Power-Efficient Implementation of Ternary Neural Networks in Edge Devices

Miguel Molina*, Javier Mendez*, Diego Pedro†, Encarnación Castillo†, Marisa Lopez Vallejo§, and Manuel Pegalajar‡
*Infineon Technologies AG, Am Campeon 1-15,85579 Neubiberg, Germany
Email: Miguel.Molina.Fernandez2@infineon.com
†Department of Electronics and Computer Technology, University of Granada, 18071 Granada, Spain
‡Department of Computer Science and Artificial Intelligence, University of Granada, 18071 Granada, Spain
§IPTC, ETSI Telecomunicación, Universidad Politécnica de Madrid, Avda. Complutense 30, 28040 Madrid, Spain

***Abstract*—There is a growing interest in pushing computation to the Edge, especially the problem-solving abilities of Artificial Neural Networks (ANNs). This paper presents a simplified method to obtain a Ternary Neural Network based on the Multilayer Perceptron. The method is focused on resource-constrained devices, where memory, computing power, and battery are some of the most relevant constraints. A dynamic threshold is estimated to perform ternarization, and a new pruning technique is proposed to obtain a drastic reduction in the ANN's size, with the corresponding decrease in resource utilization and power consumption of the resulting hardware. In addition, a support framework has been developed to automate hardware design exploration and generation from the network trained in software. Experimental results show that the proposed method and architecture, when implemented in a Field-Programmable Gate Array (FPGA), provide excellent figures in power (0.11-0.13 W) and efficiency (1,225-1,448 kfps/W) with respect to state-of-art, being its efficiency double than the maximum one reported previously.**

***Index Terms*—Constrained Devices, Energy Efficient Devices, Low Power Devices and Circuits, Real-Time Systems.**

I. Introduction

Artificial Neural Networks have gained attention in the last few decades thanks to their success in solving multiple problems, such as pattern recognition, regression and classification. Moreover, their properties as general approximators have allowed this family of algorithms to establish themselves on the podiums of numerous well-known problem solutions, as could be MNIST [1], CIFAR [2], and ImageNet [3]. However, despite their huge potential, ANNs are resource-hungry, especially in Deep Learning scenarios. That usually implies the need for complex architectures and large computing resources to reach relevant results, which is not affordable for Edge Computing and Internet of Things (IoT) applications.

Many approaches have appeared to reduce the complexity of ANNs. Some of them attempt to use techniques as sparsity increment [4], pruning [5] or shared weights [6], while others look for reducing the size of the data, from the floating point precision (FPP) of software implementations to the extreme case of 1 bit representation [7]. With a fine parameter initialization, these techniques allow an ANN to keep a high accuracy level while decreasing latency, resources, and power consumption [8].

Other researchers have focused on creating efficient hardware (HW) devices for the implementation of ANNs [9] or the adaptation of general devices, such as Graphics Processing Unit (GPUs) and FPGAs with this purpose [8]. FPGAs have gained a growing interest in the last few years to be considered a middle point between Application-Specific Integrated Circuit (ASICs) and GPUs: First, FPGAs allow the possibility of creating problem-specific tailored implementations without starting from scratch, with fully verified and perfectly documented low-cost boards. And second, their parallel processing fits quite well with the concurrent nature of ANNs, which allows them to surpass GPUs in speed, resources and power efficiency in some cases [10].

In this context, it is reasonable to think that Edge Computing and IoT efficiency can be improved by linking the implementation reduction techniques of ANNs with specific hardware solutions, where real-time or time-constrained processing and power consumption are critical. Based on the simplest ANN topology, the Multilayer Perceptron (MLP), this paper investigates a more straightforward way of designing and training Binary/Ternary Neural Networks for their hardware implementation on FPGAs. Our proposal focuses on having low power consumption and using few computational resources since both features are essential for any IoT application. Thus, we may enumerate the main contributions of this work as follows:

- A ternarization technique is proposed to help maintain the ANN accuracy while significantly reducing the computational requirements of the model. It is fully compatible with the vanilla feedforward MLP's structure [11].
- A new pruning method is presented, which can reduce the number of neurons at the hidden layers without losing accuracy. Pruning allows a decrease in both hardware requirements and power consumption.
- A specific FPGA architecture is proposed to be an alternative for low-power IoT applications. It uses a bias initialization technique to reduce the number of operations made in the device.
- A framework has been created to support the software training algorithms and to generate the hardware architecture, providing an automatic interaction between them.

