range of the output with the best accuracy for a given bit-width.

Consider a case of 16-bit fixed-point number for both the input and the output. Using Eq. 7, we can calculate that to represent the full input range of σ, i_b needs a minimum of 4 bits, and the remaining 11 bits can be allocated as fractional bits to maximise the accuracy.

IV. Function Models

In the previous sections, we discussed the basic properties of the two functions and presented a systematic method to find the bounds on the fixed-point representation. In this section, we will see how we can use these properties as the basis for an efficient hardware model of a reconfigurable arithmetic unit to compute σ, *tanh*, and e functions. Where σ is used as the elementary function, and other functions are derived from it. We first express σ and *tanh* in terms of a unified equation with a set of coefficients deciding if it is σ or *tanh* that will be computed. Next, we show how e and by extension the *softmax* can be derived from the σ. We also show how the error propagation between σ and e can be modelled and also bounded, given the max input to e.

A. A common mathematical basis for sigmoid and tanh

Most of the state-of-the-art implementations of σ use polynomial approximation on a sub-portion of the input range, see section VI. Eq. 8 presents such a polynomial approximation with P representing the degree of the polynomial, m the coefficients for the non-zero powers of x, and q the constant bias. Eq. 8 can only be used for a positive input range of the σ. Using the symmetric property of σ, Eq. 8 can be tweaked for the negative range. Eq. 9 uses the same coefficients as Eq. 8, and models the negative input range.

$$\sigma(x) \approx \sum_{i=1}^{P} m_i x^i + q\ , \quad \mathrm{x} \geq 0 \tag{8}$$

$$\sigma(\mathrm{x}) \approx \sum_{i=1}^{P} -m_i x^i + (1 - q), \quad \mathrm{x} < 0 \tag{9}$$

The *tanh* function can be computed by substituting Eqs. 8 and 9 in Eq. 3 to get Eqs 10 and 11. Notice that the two sets of equations have the same form, and the coefficients and bias constants differ only in their scaling factors. This forms the basis for the reconfigurable NACU.

$$\tanh(x) \approx \sum_{i=1}^{P} 2^{i+1} m_i x^i + (2q - 1), \quad \mathrm{x} \geq 0 \tag{10}$$

$$\tanh(\mathrm{x}) \approx \sum_{i=1}^{P} -2^{i+1} m_i x^i + (1 - 2q), \quad \mathrm{x} < 0 \tag{11}$$

B. Exponential Function for Softmax

Most DNNs classify the input in the last layer based on the softmax function. Softmax is a vector-valued function that takes a collection of inputs and normalises them into a probability distribution. Given a vector of N inputs $X = [x_1, x_2, \ldots, x_N]$ the softmax will output a vector of probabilities $SM = [sm_1, sm_2, \ldots, sm_N]$. Each probability $\mathrm{sm}_i \in SM$ is calculated according to Eq. 12, with $x_i \in X$.

$$sm_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{12}$$

$$sm_i = \frac{e^{(x_i - x_{max})}}{\sum_{j=1}^{N} e^{(x_j - x_{max})}} \tag{13}$$

Softmax behaves like a "*soft*" maximum function, where the highest input tends to be 1 while all others get flattened out to 0. The *softmax* function, given in Eq. 12, suffers from numerical instability due to the saturation of exponentials. If more than one result saturates to the maximum value, multiple classes are simultaneously associated with the same input, invalidating the classification purpose of *softmax*. By normalising the numerator and denominator of the function to $e^{x_{max}}$, the saturation problem is avoided, as shown in Eq. 13.

The benefit of Eq. 13 is that the maximum value of the exponential will be equal to $e^0 = 1$. This reduces the dynamic range of the e^x function to [0, 1].

Using Eq. 1 as a starting point, we can derive the Eq. 14 for *e*. Eq. 15 presents the propagation of uncertainty from σ to *e*, as described in [17]. In this paper, we refer to this as error propagation. Let δe and $\delta\sigma$ be the uncertainty/error in *e* and σ, respectively. The error propagation coefficient $[1/(1-\sigma)^2]$ diverges, i.e., tends to infinity, as σ saturates to 1.

$$e^x = \frac{1}{\sigma(-x)} - 1 = \frac{1}{1-\sigma(x)} - 1 \qquad (14)$$

$$\delta e = \left|\frac{\partial e}{\partial \sigma}\right| \delta\sigma = \frac{1}{(1-\sigma)^2}\delta\sigma \qquad (15)$$

