ASIC Design of Nanoscale Artificial Neural Networks for Inference/Training by Floating-Point Arithmetic

Farzad Niknia, *Student Member, IEEE*, Ziheng Wang*, Student Member, IEEE*, Shanshan Liu, *Member, IEEE*, Pedro Reviriego, *Senior Member, IEEE*, Ahmed Louri, *Fellow, IEEE* and Fabrizio Lombardi, *Life Fellow, IEEE*

Abstract— **Inference and on-chip training of Artificial Neural Networks (ANNs) are challenging computational processes for large datasets; hardware implementations are needed to accelerate this computation, while meeting metrics such as operating frequency, power dissipation and accuracy. In this paper, a high-performance ASIC-based design is proposed to implement both forward and backward propagations of multi-layer perceptrons (MLPs) at the nanoscales. To attain a higher accuracy, floating-point arithmetic units for a multiply-and-accumulate (MAC) array are employed in the proposed design; moreover, a hybrid implementation scheme is utilized to achieve flexibility (for networks of different size) and comprehensively low hardware overhead. The proposed design is fully pipelined, and its performance is independent of network size, except for the number of cycles and latency. The efficiency of the proposed nanoscale MLP-based design for inference (as taking place over multiple steps) and training (due to the complex processing in backward propagation by eliminating many redundant calculations) is analyzed. Moreover, the impact of different floating-point precision formats on the final accuracy and hardware metrics under the same design constraints is studied. A comparative evaluation of the proposed MLP design for different datasets and floating-point precision formats is provided. Results show that compared to current schemes found in the technical literatures, the proposed design has the best operating frequency and accuracy with still good latency and energy dissipation.**

Index Terms—**Artificial neural network (ANN), multilayer perceptron (MLP), floating-point, ASIC design, inference, training.**

Manuscript received January 19, 2023, and revised August 17 and December 3, 2023 and accepted February 18, 2024. The research was supported by the FUN4DATE (PID2022-136684OB-C21/C22) project funded by the Spanish Agencia Estatal de Investigacion (AEI) 10.13039/501100011033, and by NSF under Grant CCF-1953961 and Grant 1812467. *Corresponding author: Shanshan Liu* (email: ssliu@uestc.edu.cn).

Ziheng Wang, Farzad Niknia and Fabrizio Lombardi are with Department of Electrical and Computer Engineering, Northeastern University, MA 02115, USA.

Shanshan Liu is with School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu, 611731, China.

Pedro Reviriego is with Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain.

Ahmed Louri is with Department of Electrical and Computer Engineering, the George Washington University, Washington, DC 20052, USA.

I. Introduction

Machine learning (ML) has been widely utilized to model complex problems and discover and/or predict patterns in applications such as speech/handwritten recognition, face recognition, natural language processing and tracking renewable energy [1]-[3]. Often ML relies on so-called Artificial Neural Networks (ANNs) [4] to deal with complex computation for Artificial Intelligence (AI), so a hardware implementation inspired by the human brain and the operation of its biological neurons [5]. For solving complex nonlinear problems, an ANN has been shown to yield excellent performance compared to conventional methods, such as classification and regression [1]; also, the capability to parallelize ANN computation in hardware makes ML processing fast and efficient [6].

ANNs consist of artificial neurons that are distributed among multiple layers; each neuron in a layer is only connected to the neurons in the previous and next layers. The fundamental arithmetic operation of these networks is the sum of the weighted products of the neuron values from the previous layer. The large number of computations for a neuron is processed by a multiplication-and-accumulation (MAC) unit [7], [8]. One of the widely used types of ANNs is the so-called multi-layer perceptron (MLP), which has been extensively used in deep learning [9]. As all neurons in a layer of an MLP are connected to the neurons in the next layer through synapses, then the number of MAC units increases dramatically when implementing larger networks, which makes the accelerator design challenging for many applications [10].

Different hardware platforms have been proposed for implementing ANNs (as accelerators) such as GPU, CPU, FPGA, and ASIC [11], [12] Platforms such as GPUs have the capability of parallelizing complex computations to improve performance; however, the parallelization may incur in many issues such as large hardware overhead and specifically the dissipation of significant power, so limiting its application. To overcome this issue, FPGAs that are reconfigurable and programmable based on system parameters can be utilized; even though unlike GPUs, their ability to parallelize complex computational processes at high performance is very limited, they have better power characteristics. However, an FPGA-based implementation poses the issue of low speed and cannot meet the demand for platforms with even lower power dissipation. Therefore, the use of ASICs has become a very efficient solution to address these issues. Even though ASIC is not reconfigurable as FPGAs, the higher speed and low power dissipation make ASIC an excellent candidate to implement ANNs for advanced nanoscale devices and high-performance applications [12], [13].

In this paper, an efficient ASIC-based ANN design is proposed (specifically for MLPs) to implement on-chip training and inference; this is achieved relying on fully pipelined floating-point (FP) MAC units, in which the nanoscale FP multiplication and addition blocks are designed and arranged by taking the requirements of the ANN into account. The main contributions of this paper are as follows:

- A fully-pipelined MAC unit design is proposed based on our previous work of [21]; it additionally supports single addition or multiplication operations, so it can save several clock cycles for training that usually requires such computation.
- Based on the redesigned MAC units, a hybrid computational scheme is proposed to implement the forward and backward propagations of an MLP; this results in a flexible implementation that is independent of NN size.
- To mitigate the redundant gradient computations during training, an efficient computational process is proposed to implement backward propagation by dividing the often-complex calculations into several steps; this scheme reduces the entire computational latency of training.
- The implementation of MLPs with different floating-point precisions (i.e., half, single, and double formats) is investigated; hardware metrics at nano-scale (32 nm) and accuracy are evaluated. Compared to existing MLP designs found in the technical literature, the proposed design achieves the best operating frequency and classification accuracy with a reasonable latency and energy dissipation.

The rest of this paper is organized as follows. Section II briefly reviews the MLP and its principles for both forward and backward propagations. Moreover, this section also elaborates different FP precision formats for MLP computations. Section III presents the proposed hybrid computational scheme and implementation details of MLPs. Section IV fully evaluates the proposed MLP implementation with different FP precision formats; moreover, a comparison with other MLP schemes found in technical literatures is also provided to show the efficiency of the proposed hardware design. Finally, the paper ends with the conclusion in Section V.