These contributions focus on decreasing the system's power consumption and resource utilization with respect to the state-of-art. We have obtained an efficiency in the range of 1,225-1,448 kfps/W, doubling the best one found in the literature while a high accuracy remains. Furthermore, we reach 90% of the final accuracy at the first epoch, leading to state-of-the-art accuracy results without the need of long training times (20-40 epochs), as we further discuss in Section VI.

The remaining of this paper is organized as follows: Section 2 provides background on BNNs and the existing FPGA implementations. The proposed model and pruning method are presented in section 3. Section 4 describes the optimizations made towards the hardware implementation. Section 5 explains the automatic interface between software and hardware. Section 6 discusses the experimental evaluation, and to conclude, in Section 7 some conclusions are drawn.

II. Quantized Neural Networks implementations

Binary Neural Networks (BNNs) [12] are currently a hot research topic when deploying ANNs in FPGAs. This technique can be seen as an extreme quantization case whose aim is to reduce the data representation to decrease the complexity of the operations, the power consumption, and the resources and memory usage of the models [13]. Furthermore, these are the characteristics required by IoT and Edge applications. Among BNNs, there are two main variants: fully-binarized networks, where all the parameters of the network are binarized (including weights, activation functions, and input data), and partially binarized, where only some of the parameters are binarized.

In 2017 H. Alemdar et al. [14] proposed Ternary Neural Networks (TNN) for resource-efficient applications, using weights and activation functions constrained to {-1, 0, 1}. This network structure presents low energy consumption (only 0.377 W when used for the CIFAR10 dataset) while achieving high accuracy results. Similarly, M. Ghasemzadeh et al. [9] proposed a residual BNN framework to train and deploy them on FPGA platforms. They analyze the reduction in accuracy by representing features with multiple levels of residual binarization. This technique provides a trade-off between hardware performance and accuracy while the area required in the FPGA for the multi-level binarization is negligible.

Other authors used binarization not to improve the efficiency but as an accelerator, since it leads to a reduction in the number of operations required to execute the ANN [8][15][16]. Y. Umuroglu et al. [8] proposed a framework, FINN, where different layer structures were binarized and implemented in a ZC706 FPGA to test its performance. This framework achieved latency's results in the order of microseconds, while maintaining accuracy results over 80% for all the tested datasets. An improved version of the framework [15] included a hardware cost estimator and further optimizations of the architecture and platforms while maintaining high-performance results.

In this work we aim to face both goals simultaneously since optimizing the BNN can reduce latency and resource usage. The power consumption will take more relevance as the developed architecture focuses on Edge devices. To improve these features, this work is developed using a simpler model with respect to the state-of-art, which also brings some optimization regarding the hardware architecture. At the same time, we based our framework on reduced network structures, removing non-required layers such as batch normalization. This approach leads to outperforming the best results reported previously by 21 times lower consumption and twice larger efficiency. The model and its architecture are introduced in the following sections.

III. Ternary Neural Network model and training

A. Proposed model

The model proposed in this paper has the same structure as an MLP using the vanilla backpropagation (BP) algorithm for training [17]. We propose to carry out some adaptations for the ternary ANN implementation. Our modifications are based on the use of two different weights and biases:

- First, weights and biases $\tilde{W}$ and $\tilde{B}$ are used to keep the learned values of the network following the traditional backpropagation procedure. They are used in the calculation of each neuron output, as shown in equations 1 and 2. Their initialization is done with a zero mean random distribution of FPP numbers. However, $\tilde{W}$ and $\tilde{B}$ will be progressively updated to ternary values {-1, 0, 1} during the training by a ternarization technique, summarized in Table I.
- Secondly, we use and track full-floating point precision weights and biases W and B. They aim to accumulate all the updates given by the BP algorithm in equations 3 and 4. Their values are initialized to 0.