Assuming that the range of input to *e* is known and if *e* is normalised to the maximum input value as shown in Eq. 13, the error propagation from σ to *e* can be bounded. The normalised input takes the form *of* $x' = x - x_{max}$, and the range of such inputs is $x' \in [-2^{i_b}, 0]$, consequently $\sigma(x - x_{max}) \in [0,0.5]$, as shown in Fig. 1. Using the above, we can bound the σ_{max} to 0.5, and this bounds the error coefficient to 4, as shown in Eq. 16 and has accuracy comparable to σ.

$$\left|\frac{\partial e}{\partial \sigma}\right|_{max} = \frac{1}{(1-\sigma_{max})^2} = \frac{1}{(1-0.5)^2} = 4 \qquad (16)$$

This method for bounding the error is predicated on a known range of input *x*. This condition is always fulfilled while computing *softmax* function in *NNs* using fixed-point representation. Using these arguments as the basis, we next elaborate the micro-architecture of the reconfigurable arithmetic unit that can compute σ, *tanh*, *e*, and *softmax.*

V. Hardware Architecture

The micro-architecture of the morphable Non-linear Arithmetic Computation Unit (NACU)[1] is shown in Fig. 2. The design has two main parts. The first part computes the coefficient and the bias in Eqs. 8-11. The second part implements the Eqs. 8-11 to compute σ and *tanh* and Eqs. 13 and 14 to compute *e* and *softmax* functions. The next two sub-sections elaborate these two parts of NACU.

A. Sigmoid and Tanh coefficient and bias calculation

The first step is obtaining coefficients and bias for the positive range of the σ as per Eq. 8. In the implementation reported in this paper, this is realised as a piecewise linear approximation (*PWL*) using 1st order polynomial. The coefficient m_1 and the bias *q* for the σ are stored in a *LUT*. Each element of LUT provides m_1 and *q* for each linear segment in the PWL model. The remaining micro-architecture is agnostic to how m_1 and *q* for σ are calculated. The coefficient and bias for the positive range of σ are then used to calculate the coefficients and biases for a) negative range of σ, b) positive, and c) negative ranges of *tanh*. These three computations require only simple shift and twos complement operations that we explain next.

The range of *q* (Eqs. 8-11) for all three cases is in the interval $[\sigma(0),1]=[0.5,1]$. This range is based on the centrosymmetric nature of the functions, as argued in section II. Besides the limited range, the operations that act on *q* are also restricted to $1-2q$, $2q-1$ and $1-q$. Based on these facts,

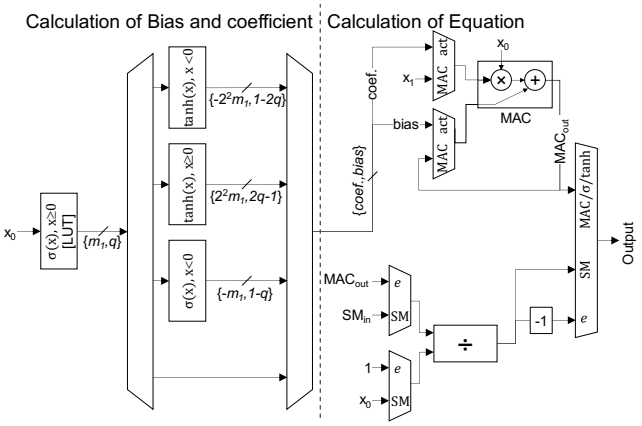

Fig. 2. Hardware architecture of the arithmetic unit

we can use the properties of fixed-point numbers to design specialised units that are simpler than the general adders/subtractors, Fig. 3. We next elaborate on the design of the simplified circuitry to compute coefficients and biases for the three cases. We adopt a common convention for all three units and denote the output of the three units as *r* and the bits of the input as a_j that can be *q* or 2*q*.

Coefficient and Bias for the negative range of σ: The interval $q \in [0.5,1]$ is split into two ranges: $q \in [0.5, 1)$ and $q=1$ to compute the result $r = 1-q$. For the first range, $1-q$ will result in integer bits being zero and the fractional bits being negated. The negation is implemented by taking 2's complement, as shown in Fig. 3a. For the second range ($q=1$), both the integer and the fractional bits are zero, and the implementation shown in Fig. 3a is still valid.