II. Preliminaries

A. Multilayer Perceptron (MLP)

MLP and its training (using the gradient descent algorithm) have been widely utilized in deep learning models. MLPs are a variant of the original perceptron model introduced by Rosenblatt [14], and they can model highly nonlinear functions while being trained by new and unseen data. As depicted in Fig. 1 (a), an MLP network consists of some interconnected layers, so modeling a non-linear transfer of an input vector to an output vector [15]. Generally, MLPs have an input layer, at least a hidden layer and an output layer. The number of neurons in the input layer is determined by the size of valid sample features, while the number of classes determines the number of neurons in the output layer. A layer consists of multiple neurons, each computing a non-linear function of the sum of the weighted neuron values from the previous layer (as shown in Fig. 1 (b)). More specifically, during inference of a fully connected network, the value of a neuron is calculated as Eqn. 1 , where n_k^{i+1} denotes the value of neuron k in layer i+1, $w_{j,k}^{i+1}$ is the related weight of neuron j in layer i mapping to neuron k in layer i+1, and m is the number of neurons in layer i. Also, b_k^{i+1} is the bias value related to n_k^{i+1} and Φ is the activation function (e.g., the widely-used ReLU [16], which is also considered in this paper).

$$n_k^{i+1} = \Phi\left(\sum_{j=1}^{m} w_{j,k}^{i+1} \cdot n_j^i + b_k^{i+1}\right) \tag{1}$$

As shown in Fig. 1 (c), the forward propagation (referred to as inference hereafter) of an MLP calculates the neuron values for the output layer based on the current values of the input features and related weights, while backward propagation compares the values of the neurons in the output layer and the related label of the input feature to calculate the error vector (denoted by e in Fig. 1 (c)); then this error propagates through the input layer by adjusting the weights in each layer accordingly.

Based on Fig. 1 (c), initially, the blue paths (representing the inference process of an MLP) calculate the value of each neuron by propagating data from the input layer to the output layers. The value of the neurons is depicted in black; then by assigning the error using a proper error loss, a similar process begins from

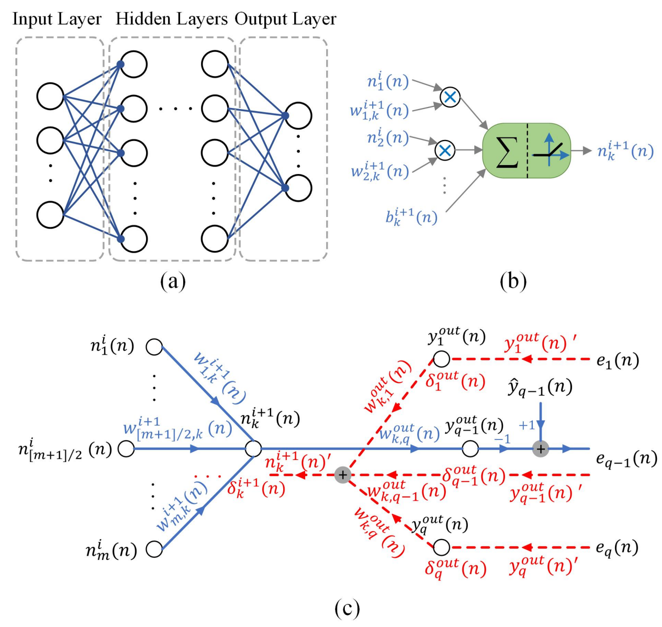


Fig. 1. An overview of MLP: (a) network; (b) computation of single neuron; (c) forward (blue) and backward (red) propagations in MLPs.

the output layer and propagates through the inner layers (the red paths) for training. In this paper, the widely used mini-batch gradient descent algorithm [17] is considered for training and the mean squared error (MSE) is used as the loss function as represented in Eqn. 2. In this function, the error loss (L) represents the mean of the squared differences between the calculated output vector y and the expected/correct output vector $\hat{y}$, where N is the number of neurons in the output layer.

$$L(y,\hat{y}) = \frac{1}{N}\sum_{j=0}^{N}(y_j - \hat{y}_j)^2 \tag{2}$$

For updating each weight in the network during training, the first step calculates the gradient of the loss L related to such specific weight ($\partial L/\partial w_{j,k}^{i+1}(n)$) using the chain rule. By simplifying the chain rule for each weight in the network, the final value of the gradient is given in Eqn. 3, where $\delta_k^{i+1}(n)$ is the calculated gradient value using the chain rule starting from the output layer to neuron $n_k^{i+1}(n)$, $n_j^i(n)$ is the neuron in layer i, which is mapped to neuron $n_k^{i+1}(n)$ in layer i+1 through the weight $w_{j,k}^{i+1}(n)$ [9], and n is the number of iteration (i.e., the number of mini-batches).

$$\frac{\partial L}{\partial w_{j,k}^{i+1}(n)} = \delta_k^{i+1}(n) \times n_j^i(n) \tag{3}$$

The calculation of δ (as related to each weight) is similar for all layers except the output layer. Eqn. 4 formalizes these calculations, where q ranges from 1 to N, and $\hat{y}_q(n)$ represents the class related to neuron q in the output layer. For layers other than the output layer, the value of $\delta_j^i(n)$ for neuron $n_j^i(n)$ in layer i is found by multiplying $n_j^{i\prime}(n)$ with the sum of products of the calculated δ for each neuron in layer i+1 ($\delta_k^{i+1}(n)$) and their relevant weight ($w_{j,k}^{i+1}(n)$) with $n_j^i(n)$, where k ranges from 0 to the number of neurons in layer i+1 (denoted by l).

$$\begin{cases} \delta_q^{out}(n) = \frac{2}{N}(y_q^{out}(n) - \hat{y}_q(n)) \times y_q^{out\prime}(n) \\ \delta_j^i(n) = n_j^{i\prime}(n) \times \sum_{k=1}^{l} \delta_k^{i+1}(n) \times w_{j,k}^{i+1}(n) \end{cases} \tag{4}$$

The final adjusted value of the weight for a mini-batch iteration is given in Eqn. 5, where η is the learning rate.

$$w_{j,k}^{i+1}(n+1) = w_{j,k}^{i+1}(n) - \eta \frac{\partial L}{\partial w_{j,k}^{i+1}(n)} \tag{5}$$

TABLE I
FLOATING-POINT REPRESENTATIONS AND FORMATS

Precision	# Bits			Bias
	Total	Exponent	Mantissa	
HP	16	5	10	15
SP	32	8	23	127
DP	64	11	52	1023