$$sum_i^k = \sum_{j=1}^{n_{k-1}} (y_j^{k-1} \tilde{W}_{ji}^k) + \tilde{B} i^k \tag{1}$$

$$y_i^k = a_i^k(sum_i^k) \tag{2}$$

$$W_{ji}^k = W_{ji}^k - \alpha \Delta \tilde{W}_{ji}^k \tag{3}$$

$$B_i^k = B_i^k - \alpha \Delta \tilde{B}_i^k \tag{4}$$

In these equations, k denotes the layer, i is the respective neuron in each layer k, and j is the index for the j-th output from the previous layer. α is the learning rate, a^k is the activation function of all neurons at layer k (whose number is n^k), and y_i^k denotes their output. Equations 1 and 2 show the calculation of each neuron's outputs as a perceptron, while equations 3 and 4 are the updates done after BP.

W_{ji}^k and $\tilde{W}_{ji}^k$ are related through a dynamic threshold (T). When $abs(W_{ji}^k)$ surpasses T_{ji}^k, $\tilde{W}_{ji}^k$ could be updated to a new ternary value in the range {-1, 0, 1}. This threshold is initialized with an integer value, which is the same for each weight. Then, in case of surpassing, T_{ji}^k adopts the absolute value of W_{ji}^k, multiplied by a factor (f in Table I). This factor needs to be greater than 1 to ensure the growth of T during the training.

TABLE I
WEIGHTS UPDATING WITH THE DYNAMIC THRESHOLD T.

If	and	Then		
W_{ji}^k	$Int(\tilde{W}_{ji}^k)$ =	W_{ji}^k	$\tilde{W}_{ji}^k$	T_{ji}^k
$< -T_{ji}^k$	-1	W_{ji}^k	-1	T_{ji}^k
$< -T_{ji}^k$	0	0	-1	$Round(abs(W_{ji}^k)) \times f$
$< -T_{ji}^k$	1	0	0	$Round(abs(W_{ji}^k)) \times f$
$> T_{ji}^k$	-1	0	0	$Round(abs(W_{ji}^k)) \times f$
$> T_{ji}^k$	0	0	1	$Round(abs(W_{ji}^k)) \times f$
$> T_{ji}^k$	1	W_{ji}^k	1	T_{ji}^k

The concept of a dynamic threshold has been explored before by other approaches. For example, in [14] H. Alemdar et al. based their ternarization model on a step function with two trained thresholds per neuron as activation function. However, we use it at the BP algorithm during the training without limiting the activation function to a step representation.

The six possible updating cases are summarized in Table I. The reason of getting the integer value of $\tilde{W}_{ji}^k$ is that these weights are initialized as a small FPP. Therefore, they do not get its ternary value until they are updated. This process has shown a faster convergence and performance in the experiments, rather than directly starting with their respective ternary values. It should be noted that when $\tilde{W}_{ji}^k$ is going to keep the same value (either because $abs(W_{ji}^k)$ does not overtake T_{ji}^k or because it tends to continue rising out of the imposed limit -1, 1), there is not any update. In this case, $abs(W_{ji}^k)$ will continue increasing, giving fewer opportunities to $\tilde{W}_{ji}^k$ to change its value.

Four different examples are shown graphically in Figure 1. A factor f = 1.5 is used for all of them. Only the examples numbered 2 and 4 have had an update, due to surpassing the threshold. For number 3 the threshold is also exceeded positively, but as $\tilde{W}$ had already taken its maximum positive value, nothing changes.

E.g.:	T	W	$\tilde{W}$		T	W	$\tilde{W}$
1	7	-0.5	0		7	-0.5	0
2	3	-6.6	0	Then	10	0	-1
3	8	9.3	1		8	9.3	1
4	2	3.7	-1		6	0	0

Fig. 1. Examples of weights updating (updates are in red).