Coefficient and Bias for the positive range of tanh: We again split the range $2q \in [1, 2]$ into two intervals: $2q \in [1, 2)$ and $2q = 2$ to compute the result $r = 2q-1$. For both intervals, since we subtract integer 1, the fractional bits are unaffected and passed on as it is, as shown in Fig. 3b. The integer bits for the first interval $2q \in [1, 2)$ are $a_1a_0=01$, and when 1 is subtracted they become $a_1a_0=00$. For the second interval $2q=2$, the integer bits are $a_1a_0=10$, subtracting 1 makes them $a_1a_0=01$. We can avoid subtraction by propagating a_1 as a_0 to cover both intervals, as shown in Fig. 3b.

Coefficient and Bias for a negative range of tanh: Once again, we split the range $-2q \in [-2,-1]$ into two intervals: $2q \in [-2, -1)$ and $2q=1$ to compute the result $r = 1-2q$. As argued in the previous case, the fractional bits remain unaffected, as shown in Fig. 3c. When $2q$ is in the first interval $[-2, -1)$, the integer bits of *r* will be $1-2 = -1$, i.e., all bits will be 1. In the second interval, where $2q=1$, the integer bits of *r* will be all 0 since $1-1=0$. We can avoid subtraction with an observation that for the first interval, $a_0=0$ and the second interval $a_0=1$.

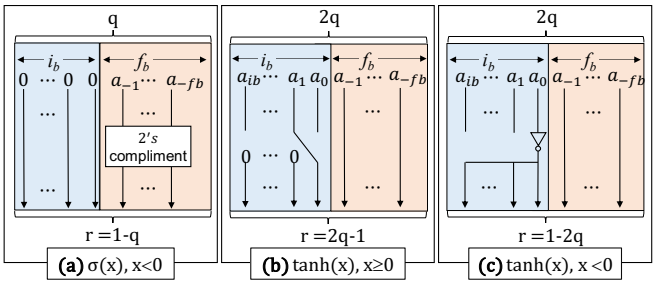

Fig. 3. Proposed design for replacing subtractors for calculation of bias

[1] The RTL HDL design of NACU, test-bench, reference model in Matlab can be found on a publicly available repository https://github.com/silagokth/NACU

This allows us to feed the inversion of a_0 to all integer bits of r to cover both intervals as shown in Fig. 3c.

B. Sigmoid, Tanh, Exponential and Softmax Computation

Once the coefficients and biases for σ and *tanh* are computed as explained above, the computation of σ and *tanh* are straightforward using a multiply and add unit, as shown in the top-right corner of Fig. 2. The multiply and add unit is also shown to have a feedback path to accumulate the sum to function as a typical *MAC* unit. This *MAC* unit serves two purposes. The first is to accumulate a convolution sum that is common in *ANNs* before the non-linearity is applied. The second is to compute the normalisation factor for *softmax* function, the denominator in Eq. 13.

To compute e^x, we implement Eq.14; we first compute $\sigma(-x)$, as explained above. The result is then fed to a divider followed by a decrementor, as shown in Fig. 2. The decrementor can be optimised, as explained next. Due to the normalisation of the first term, the range of σ' of Eq. 14 is bounded: $\sigma' = \frac{1}{\sigma(x_{max}-x)} \in [1,2]$. The interval of the term $\sigma'$is the same as the that of $2q$ in subsection V.A, so $\sigma' - 1$ can be implemented using the circuit in Fig. 3c.

Finally, to compute the *softmax* function, we need to first compute the common normalising factor, the denominator in Eq 13, using the circuitry described above. Next, for each input, the $e(x_i - x_{max})$ is computed and scaled with the normalising factor to compute the *softmax* function.

VI. Related Work

This section presents an overview of the most widely used architectural alternatives for the σ, *tanh*, and e and by implication also *softmax*. We first categorise the landscape of architectural alternatives in generic dimensions and compare them qualitatively in a normalised manner. This is followed by reviewing specific related work by different research groups and NACU and classifying them into one of the discussed categories. We defer the quantitative results of NACU and a best-effort comparison with the related work to section VII.

There are three broad categories of implementation alternatives to compute the non-linear functions. In all three categories, the range of the function is uniformly or non-uniformly divided into segments. Each segment approximates the function of the range it represents. The approximation can be a constant, a straight line, or a higher-order polynomial. All other things being equal, non-uniform segments can achieve better accuracy because, by definition, non-linear functions have non-uniform gradients, and we can better approximate the function by having smaller segments in regions with a high degree of non-uniformity and vice-versa.