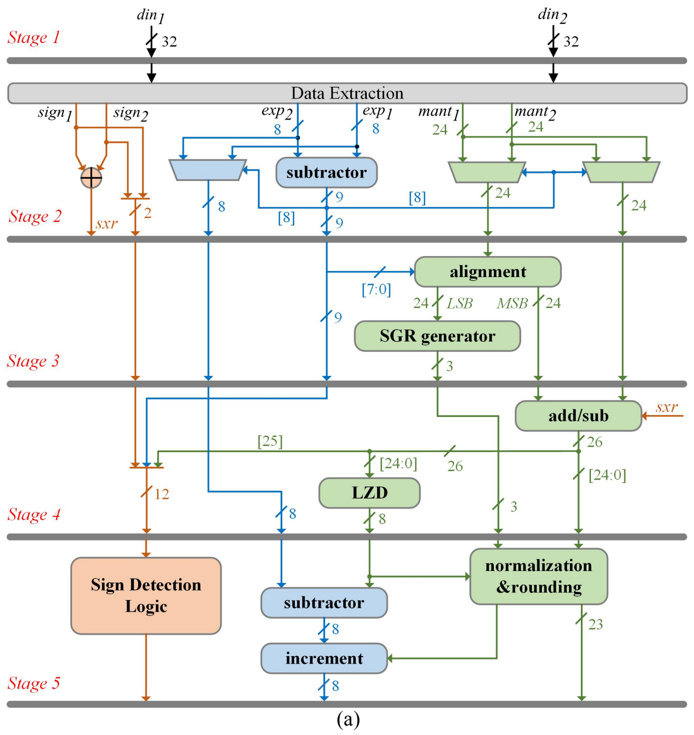
B. IEEE 754 Standard

Floating-point (FP) numbers have been extensively used in digital processing; the IEEE 754 standard is considered in this paper. Based on this standard, an FP number is represented by using three parts denoted by S, E and M that stand for the sign, exponent and mantissa bits, respectively [18]. The value of an FP number in this format is given as Eqn. 6.

$$FP\,Value = (-1)^S \times 2^{(E-Bias)} \times (H.M), \tag{6}$$

where $Bias$ represents the exponent bias value (specific to each precision); H is the hidden bit which is the logic OR of the exponent bits. In this paper, the effect of half (HP), single (SP) and double (DP) precision formats are investigated to assess accuracy and hardware for training ANNs. More details about these different FP representations are provided in Table I [19].

Many MLP designs only focus on accelerators for inference and the models are usually pre-trained by software; in this case,



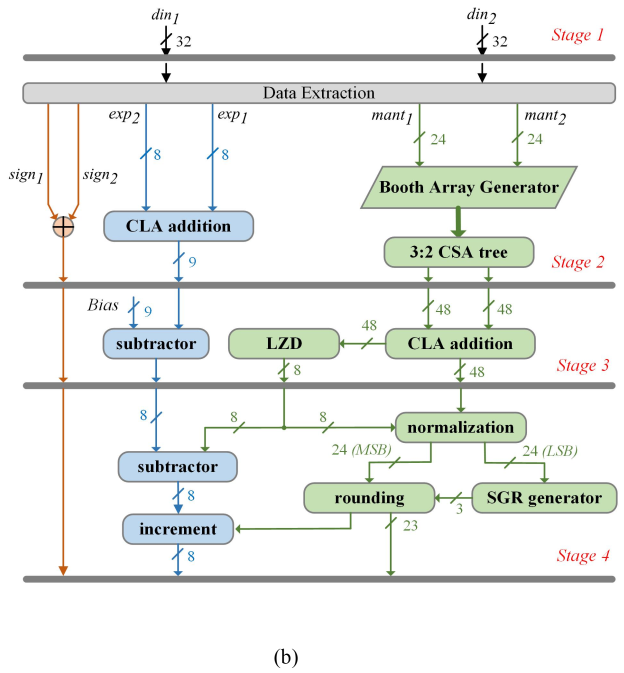


Fig. 2. Proposed pipelined floating-point computational units (single-precision format): (a) an adder with 5 pipelined stages; (b) a multiplier with 4 pipelined stages. The orange, blue and green data paths are related to the sign, exponent and mantissa data paths respectively.

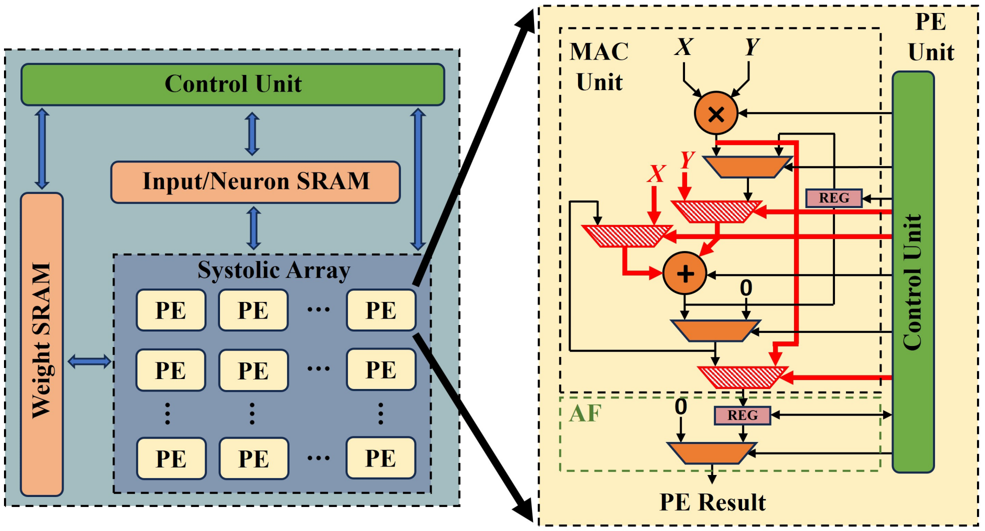


Fig. 3. An overview of the proposed MLP design using a PE array (left) and the MAC unit design in each PE (right). The MAC unit is shared for both forward and backward propagations; however, the red muxes and wires are triggered only during training.

the fixed-point computation is commonly employed. However, the objective of this paper is to implement training using hardware with a reasonable overhead to achieve a high accuracy. Therefore, the FP format is employed in the proposed design to address the limited dynamic range of fixed-point numbers [34], [35] which make them less efficient for training and applications which need a very high accuracy [36], [37]; also, the effectiveness of the proposed design with FP computation is verified by comparison with existing designs found in the technical literature in Section IV.

III. Proposed Design

In this paper, a high-performance hardware design for implementing both forward and backward propagations of an ANN is proposed by utilizing a hybrid floating-point MAC array; the network computation for the different data precisions is considered. Based on FP multiplication and addition algorithms [18]-[20], a fully pipelined MAC circuit design has been proposed in our previous work [21],[22]; it is redesigned in this paper to achieve high-performance requirements of the ANN implementation (e.g., high operational frequency). The design of different units as well as the ANN (in particular, an MLP) are described next[1].