B. Training data preprocessing and network deployment

The training part of this framework has been developed using Python (v3.8.3) and the library NumPy. For a better comparison with other approaches (most of them in Table V), the only preprocessing made in the experiments is input data binarization. It is implemented with a simple threshold that allows choosing between a 0/1 input from the initial value. While the input layer has a binarized input, the rest have until five possible input values depending on the activation function, which is different for training and deployment:

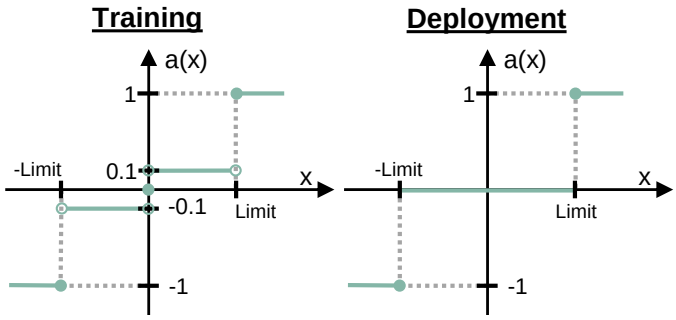

Fig. 2. Activation function used in training (left) and its ternary deployment (right).

1) Training: For training, a simplification of the ternary hyperbolic tangent (tanh) has been selected, as shown in Figure 2. The aim is to consider that the result obtained in each neuron is a sum of +1 and -1 values. As a counter, a certain number of these 1 values, called $Limit$, is needed to trigger a negative or positive value. This $Limit$ can be calculated with equations 5 and 6:

- Equation 5 tries to adapt the difficulty of overtaking the $Limit$ to the number of inputs a neuron has, $Inputs^k$, being 1 the minimum possible value. For example, if the first layer has 784 inputs ($Inputs^0 = 784$) this means that $Limit^0 \approx 6.29$. This equation provides a simple solution that works when the average values of the outputs ($\overline{abs(y^k)}$) far exceed $Limit^k$.

$$Limit^k = \sqrt[4]{Inputs^k} + 1 \tag{5}$$

- An adaptive solution has been created for those cases where $\overline{y}^k$ is strictly low. It will happen when the previous layer has insufficient neurons to propagate high enough results. Thus, the distance between a negative and a positive value must be reduced to achieve a good performance of the training method. To avoid the increment of neurons in these layers, and therefore the required hardware resources, each neuron will have its own adapted $Limit$. This $Limit$ can be calculated with equation 6, which uses a factor of the positive outputs' average ($\overline{y}_+^k$). Positive outputs are used instead of negative ones because they are more restrictive in a one-shot learning approach, where only one output neuron is expected to be positive for each classification. In addition, a configurable parameter λ allows to control the updating rate of $Limit^k$, which is initialized with equation 5:

$$Limit^k = \lambda\ Limit^k + (1-\lambda)\ \frac{\overline{y}_+^k}{4} \tag{6}$$

Both equations are available to be selected by the user, who must analyze and decide which one fits the application under study better. Furthermore, the activation function has a small step between the upper and the lower $Limit$ (see Figure 2).

Its value is another configurable parameter chosen to be 0.1 after a deep exploration during the experiments. It helps the network by providing some flexibility for the training, while its value continues close to 0.

Finally, the derivative approximation of the activation function, a', is expressed as *how much change is wanted in each function interval*. Despite having three intervals, it has been placed an additional one in the middle only as a step between extremes, which improves the training. That means that an output in this interval will have a large derivative approximation associated, allowing the neuron to move its output quicker to one of the extremes. At the same time, the extremes need a smaller a' to decrease the chance of leaving this state. In other words, the key is the ratio between both derivatives (without forgetting a correct setting for the learning rate). Giving 1 to the middle interval, a ratio in the range of 10-100 presents good performance in the experiments, remaining a piecewise function like:

$$a'(x) = \begin{cases} 0.05 & \text{if} \quad x \leq -Limit \\ 1 & \text{if} \quad -Limit \leq x \leq Limit \\ 0.05 & \text{if} \quad Limit \leq x \end{cases} \tag{7}$$

2) Deployment: The presented model provides two possibilities for its deployment:

- To change the small step used in the activation function of the training to value 0, getting a ternarized solution (see Figure 2).
- To use a Sign function, in case a binary implementation is preferred.

The last option implicates a slight accuracy reduction ~~for the first option~~, but it allows having a smaller implementation in the FPGA (see section VI).