All three alternatives can be implemented by look-up tables (*LUTs*) for uniform segments and *RALUTs* (Range Addressable *LUTs*) for non-uniform segments. Depending on how the segment is approximated, each entry in *LUT/RALUT* would be a constant or a list of polynomial coefficients. When segments are approximated by a straight line, the alternative is commonly called piecewise linear model (*PWL*), and when the segment is non-uniform, the alternative is called *NUPWL*. There is no widely accepted acronym when segments are approximated by higher-order functions.

The four alternatives discussed above - *LUT*/*RALUT* or *PWL*/*NUPWL* – lend themselves well for implementing σ and *tanh* because they have bounded range [0, 1]. e, on the other hand, can be derived from σ as we proposed in subsection IV.B, or it can be approximated by a higher-order polynomial where the entire desired range is treated as a single segment.

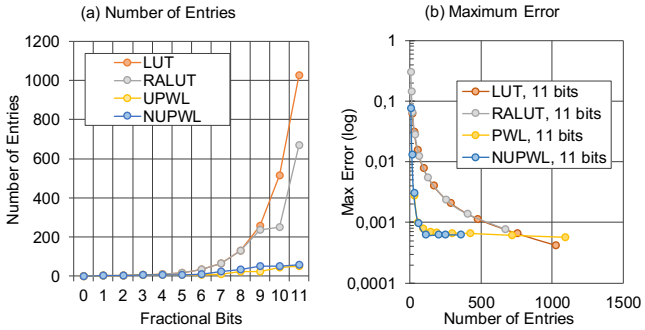

Fig. 4. Graph presenting (a) LUT entries depending on fractional bits and (b) Maximum error depending on number of entries for 11 fractional bits

We next analyse the trade-offs between the four most widely used alternatives for implementing σ and *tanh* – *LUT/RALUT, PWL, NUPWL*. A common rule that applies to all four alternatives is that accuracy is improved at the expense of increasing their implementation cost: more fractional bits and more entries. Fig. 4a compares the four alternatives and shows how the implementation cost increases to achieve the same level of accuracy, e.g., with 10 fractional bits, *PWL/NUPWL* alternative achieves the same level of accuracy with just ~50 entries compared to 668 and 1026 entries for *RALUT* and *LUT* alternatives. Fig. 4b presents the scaling of the maximum error depending on the number of entries. It can be seen that PWL and NUPWL have better scaling than LUT\RALUT, and the error improvement flattens out after a certain point. Even if the NUPWL allows for better error with fewer entries, the improvement is minimal since it occurs after the knee of the curve. For the implementations presented in Fig. 4, all possible interval sizes, ranges and fixed-point formats were explored, and the one with the best accuracy was selected.

Based on the above primer on the landscape of implementation alternatives and their trade-offs, we review the reported related work. A RALUT implementation of *tanh* is reported in [4]. This work divides the input into three regions, a pass region where $tanh(x) \approx x$, an elaboration region where the input is covered by a *RALUT,* and a saturation region where the output is constant. This leads to a reduced number of entries for the *RALUT*. The work also provides an analysis of the interval boundaries and number of *LUT* entries needed to achieve maximum accuracy. [5] gives an overview of such a table-based implementation of the *tanh*.

An implementation of σ is presented in [6] using a 4-interval *NUPWL*. In contrast, [7] reports a *PWL* implementation where they used a recursive approach to progressively refine and dimension the number of segments to achieve the desired level of accuracy. In [8], a hybrid approach is proposed, where a *PWL* gives a coarse approximation, and then a *RALUT* refines the *tanh* curve.

Using polynomials of higher degree can provide sufficient accuracy, without the need for partitioning the input. Works reported in [6, 9] use parabolic curves to model the σ and *tanh* functions, with reasonable accuracy up to the saturation region. We note that all the works mentioned above use coefficients that are powers of two, to enable using shift operations for multiplication. A 2nd order Taylor series based implementation is reported in [10]. For it to achieve the same level of accuracy, the input needs to be partitioned in multiple intervals. A very different approach is taken by [11], where