A. Floating-point Adder and Multiplier

The proposed FP adder design is shown in Fig. 2(a); the data widths given in this figure are for single-precision format as an example, but the circuit design and computation process are the same for half/double precision formats. This adder has five pipelined stages to complete the computation and each step operates as follows:

State 1: The first stage filters and extracts the sign, exponent, mantissa, and hidden bits of the input numbers which are meant to be added.

[1]More details including the addition and multiplication algorithms, and the discussion on different network implementation schemes, can be found in the supplemental material.

State 2: This stage subtracts the exponent of two input numbers to calculate the offset between them. Then, based on the resulting sign, the mantissa of the smaller exponent is selected for alignment. The number of right shifts is determined by the offset value. Finally, the unchanged mantissa (24 bits), the to be aligned mantissa (24 bits), and the exponent of the larger number (8 bits) are moved to the next step.

State 3: This stage applies the alignment and then based on the aligned data, the SGR (Sticky, Guard and Round bits) generator is used to generate the bits required for rounding in the next steps.

State 4: Addition is performed for the 24 most significant bits of the aligned and the unchanged significands based on the difference in the signs of the two input numbers. The resulting value is sent to the leading zero detector (LZD) block (if the result is negative, the add/sub unit inverts it back to a positive format, while reporting the negative sign to the next step for sign detection).

Stage 5: In this step, the LZD offset obtained in the previous step is subtracted from the exponent of the larger number. Similarly, the final mantissa is obtained after the normalization and rounding steps. Also, In the LZD, it is assumed that the point is to the right after the carry bit (the point is shifted one bit to the left). Therefore, the final exponent needs to be increased by 1. Moreover, after rounding, an overflow may occur; in this case, the exponent must be updated again by incrementing it by one. Also in this stage, the sign detection logic chooses the final sign.

The FP multiplier design uses a similar computation algorithm, and requires (as shown in Fig. 2(b)) the following four pipelined stages:

Stage 1: Like the adder, filtering and extraction of the exponent, mantissa, and hidden bits from the primary input numbers take place in the first step.

Stage 2: The second step deals with the multiplication of the mantissas. A Booth radix-4 multiplier [23] is used. An array of multipliers is utilized and then added by a carry-save adder

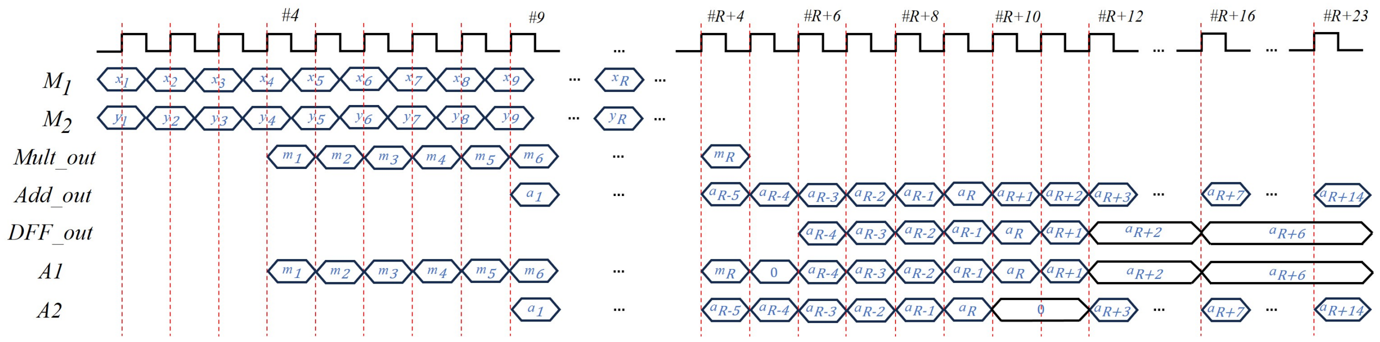


Fig. 4. Cycled execution of the proposed MAC unit.

Algorithm 1. MAC Computation Process

```
X and Y are floating-point input sequences valid till cycle R.
M1, M2 are the inputs of multiplier.
A1, A2 are the inputs of adder.
Mult_out, Add_out, DFF_out, and dout are the outputs of
multiplier, adder, flipflop, and the final value of MAC unit.

Step 1: (Multiplication)
  while (cycle ≤ 4)
    Activate multiplication and deactivate addition.
    M1 = X, M2 = Y; A1 = 0, A2 = 0;

Step 2: (Multiplication & Accumulation)
  while (5 ≤ cycle ≤ R+4)
    Activate multiplication and activate addition.
    M1 = X, M2 = Y;
    A1 = Mult_out, A2 = 0 till cycle 9 then switches to Add_out;

Step 3: (Flushing)
  while (R+6 ≤ cycle ≤ R+23)
    Deactivate multiplication and activate addition.
    A1 = DFF_out
    if (R+10 ≤ cycle ≤ R+11)    A2 = 0;
    else    A2 = Add_out;

    if (R+5 ≤ cycle ≤ R+12 & cycle = R+16)
      DFF_out = fetch(add_out);
    else DFF_out = unchanged;
```

(CSA) tree. The values of the exponents are also added by a carry-look-ahead (CLA).

Stage 3: The result of the CSA tree is obtained using a CLA adder. The LZD value is calculated based on it; a subtractor decreases the bias value from the sum of the exponents.

Stage 4: The final exponent value is found by decreasing the LZD offset from the exponent sum of Stage 3. The design normalizes the multiplication in Stage 3 using the LZD offset and then, it generates the SGR bits for rounding of the mantissa. The procedure/implementation for decrementing and incrementing is the same as in the final stage of FP adder.

B. MAC Unit

As introduced previously, the MAC unit is often implemented to perform the computation in the neurons. Fig. 3 (right) shows the pipelined MAC unit design proposed in this paper and it is shared by both forward and backward propagations. Compared to the MAC design of [21], the modified version of this paper can bypass the multiplication or addition operations when needed. This makes the back propagation more efficient, because when only an addition or multiplication is required during training, the number of clock cycles of a MAC is reduced from 9 to 5 or 4. Specifically and shown in Fig. 3, the red multiplexers and wirings in the MAC unit are used: (1) to send the primary inputs to the adder while bypassing the multiplier for the addition process, or (2) send the output of the multiplier to the primary output while bypassing the addition for the multiplication process. Therefore, these red multiplexers are not triggered during inference, but they start to switch during training due to the need for both MAC and single multiplication/addition operations.