C. Network optimization

Two optimization techniques have been developed to improve the performance of the proposed model:

1) Network size optimization: Pruning is a common approach in deep neural networks to eliminate redundant or negligible neurons/layers [5]. During the training of our models we observed that a high percentage of neurons was always returning the same output. To deal with it in an optimal way, a new pruning algorithm has been used. Most pruning algorithms focus on the neurons that contribute the least to the output results [18] [19]. Instead of eliminating these less significant neurons, our pruning algorithm reabsorbs those neurons that present static outputs. It takes advantage of the MLP structure, keeping the accuracy of the network while the number of neurons goes down. Working over the bias of the next layer, k, a neuron j in layer $k-1$ is deleted by adding or subtracting its respective weights, as is shown in Figure 3.

Similar to other pruning methods, this technique is used only for the hidden layers. During an epoch, if an output keeps the same value more than a given number of times (represented as percentage x in Figure 3), the pruning algorithm removes the neuron. In particular, if you select this percentage x as 100%, this pruning technique can keep all the information

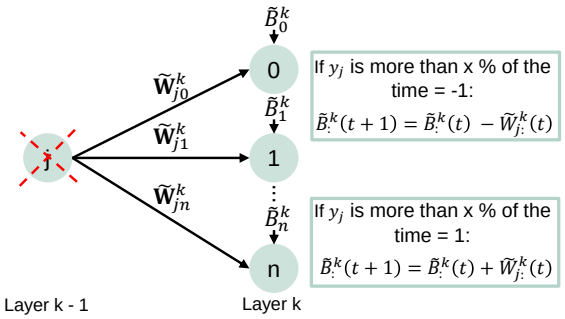

Fig. 3. Proposed pruning method.

given by a neuron in the bias of the following layer, as that means this neuron is always giving the same value as output. Thus, this neuron could be removed without any accuracy loss in the network. The consequent disadvantage is the increase of the bias value, which requires more bits for its representation. However, this growth is insignificant compared to the resources saved when a neuron is removed.

This pruning technique is activated after a few epochs of training, selectable by the user, to give the network enough time to fit. Table III shows the pruning performance results.

2) Training time optimization: The updating technique described in section III-A is a time-consuming bottleneck of the algorithm. To increase the speed, T, W, and $\tilde{W}$ are pre-ordered, sorting first the position of the weights that surpassed their threshold and need to be updated. The algorithm breaks the loop after updating them, without taking care of the rest:

$$P_1^k = (Sign(W^k) \neq Int(\tilde{W}^k)) \tag{8}$$

$$P_2^k = T^k - Abs(W^k)) \tag{9}$$

$$P_{index}^k = argsort(P_1 P_2) \tag{10}$$

P_1^k is a 0/1 matrix where a 1 indicates the possibility of having an update. The negative values obtained in the matrix P_2^k point up which thresholds are surpassed by the absolute value of W^k (with a negative value). P_{index}^k represents the sorted indexes of the original matrix, from the minimum to the maximum. They allow selecting the weights that are going to be updated.

IV. Hardware Architecture

In this section, the proposed hardware architecture is explained. It has been optimized to support a mix between ternary and binary weights while keeping a binarized dataflow.

A. Overall architecture

The design used in this framework aims to reduce the energy needed by the system while keeping the robustness of the typical parallel processing of ANN. For this reason, each neuron has its independent hardware resources. However, there is a shared input for all neurons of the same layer. This input is one bit, changing and arriving to all neurons every clock cycle. It is processed in parallel and stored in each

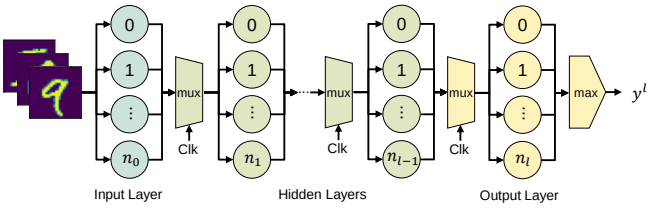

Fig. 4. General structure of the ANN.

neuron's accumulator. A multiplexer selects the input from the previous layer, which a controller manages. The controller allows the layers to work in pipeline, increasing the system's speed. However, the latency is marked by the largest layer: the rest must wait to get their new batch of inputs. Then, a max function block is used at the output to get the classified result. The general structure of the ANN can be seen in Figure 4.