TABLE I. RELATED WORK

	[6]	[6]	[6]	[10]	[10]	[11]	[4]	[5]	[8]	[13]	[14]	[14]	NACU
Implem.	NUPWL	2nd order Taylor	2nd order Taylor opt	1st order Taylor	2nd order Taylor	Based on e^x	RALUT	RALUT	PWL & RALUT	6th order Taylor	CORDIC	Parabolic	PWL
Area [µm²]	Not applicable	Not applicable	Not applicable	Not reported	Not reported	Not applicable	1280.66	11871.53	5130.78	20700	19150	26400	9671
Logic Elem.	246/ 15408	368/ 15408	167/15408	Not reported	Not reported	287, 372/ 16416	Not applicable	Not applicable	Not applicable	Not applicable	1837/ 18752	481/ 18752	Not applicable
Tech. Node [nm]	65	65	65	40	40	90	180	180	180	65	65 ASIC, 90 FPGA	65 ASIC, 90 FPGA	28
LUT entries	7	4	4	102[a]	28[a]	Not applicable	14	127	Not reported	Not applicable	Not applicable	Not applicable	53
Nr. of Bits	16	16	16	16	16	6 to 14	9 in, 6 out	10	10	18[b]	21	18	16
Clock Period [ns]	10	10	10	2.677	2.677	2.605, 2.294	2.12	2.12	2.8	40.3	86, 140.74	20.8, 70.41	3.75
Latency [Cycles]	2	2	3	4	7	4, 5	1	1	1	1	1	1	3, 3, 8
Functions	σ	σ	σ	σ	σ	σ, *tanh*	*tanh*	*tanh*	*tanh*	*e*	*e*	*e*	σ, *tanh*, *e*, *softmax*

[a] Number of intervals, the authors do not report the LUT entries [b] Fractional bits

they compute σ in terms of e as defined in Eq. 1. This work implements e based on [12] and increments it by 1. The work also implements *tanh* in terms of σ as defined in as Eq. 3.

Implementation of e is considered a more significant challenge as it requires higher order polynomials to achieve sufficient accuracy. The work in [13] makes use of a 6th order Taylor expansion to describe the whole exponential curve, compared to the 2nd order used for σ or *tanh*. A parabolic synthesis approach is presented in [14] that obtains the e by the multiplication of several parabolic curves. Also, [14] introduces and compares with a CORDIC based implementation that is also used in [15] to implement e. A different approach is taken in [12], by exploiting the exponential change of base to express e as a power of 2. It splits the calculation between the fractional and integer bits. The fractional part is approximated as the line $1+x$, and the 2nd power of the integer part is implemented using bit shifts.

VII. RESULTS AND COMPARISON

In this section, we report the implementation results of NACU and make a best-effort comparison with the state-of-the-art. We mainly compare accuracy, both max and average error. Where possible, we also compare the area. NACU has been implemented as an ASIC macro in 28 nm node and can operate at 267 MHz. Fig. 5 presents the area break down, power consumption, and latency for different functions. All results are based on post-layout data, including simulation for power numbers, with all the parasitic and actual wire delay and capacitances of the entire design.

The area of NACU is dominated by a pipelined divider. It is possible to reduce the area by adopting a sequential divider, as reported in [11]. We note that the cost of the pipelined divider is justified since it is shared between the e and *softmax* functions and gives us a higher throughput. The coefficient calculation area includes the *LUTs* for the *PWL* approximation and the dedicated units, as described in section V. The rest of the units are shared between the σ, *tanh*, e, and MAC

	Power [mW]	Latency [cycles]
σ	2.159	3
tanh	1.95	3
e	3.74	8

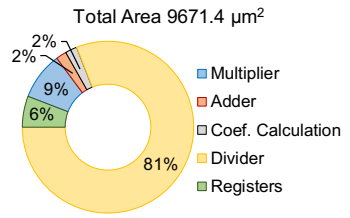

Fig. 5. Experimental results and area breakdown of NACU

functions. We note here that the area of the coefficient and bias calculation is comparable to that of the adder, as illustrated in Fig. 5. Adopting dedicated *LUTs* for the *tanh* or even using a generic adder to derive slope and bias from the σ LUTs would have nearly doubled the area.

We compare NACU with the state-of-the-art in three dimensions. The first dimension is its reconfigurability. NACU is designed to be used as part of coarse grain reconfigurable architectures (CGRAs) that can be dynamically configured for any mix of *ANNs* and *SNNs* (spiking neural networks) in the same fabric instance. Such a CGRA needs these varieties of non-linearity available in the same unit. Table I shows that this has not been the objective of the reported implementations. The other two dimensions are related to the accuracy and the implementation costs.