The general MAC computation is described next. Assume that the inputs X and Y are valid for R cycles, the multiplier unit is activated and generates the product of the input pair after 4 cycles. Then, the multiplexers transfer this product to one input of the adder, as activated by the control unit. Since the adder requires 5 cycles, the other input of the adder is kept at 0 (as selected by the second multiplexer). At the end of the 9th cycle, both the multiplier and the adder have valid outputs. Therefore, the second multiplexer switches to the output of the adder to perform the accumulation and obtain the final result. The MAC computation process and its related waveform are then described as follows (also presented in Algorithm 1 and Fig. 4):

- All input pairs to the multiplier are received in the first R cycles.
- Then till cycle $R + 4$, the accumulation of the output of the multiplier (Mult_out) and the output of the adder is accomplished.
- After receiving the last product result in cycle $R + 4$, the pipeline stages can be flushed.
- For flushing, the flip-flop (DFF) unit stores the output of the adder (starting with cycle $R + 6$); it (as DFF_out) accumulates with the output of the adder at cycles $R + 6$, $R + 8$ and $R + 10$, respectively.
- At cycle $R + 12$, DFF saves the output of the adder generated from the data fetched at cycle $R + 6$.
- After the results of the data fetched from cycle $R + 8$ are ready at cycle $R + 13$, then they are fetched (a_{R+2} and a_{R+4}) at cycle $R + 14$ by the adder. The result is generated by cycle $R + 18$.

TABLE II
STAGE METRICS OF PIPELINED FLOATING-POINT MULTIPLIER/ADDER AFTER SYNTHESIS FOR DIFFERENT PRECISION FORMATS

Floating-point Unit*	HP			SP			DP		
	Area (um^2)	Power (mw)	Delay (ns)	Area (um^2)	Power (mw)	Delay (ns)	Area (um^2)	Power (mw)	Delay (ns)
Multiplier #2	1629.32	1.25	0.80	5935.53	3.90	1.06	24166.81	12.28	1.29
Multiplier #3	1190.67	0.90	1.07	3234.99	1.88	1.25	5786.09	2.98	1.36
Multiplier #4	1063.59	0.73	1.05	2065.43	1.31	1.24	4869.91	2.54	1.50
Adder #2	933.98	0.83	0.89	1883.97	1.47	1.05	3947.87	2.73	1.30
Adder #3	971.59	0.80	1.07	2314.23	1.53	1.25	4826.70	2.64	1.50
Adder #4	791.40	0.64	0.88	1671.25	1.08	1.06	3704.66	1.94	1.19
Adder #5	359.87	0.35	0.41	656.45	0.55	0.51	1236.16	0.89	0.61

* The number that follows # refers to the pipeline stage of each unit.

- Similarly, the result of the fetched inputs at cycle R + 11 is ready at cycle R + 15. The DFF saves it starting with cycle R + 16 and then the adder fetches it as well as a_{R+9} at cycle R + 19; hence, the valid output is generated at cycle R + 23.

A processing element (PE) array that is similar to the systolic array of [24], [25], is designed by employing the proposed MAC units to implement the forward and backward propagations of an MLP. However, different from [24], [25], the proposed design is intended for FP computation because this core is used for on-chip training too; moreover, the proposed PE array is fully pipelined for achieving a high operational frequency. This will be explained in detail in the following subsections.

C. Forward Propagation of MLPs

The proposed MLP design uses a hybrid configuration to allow the use of the same PE array (Fig. 3) for both inference and training without imposing additional hardware overhead (and regardless of memory size). In the proposed hybrid implementation scheme, each PE consists of a MAC unit and an activation function (AF). The number of PEs is critical in parallelizing the process and making the MLP computation efficient. However, due to the limitations in hardware metrics for some applications (such as mobile and low power systems), a balance between latency/operational frequency and power/area characteristics must be pursued. As a trade-off, the proposed scheme utilizes 64 PEs in total for the MLPs considered in this paper, but the number can be adjusted for different models. Therefore, in our case, if a layer has A neurons, then $\lceil A/64 \rceil$ iterations are needed to allocate the PE units for each neuron in this layer during forward/backward propagation (where each PE represents a single neuron).

For implementing forward propagation, if the network has A neurons in layer i and B neurons in layer i+1, the hybrid PE array needs $\lceil B/64 \rceil$ iterations to obtain the neuron values for layer i+1; each iteration takes A+1 cycles, because each PE consists of a single MAC unit and receives all input pairs (neuron values and their related weight) plus a bias value serially from layer i (consisting of A neurons). Then it takes ($\lceil B/64 \rceil$ × (A+1)) cycles to complete with layer $i+1$. The same procedure is applied to other layers in the network. The final value of each PE (*PE result*) is then saved; finally, the results are used for calculations in the next layers during forward propagation.

Therefore, the hybrid design is independent of network size, because it performs calculations iteratively in multiple steps. As shown in Fig. 3, a control unit is required in each PE in addition to the MAC and AF units. The control unit operates when the start signal is active (the inputs X and Y are valid) and the *I/T* signal determines the inference or training. When *I/T* is 1, the PE operates for inference and includes the AF by considering the sign bit of the MAC output (*dout*). Therefore, the *PE result* is 0 if the most significant bit of *dout* is one, otherwise it is equal to *dout.* When *I/T* is 0, the PE operates the training calculation so bypassing AF; in this case, the PE result is always equal to *dout.*

D. Backward Propagation of MLPs

For backward propagation, we use the mini-batch training technique based on the gradient descent algorithm (reviewed in Section II-A). To balance hardware with delay, the same PE array as for inference is utilized for backward propagation. In this case, the total area and power dissipation per cycle remain approximately constant.