B. Weights and bias extraction

Once the network is trained in software, the learned values are stored in text files (one per neuron). These files could be easily read by a synthesis tool (Vivado in our case [20]), and then be implemented in different ROMs (Read Only Memories). Important optimization decisions have been made:

1) Bias initialization: Instead of having an adder/subtractor in each neuron, only one of these operations is needed if we apply a specific initialization. The idea is to increase or decrease the bias of each neuron, taking into account the number of positive (or negative) values that it receives. Let's call Nz_i^k the number of weights that are zeros in neuron i from layer k. If the number of expected inputs is $Inputs^{k-1}$, and the inputs have only a binary value, each bias could be updated as:

$$\tilde{B}i^k(t+1) = \tilde{B}i^k(t) + (Inputs^{k-1} - Nz_i^k) \qquad (11)$$

This method implies that all neurons are initialized as if all their non-zero inputs were positive (+1). Then, the neuron needs just a subtractor, as only negative values will arrive from the "multiplier" block. A subtractor has been chosen instead of an adder because of a better performance of the pruning algorithm. Normally the bias tends to get negative values after pruning. To compensate it, the possible positive inputs are added, avoiding the increase of bits needed for their memory. I.e.: if the final bias of a neuron is -56 and its previous layer has 100 neurons, it is initialized to 44 (7 bits required) instead of -156 (9 bits required).

2) Input weights: Thanks to the previous optimization, only 1 bit is required for the weights of the input layer. This bit will distinguish between negative or non-negative weights (equation 12). Then, since the input to this layer has a 0 or 1 value, only when input and weight are 1 it is propagated to the subtractor (an AND operation is used instead of an XOR).

$$\tilde{W}ji^0(t+1) = \begin{cases} 1 & \text{if } \tilde{W}ji^0(t) = -1 \\ 0 & otherwise \end{cases} \qquad (12)$$

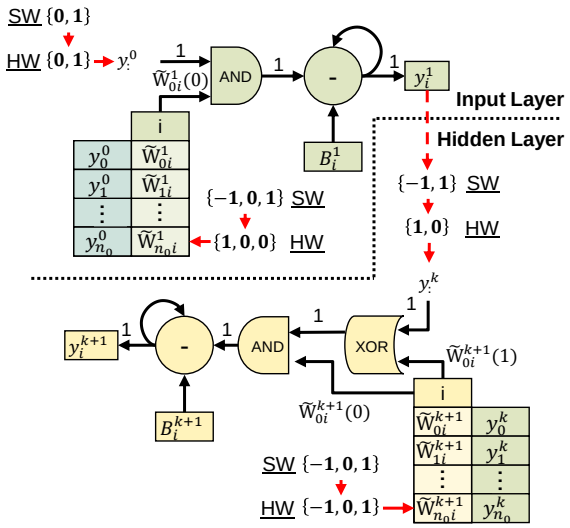

Fig. 5. General structure of a neuron i. Above: Input layer's structure. Below: Hidden layers' structure.

C. Layer and neuron structure

There are some differences between the input and the hidden layers, represented in Figure 5:

- The input layer does not need a multiplexer as only 1 bit (1 pixel) arrives every clock cycle. For the hidden layers, the multiplexers connect the output registers of each neuron with the following layer. The result of each neuron is stored in those registers.
- Section V shows that the "multiplier block" at the input layer is an AND gate, while an XOR gate is needed for the hidden layers. It is in charge of studying the input sign and the weight. The next step, an AND gate allows negative values to arrive at the subtractor.
- Both types of layers use 1 bit as input. Figure 5 shows the neurons' structure with corresponding software (SW) and hardware (HW) values. The same happens with the weights. Furthermore, input weights are 1-bit encoded while the hidden layers are 2-bit encoded.
- The number of bits used at the subtractors and the bias is the same in each layer. After pruning, each bias becomes an integer value. Due to this, each layer is created with the minimum amount of bits needed to represent their maximum. It could be optimized by customizing each subtractor to its corresponding bias. However, the gain is low: it was observed that for each layer the bias ends up with values in the same range of bits. Configurability was preferred with respect to the small improvement that it could bring to the proposed architecture.

It is important to notice that only the sign is propagated within the neuron. Furthermore, all positive or zero operations are already done with the proposed bias initialization. That allows to reduce the power consumption at the subtractor: On the one hand, fewer operations need to be made. On the other hand, the subtractor's output will change only two times per

frame: with the reset (taking the bias sign), and when the accumulator value becomes negative. That is important, as in HW design a large percentage of the power consumption is due to the switching activity.