The results of comparing NACU's accuracy with those of the related work are plotted in Fig. 6. All comparisons are normalised to the 16-bit NACU. The left column shows the max error for σ, *tanh*, and e. The right column shows the average error for σ and *tanh,* where it is available. For e, none of the related work reports average error. In Fig. 6c, d, and e we also report the accuracy results of the NACU using different bit-widths. The selected bit-widths match the ones used in the related work and allow for better comparison. Table I summarises the implementation costs for the reported related-work, against whom we compare. The reported metrics in Table I are not scaled to NACU's technology, but they are reported as in the original work. Area comparison with the σ and *tanh* implementation is difficult, due to very old technology used [4,5,8] and FPGA implementations [6,11].

A. Comparison of the σ function

The *NUPWL* approximation of [6] avoids multipliers using power of two shifts and for this reason, has 10X worse max error compared to NACU. The same group also reports two implementations of 2nd order Taylor expansions in [6]. However, the use of a multiplier in the Taylor series does not result in any accuracy improvement. The work reported in [10] splits σ into 102 segments to achieve 10X better accuracy compared to NACU. The large number of segments implies large LUTs, but these are not reported in the paper. A 2nd order Taylor series implementation with fewer segments is also reported in [10]. This gives comparable accuracy but, as expected, with increased latency – 7 cycles as opposed to 4 cycles for the 1st order. The average error for [6] is comparable to its max error, [10] does not report average error. The σ implementation reported in [11] is based on first

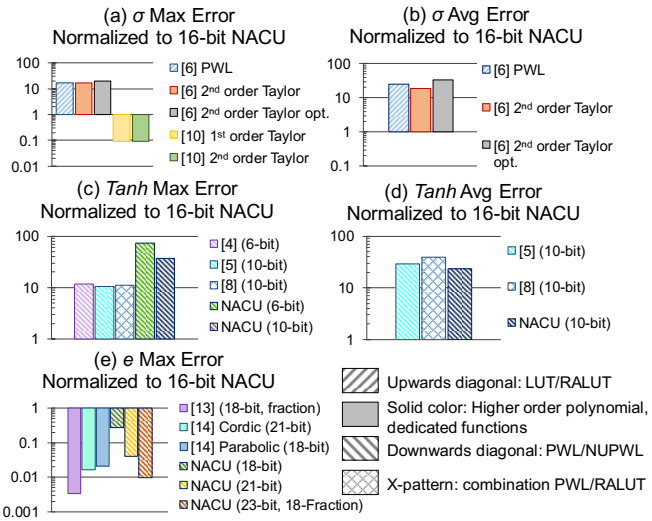

Fig. 6. Error plots comparing with state-of-the-art, (a)-(c) Maximum error, (d) & (e) Average error normalised to the proposed architecture (lower is better)

implementing *e* and using Eqs. 1. We argue that NACU is more efficient because it relies on division, only for the *e*/*softmax* function, which is only required in the last layer. Whereas [11] would need division in all layers for the *σ* /*tanh* activation functions. The work in [11] reports RMSE of 9.1×10^{-3} with 0.998 correlation for the *σ*. In comparison, NACU achieves 2.07×10^{-4} RMSE with 0.999 correlation.

B. Comparison of the tanh function

The *tanh* implementation reported in [11] is also based on first implementing *e* and calculating tanh base on Eq. 3. The design in [11] reports RMSE of 1.77×10^{-2} with 0.999 correlation for the *tanh* function., compared to NACU's 2.09×10^{-4} RMSE and 0.999 correlation. The other solutions for *tanh* reported in [4,5,8] are all based on *RALUT*. Comparing *tanh LUTs* to our work leads again to a trade-off between area and accuracy, the lower error of *PWL* in *NACU* provides 10X accuracy but comes at the expense of higher bit-width and an extra multiplier and an adder. As noted by [10], the multiply and add unit can also be used as a MAC for computing the linear computation as well. For this reason, we claim that a *PWL* approach for non-linear activators is the most natural fit for *ANN* hardware implementation.