Assume that the mini-batch size is given by M, then Eqn. 2 is calculated for each of the M samples in the batch and then averaged before adjusting the weight, i.e., Eqn. 4 is converted to:

$$w_{j,k}^{i+1}(n) = w_{j,k}^{i+1}(n) - \frac{\eta}{M}\sum_{b=1}^{M} \frac{\partial L_b}{\partial w_{j,k}^{i+1}(n)}. \qquad (7)$$

Therefore, instead of decreasing the gradient of the loss function for each element in the dataset, the average gradient for M samples is decreased to perform training. To perform the computations by using the proposed hybrid implementation scheme, the following two potential strategies can be utilized. The first solution is to calculate the gradient of the loss function for each neuron and for each of the M samples separately and then apply Eqn. 7. However, this method is not efficient because there are many redundant gradient calculations. For instance, for updating two arbitrary weights in layers i and i+1, redundant calculations can be observed from Eqn. 8 and Eqn. 9, respectively. Based on Eqn. 3, the term $(\sum_{g=1}^{W} \delta_g^{i+2}(n) \times w_{k,g}^{i+2}(n)) \times {n_k^{i+1}}'(n)$ of Eqn. 8 represents the calculated gradient value $\delta_k^{i+1}(n)$ related to an arbitrary neuron k in layer i+1; this value is required to calculate the gradient of the loss function (L) to $w_{j,k}^{i+1}(n)$. Similarly, for another arbitrary weight in the previous layer ($w_{h,j}^{i}(n)$), the value of δ for each neuron in layer i+1 (including the previously mentioned neuron k) is

TABLE III
SYNTHESIS RESULTS FOR MLP IMPLEMENTATION WITH DIFFERENT FLOATING-POINT PRECISIONS

Dataset	Precision	Area (mm^2)	Power (mw)	Freq. (MHz)	Inference Delay (us)	Mini-batch training delay and the accuracy after training					
						Size = 32		Size = 64		Size = 128	
						Delay (ms)	Accuracy	Delay (ms)	Accuracy	Delay (ms)	Accuracy
MNIST	HP	0.37	258.33	934	2.51	1.60	98.28 %	3.14	98.40 %	6.25	98.48 %
	SP	0.96	550.49	800	2.94	1.87	98.32 %	3.67	98.45 %	7.29	98.51 %
	DP	2.84	1278.54	666	3.53	2.24	98.33 %	4.41	98.47 %	8.75	98.51 %
SVHN	HP	0.37	258.33	934	6.60	5.14	89.72 %	10.12	90.05 %	20.10	90.22 %
	SP	0.96	550.49	800	7.72	6.01	89.90 %	11.83	90.15 %	23.48	90.29 %
	DP	2.84	1278.54	666	9.26	7.21	89.94 %	14.19	90.16 %	28.17	90.31 %

required to calculate the value of $\delta_j^i(n)$ for neuron j in layer i; this computation is represented by the term $((\sum_{g=1}^{W} \delta_g^{i+2}(n) \times w_{c,g}^{i+2}(n)) \times n_c^{i+1'}(n))$ in Eqn. 9, which is the same as in Eqn. 8. Therefore, during the gradient calculations by propagating through each inner layer starting from the output layer, this term is calculated redundantly.

$$\frac{\partial L}{\partial w_{j,k}^{i+1}(n)} = (\textstyle\sum_{g=1}^{W} \delta_g^{i+2}(n) \times w_{k,g}^{i+2}(n)) \times n_k^{i+1'}(n) \times n_j^i(n) \quad (8)$$

$$\begin{aligned}\frac{\partial L}{\partial w_{h,j}^{i}(n)} &= (\textstyle\sum_{c=1}^{P} \delta_c^{i+1}(n) \times w_{j,c}^{i+1}(n)) \times n_j^{i'}(n) \times n_h^{i-1}(n) \\ &= (\textstyle\sum_{c=1}^{P} ((\sum_{g=1}^{W} \delta_g^{i+2}(n) \times w_{c,g}^{i+2}(n)) \times n_c^{i+1'}(n)) \times \\ &\quad w_{j,c}^{i+1}(n)) \times n_j^{i'}(n) \times n_h^{i-1}(n) \qquad (9)\end{aligned}$$

Although these redundant gradient computations do not affect the hardware of the proposed design (it is a hybrid scheme and independent of the size of network), they significantly increase latency. To address this issue, we use an alternative scheme to decrease the latency.

So instead of computing the gradient of the loss function to each weight separately, we first calculate the value of δ for each neuron; then the term $\partial L/\partial w$ for each weight is calculated based on the previously saved δ values. This method is very similar to inference and the goal is to calculate $\partial L/\partial w$ for all weights, starting from the output layer through the inner layers. Specifically, for an arbitrary neuron k in layer i+1, the value of δ_j^{i+1} is calculated by initially accumulating the product of δ^{i+2} for each neuron in layer i+2 and its related weight with neuron j in layer i. Then, based on Eqn. 4, the accumulated value is multiplied by the derivative of the value of the current neuron (n_j^i) from inference. Since ReLU is utilized as activation function[2], the final values of the PEs during inference are either 0 or positive; then, the accumulation result is assigned to 0 if the neuron value is 0 (because the derivative is 0), otherwise it is saved as $\delta_k^{i+1}(n)$ (because the derivative of ReLU for positive values is 1). For simplicity in design, during training the PE receives the value of n_j^i and then based on this value, the control unit decides to keep the PE in the reset or active mode. If it stays in the reset mode, the PE result is 0, otherwise it is equal to the accumulated value. Like inference, as the model is hybrid, then this process is performed over multiple steps for each layer.

[2]As application specific, the ReLU function has been considered in this paper. However, the proposed design methodology is also applicable to other types of activation functions. In this case, the backpropagation flow may be changed by adding some logic circuitry to calculate the derivative of the activation function; this is not further investigated in this paper.

After computing all δ values for each neuron, the value of the gradient for each weight is processed as in Eqn. 3. For mini-batch training, the current gradient value is accumulated with the previous gradient values for all samples in the mini-batch, and the average value is used to adjust the weights as per Eqn. 7. To do so, for the arbitrary weight $w_{j,k}^{i+1}$, the PE array receives the (δ_k^{i+1}, $n_j^i(n)$) and (accumulated gradient,1) pairs to accumulate the value of the current gradient with the accumulation of previous gradients (referred to as the accumulated gradient) for each weight in the NN. This process also takes multiple steps; overall, the PE array receives the pairs $(\frac{-\eta}{M}, \sum_{b=1}^{M} \frac{\partial L}{\partial w_{j,k}^{i+1}(n)})$ and $(w_{j,k}^{i+1}(n),\ 1)$ for accumulation (where η is the learning rate). Since the coefficient of $\frac{-\eta}{M}$ is previously saved in memory, then no additional computational hardware is required in the hybrid PE array.

IV. EVALUATION

The proposed design of the MAC unit and the hybrid MLP network using the MAC array are implemented using Verilog-HDL and then synthesized using the Cadence Genus Synthesis Solution. The used library is 32nm with a corner of TT and temperature of 25°C. Two widely used ML datasets (MNIST [26] and SVHN [27]) are used for assessing the proposed MLP design. The network for all datasets has four layers, and the number of neurons per layer is 784, 100, 200, 10 for MNIST, and 1024, 100, 200, 10 for SVHN.

The hardware of the MAC unit is evaluated first. As discussed previously, the MAC unit computes by utilizing a FP adder and multiplier iteratively; hence, the critical path delay is determined by the worst delay among the stages for the addition and multiplication. Therefore, the hardware metrics of different pipeline stages for FP multiplier/adder (Fig. 2) are evaluated in detail. The first stage for the multiplier/adder only filters the input data with no combinational hardware, therefore such configuration is not considered and evaluated. Table II reports the synthesis results for different stages of the MAC unit with different FP precision formats; as expected, the circuit for HP computation incurs in the lowest hardware for all evaluated metrics, while DP requires the largest. To evaluate the combinational elements in each stage, some flip-flops must be inserted in the input and output to allow the set of timing constraint on the circuits of each stage during synthesis. Therefore, the reported results include the flip-flops too. As per the worst path delay of the MAC unit given in Table II (i.e., 1.07 *ns* for HP, 1.25 *ns* for SP, and 1.50 *ns* for DP), the timing

TABLE IV
COMPARISON OF DIFFERENT MLP INFERENCE DESIGNS FOR MNIST DATASET

Design*	**Technology**	**On-chip Training**	**Network Size**	**Data Type**	**Frequency (*MHz*)**	**Area (mm^2)**	**Energy (*μJ*)**	**Inference Latency (*μs*)**	**Accuracy**
[28]	FPGA-28nm	×	784-8000-10	Fixed-point	266	-	-	120.00	96.55%
[29]	FPGA-16nm	×	784-12-10	SP floating-point	100	-	0.88	**1.55**	93.25%
[30]	FPGA-20nm	×	784-1200-1200-10	Fixed-point	250	-	373.41	845.00	98.44%
[31]	ASIC-130nm	✓	784-144-10	Fixed-point	200	6.76	6.41	2.98	94.00%
[32]	ASIC-45nm	×	784-1200-600-10	Fixed-point	-	-	1820.00	-	**98.88%**
[33]	ASIC-65nm	×	784-256-10	Fixed-point	500	0.55	3.38	25.63	96.40%
Proposed	ASIC-32nm	✓	784-100-200-10	HP floating-point	**934**	**0.37**	**0.65**	2.51	98.40%
				SP floating-point	800	0.96	1.61	2.94	98.45%

* For [28] and [32], only the performance metrics reported in the references are given, so few entries are missing in this table.

constraint of the PE array is accordingly set during synthesis to ensure proper operation.

The synthesis results of the proposed MLP design (i.e., the PE array) for both inference and training (using mini-batch sizes of 32, 64, and 128 as examples) are given in Table III; the hardware metrics for different datasets and FP precision formats are evaluated without considering the SRAM requirements. Since a hybrid design is proposed for NN computation, area and power dissipation per cycle remain constant for different datasets when using the same precision format; however, the delays of inference (forward propagation) and training (forward and backward propagation) depend on the NN size (shown in Table III). As per this table, HP and SP tend to be a better solution for MLP computation compared to DP; while achieving the same or slightly lower ML accuracy than DP, HP and SP permit a significantly higher operating frequency and lower hardware overhead for the ASIC hardware.

As shown in Fig. 3, the proposed MLP designs require SRAMs external to the PEs to store the network parameters and temporary computation results; the size of the memories is dependent on the dataset as well as directly proportional to the FP data format (in the case of SP, the memory is twice as large as HP and only half of DP). Our evaluation has found that for SP, the proposed designs require 786.79 MB for MNIST and 974.29 KB for SVHN (the memory size should be adjusted for other FP data formats as previously discussed).

For comparison purposes, Table IV shows the results for different MLP designs found in the technical literature for the MNIST dataset; the proposed designs with HP and SP computation formats are considered for comparison. As per Table IV, the proposed design offers the best operating frequency despite the added complexity of using a FP implementation compared to some of the others. When considering accuracy, the proposed design also shows a very good result due to the mini-batch training and FP computation. Although the design of [32] achieves the highest accuracy, it is due to a more complex model, so accounting for a large energy dissipation.

Due to the different employed technology conditions (such as high node feature size and FPGAs as platform for design), a quantitative comparison in terms of hardware for the MLP designs is not always possible; however, the proposed design has reasonable energy dissipation and latency (as reported in Table IV). Moreover, the proposed MLP design is independent of network size, which makes it a general-purpose core for implementing ANNs. It should also be noted that only the proposed scheme and the design of [31] have an on-chip training capability and the proposed scheme has the better performance.

V. CONCLUSION

In this paper a high-performance ASIC-based design for implementing the computational core for both inference and training (on-chip) of ANNs has been proposed; this design relies on a hybrid processing element (PE) array that employs pipelined floating-point MAC units. The proposed design is highly flexible because its hardware is independent of network size and allows the use of the same PE array for both forward and backward propagations without imposing any additional hardware overhead (however, this independence in metric is not applicable to latency). An MLP has been designed for both these propagations and by utilizing three different floating-point formats (half, single and double precision), the evaluation results have shown that the design with either half or single precision format is more attractive for MLP implementation due to the significantly lower hardware (while achieving the same or slightly lower classification accuracy compared to double precision). Moreover, in general, the proposed design has been shown to be superior in terms of operating frequency and energy compared to other schemes found in the technical literature.

REFERENCES

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in Artificial Neural Network Npplications: A Survey," *Heliyon*, vol. 4, no. 11, pp. 1-41, Nov. 2018.

[2] A. P. Marugán, F. P. G. Márquez, J. M. P. Perez, and D. R. Hernández, "A Survey of Artificial Neural Network in Wind Energy Systems," *Applied energy,* vol. 228, pp. 1822-1836, Oct. 2018.

[3] L. M. Elobaid., A. K. Abdelsalam, and E. E. Zakzouk, "Artificial Neural Network-based Photovoltaic Maximum Power Point Tracking Techniques: A Survey," *IET Renewable Power Generation,* vol. 9, no. 8, pp. 1043-1063, Nov. 2015.

[4] A. Savich, M. Moussa, and S. Areibi. "A Scalable Pipelined Architecture for Real-time Computation of MLP-BP Neural Networ," *Microprocessors and Microsystems,* vol. 36, no. 2, pp. 138-150, Mar. 2012.

[5] Q. Zhang, H. Yu, M. Barbiero, B. Wang, and M. Gu. "Artificial Neural Networks Enabled by Nanophotonics," *Light: Science & Applications,* vol. 8, no. 42: pp. 1-14, May 2019.

[6] T. Nordström and B. Svensson. "Using and Designing Massively Parallel Computers for Artificial Neural Networks." *Journal of parallel and distributed computing,* vol. 14, no. 3, pp. 260-285, Mar. 1992.