C. Comparison of the e function

The max error comparison of *e* shows that NACU is 10X worse. This is explained by the fact that [13,14] uses 18 to 21 bits as opposed to NACU's 16 bits. NACU implementations that use larger bit-widths can reach accuracies closer to the related work. The use of higher-order function allows for better accuracy but increases the complexity of the design. The area of [14] scaled to 28nm node is ~5800µm^2 for the smallest CORDIC implementation. In comparison, NACU takes ~9600 µm^2 but implements not just *e* but also *σ*, *tanh*, and *softmax*. Further, note that [14] is a sequential design that would take 42 ns scaled to 28 nm node using data from [16]. In contrast, NACU, with its pipelined design, takes 90 ns for filling the pipeline and 3.75ns for computing each consecutive *e*. For comparison, the 6th order Taylor implementation [13] scaled down to 28nm [16] will have an area of ~6200µm^2 and period of 20ns, and the parabolic implementation [14] will have an area of ~8000µm^2 and a period of 10ns.

The *e* design in [12] looks promising due to its potentially low hardware cost and reasonable precision. However, we cannot quantify area comparison as none is reported. Overall, NACU provides a good balance between versatility, accuracy, area, and latency compared to the related work.

VIII. Conclusion

In this work, we presented a common mathematical basis for implementing the three most common non-linear functions of *NNs*, *σ, tanh,* and *e*. We also presented a formal method to find the best fixed-point representation. We propose a novel architecture that can be dynamically configured to calculate the three different functions together with the softmax function and the MAC operation. The numerical properties of the functions are exploited to derive a more efficient architecture. The design provides accuracy comparable with the state-of-the-art and is proven to be more cost-efficient than the other implementations. The proposed architecture's versatility targets reconfigurable neural network architectures to fulfil their diverse non-linear activation needs. In the future, we plan to optimise out the conventional divider with an approximate one. This will allow us to significantly lower the area cost with a small reduction in overall accuracy.

Acknowledgement

This work was supported by Vinnova as part of the Chrest II project.

References

[1] Y. Chen, et al., "DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning," In Commun. ACM, vol. 59, pp. 105–112, 2016.

[2] A. Majumdar, et al., "A Massively Parallel, Energy Efficient Programmable Accelerator for Learning and Classification," In ACM Trans. Archit. Code Optim., vol. 9, pp. 1–30, 2012.

[3] S. Hashemi, et al., "Understanding the impact of precision quantization on the accuracy and energy of neural networks," In DATE, 2017.

[4] B. Zamanlooy, et al., "Efficient VLSI Implementation of Neural Networks With Hyperbolic Tangent Activation Function," In IEEE Trans. VLSI, vol. 22, pp. 39-48, Jan. 2014.

[5] K. Leboeuf, et al., "High Speed VLSI Implementation of the Hyperbolic Tangent Sigmoid Function," In ICCIT, 2008.

[6] I. Tsmots, et al., "Hardware Implementation of Sigmoid Activation Functions using FPGA," In CADSM, 2019.

[7] K. Basterretxea, et al., "An Experimental Study on Nonlinear Function Computation for Neural/Fuzzy Hardware Design," In IEEE Trans. Neural Networks, vol. 18, pp. 266-283, Jan. 2007.

[8] A. H. Namin, et al., "Efficient hardware implementation of the hyperbolic tangent sigmoid function," In ISCAS, 2009.

[9] V. P. Nambiar, et al., "Hardware implementation of evolvable block-based neural networks utilizing a cost efficient sigmoid-like activation function," In Neurocomputing, vol. 140, pp 228-241, 2014.

[10] R. Finker, et al., "Controlled accuracy approximation of sigmoid function for efficient FPGA-based implementation of artificial neurons," In Electron. Lett., vol. 49, pp. 1598–1600, 2013.

[11] S. Gomar, et al., "Precise digital implementations of hyperbolic tanh and sigmoid function," In ACSSC, 2017.

[12] S. Gomar, et al., "Digital multiplierless implementation of biological adaptive-exponential neuron model," In IEEE Trans. Circuits Syst. , vol. 61, pp. 1206–1219, 2014.

[13] P. Nilsson, et al., "Hardware implementation of the exponential function using Taylor series," In NORCHIP 2014

[14] P. Pouyan, et al., "A VLSI implementation of logarithmic and exponential functions using a novel parabolic synthesis methodology compared to the CORDIC algorithm," In ECCTD, 2011.

[15] M. Heidarpour, et al., "A CORDIC Based Digital Hardware For Adaptive Exponential Integrate and Fire Neuron," In IEEE Trans. Circuits Syst., vol. 63, Nov. 2016.

[16] A. Stillmaker, et al., "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," In Integration, vol. 58, pp. 74–81, Jun. 2017.

[17] S. V. Gupta, "Propagation of Uncertainty," In Measurement Uncertainties, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 109–129, 2012.