[7] E. Giacomin, S. Gudaparthi, J. Boemmels, R. Balasubramonian, F. Catthoor and P. E. Gaillardon, "A Multiply-and-Accumulate Array for Machine Learning Applications Based on a 3D Nanofabric Flow,"*IEEE Transactions on Nanotechnology*, vol. 20, pp. 873-882, Dec. 2021.

[8] N. Nedjah, R. M. da Silva, L. M. Mourelle, and M. V. C. da Silva, "Dynamic MAC-based Architecture of Artificial Neural Networks Suitable for Hardware Implementation on FPGAs," *Neurocomputing*, vol. 72, no. 10-12, pp. 2171-2179, Jun. 2009.

[9] Haykin, Simon. *Neural networks and learning machines,* third edition. Pearson Education India, 2009.

[10] L. Ye, Z. Wang, Y. Liu, P. Chen, H. Li, H. Zhang, *et al*, "The Challenges and Emerging Technologies for Low-power Artificial Intelligence IoT Systems," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 68, no. 12, pp. 4821-4834, Aug. 2021.

[11] A. G. Blaiech, K. B. Khalifa, C. Valderrama, M. A. C. Fernandes, and M. H. Bedoui. "A Survey and Taxonomy of FPGA-based Deep Learning Accelerators," *Journal of Systems Architecture,* vol. 98, pp. 331-345, Sept. 2019.

[12] M. A. Talib, S. Majzoub, Q. Nasir, and D. Jamal, "A Systematic Literature Review on Hardware Implementation of Artificial Intelligence Algorithms," *The Journal of Supercomputing,* vol. *77,* no. 2, pp. 1897-1938, May 2020.

[13] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating CNN Inference on ASICs: A Survey," *Journal of Systems Architecture*, vol. 113, Feb. 2021.

[14] H. Ramchoun, Y. Ghanou, M. Ettaouil, and M. A. J. Idrissi. "Multilayer Perceptron: Architecture Optimization and Training," *International Journal of Interactive Multimedia and Artificial Intelligence, vol. 4, no. 1,* Sept. 2016.

[15] M. W. Gardner, and S. R. Dorling. "Artificial Neural Networks (The Multilayer Perceptron)—A Review of Applications in The Atmospheric Sciences," *Atmospheric environment,* vol. 32, no. 14-15, pp. 2627-2636 Aug. 1998.

[16] D. Fan, Y. Shim, A. Raghunathan, and K. Roy, "STT-SNN: A Spin-Transfer-Torque Based Soft-Limiting Non-Linear Neuron for Low-Power Artificial Neural Networks," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 1013-1023, Nov. 2015.

[17] G. Hinton, N. Srivastava, and K. Swersky. Neural Networks for Machine Learning: Lecture 6a Overview of Mini-batch Gradient Descent. *Feb.* 2012.

[18] S. Galal and M. Horowitz, "Energy-Efficient Floating-Point Unit Design," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 913-922, Jul. 2011.

[19] IS Committee. (2008). 754–2008 IEEE Standard for Floating-point Arithmetic. IEEE Computer Society Std, 2008.

[20] J. M. Muller, N. Brisebarre, F. De Dinechin, C. P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres," *Handbook of Floating-point Arithmetic,*" Basel, Switzerland, Birkhäuser. vol. 1, 2018.

[21] F. Niknia, Z. Wang, S. Liu, and F. Lombardi, "Nanoscale Design of Multi-Layer Perceptrons using Floating-Point Arithmetic Units," *in IEEE 22nd International Conference on Nanotechnology*, Nov. 2022, pp. 16-19.

[22] F. Niknia, Z. Wang, S. Liu, A. Louri, and F. Lombardi, "Nanoscale Accelerators for Artificial Neural Networks: Arithmetic Design, Analysis and ASIC Implementations," *IEEE Nanotechnology Magazine*, Accepted Aug. 2022.

[23] H. Zhang, D. Chen and S. B. Ko, "Efficient Multiple-Precision Floating-Point Fused Multiply-Add with Mixed-Precision Support," *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1035-1048, Jul. 2019.

[24] Y. H. Chen, T. J. Yang, J. Emer and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292-308, Jun. 2019.

[25] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," arXiv preprint arXiv:1811.02883, 2018.

[26] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, 1998. "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[27] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng, 2011. Reading Digits in Natural Images With Unsupervised Feature Learning.

[28] R. Wang, C. S. Thakur, G. Cohen, T. J. Hamilton, J. Tapson, and A. van Schaik, "Neuromorphic Hardware Architecture Using The Neural Engineering Framework for Pattern Recognition," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 3, pp. 574-584, Jun. 2017.

[29] I. Westby, X. Yang, T. Liu, and H. Xu, "FPGA Acceleration on A Multi-Layer Perceptron Neural Network for Digit Recognition," *The Journal of Supercomputing,* vol. 77, no. 12, May 2021.

[30] Y. Wang, H. Zhang, K. I. Oh, J. J. Lee, and S. B. Ko, "Energy efficient spiking neural network processing using approximate arithmetic units and variable precision weights," *Journal of Parallel and Distributed Computing*, vol. *158*, pp.164-175, Dec. 2021.

[31] D. Kim, J. Kung, and S. Mukhopadhyay, "A Power-Aware Digital Multilayer Perceptron Accelerator with On-Chip Training Based on Approximate Computing," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 164-178, Feb. 2017.

[32] S. S. Sarwar *et al.*, "Energy Efficient Neural Computing: A Study of Cross-Layer Approximations," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 796-809, Dec. 2018.

[33] F. Tu, *et al.*, "Reconfigurable Architecture for Neural Approximation in Multimedia Computing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 3, pp. 892-906, Mar. 2019.

[34] X. Sun, J. Choi, C. Chen, *et al.*., "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks," *Advances in neural information processing systems32,* Dec. 2019.

[35] M. Courbariaux, Y. Bengio, and J. P. David. "Training Deep Neural Networks with Low Precision Multiplications." arXiv preprint arXiv:1412.7024, Sept. 2015.

[36] M. A. Cavuslu, C. Karakuzu, S. Sahin, and M. Yakut. "Neural Network Training based on FPGA with Floating Point Number Format and it's Performance." *Neural Computing and Applications* vol. 20, pp. 195-202, Mar. 2011.

[37] M. Junaid, S. Arslan, T. Lee, and H. Kim. "Optimal Architecture of Floating-point Arithmetic for Neural Network Training Processors." *Sensors*, vol. 22, no. 3 pp. 1230-1245, Feb. 2022.