Both the proposed model and the hardware architecture are managed by a framework, which is introduced next.

V. Framework

The key goal of the framework is to train and optimize the network in Python and automatically generate its FPGA implementation using Xilinx Vivado (v.2020). To achieve it, the process is summarized in six steps: (1) Selection of the network's parameters by the user, (2) hardware resources estimation, (3) training of the network, (4) weights and bias extraction, (5) generation of VHDL (VHSIC Hardware Description Language) code, and finally, (6) hardware implementation. Most of these steps have already been explained in the previous sections. However, (1), (2), and (5) are detailed here. The flow graph of this framework is represented in Figure 6:

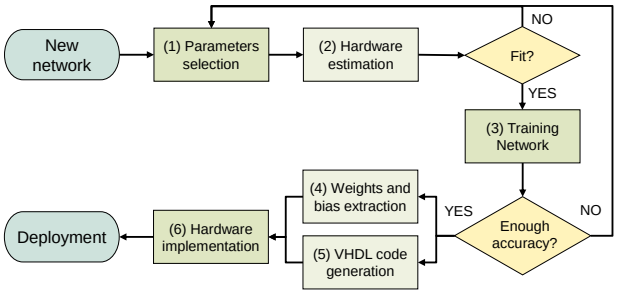

Fig. 6. Framework's flow graph.

A. Parameters' selection

The following list summarized the configurable parameters available in the framework:

- The number of layers and neurons in each layer.
- The number of epochs and the learning rate.
- The initial values of $\tilde{W}$, $\tilde{B}$ and T (Section III-A).
- The step value of the activation function (Figure 2).
- The number of epochs before applying pruning (Section III-C1).

B. Hardware estimator

An estimator has been created for the framework, which allows performing a hardware design space exploration before the model's training. Based on the user input configuration for the ANN, it returns the expected resources that the implementation will need. Thanks to it, the user could modify the network before its implementation when the expected resources exceed those available in the FPGA. Only LUTs (Look Up Tables) and FF (flip-flops) are employed for the implementation. The ROMs with the weights are the biggest LUTs consumers, followed by the subtractor. They are estimated in each layer k as follows:

$$ROM^k_{LUTs} = (0.017\ Inputs^k\ bits_{\tilde{W}^k} + 3)\ Neurons^k \quad (13)$$

$$Subtractor^k_{LUTs} = \frac{bits_{\tilde{B}^k} + 1}{2} Neurons^k \quad (14)$$

$$Subtractor^k_{FF} = (bits_{\tilde{B}^k} + 1) Neurons^k \quad (15)$$

where $bits_{\tilde{B}^k}$ (or $bits_{\tilde{W}^k}$) indicates the number of bits used to represent the biases (or weights) at layer k. I.e.: $bits_{\tilde{W}^0} = 1$. Equation 15 provides the estimation of the FFs, which are used mainly by the accumulator and the neuron's output.

These three equations provide an approximation of the model. They have been extracted after an experimental process on a PYNQ-Z1 board. First, several setups (with random values for the weights, bias, and neurons per layer) were compiled and analyzed from a regression perspective. Then, the regression equations have been biased to cover these experimental values with an overestimation criterion, preventing overflow on the board. This has been done without considering outliers, to keep a good correlation with the actual values. After that, the estimator was validated by comparing its outputs with actual implementation results. This estimator does not surpass a 10% of error with respect to the real values during the experiments (as shown in Table II, Column 8). Furthermore, the user can ensure that the FPGA will have enough resources for the implementation if the estimation is made before the reduction during the training by the pruning method. Finally, this estimation procedure has been optimized for deep networks (at least one hidden layer), getting worse results when the architecture is only made of the input layer.

C. VHDL code generation

Although most VHDL code done for the implementation is configurable, some parts, such the arrays that connect each layer or the activation functions, have to be adapted for each network. To achieve it, a general network template has been defined. This template contains the VHDL definition of the different layers and HW blocks used in the architecture. With it, the framework generates the VHDL files used by Vivado, taking into account the new configuration selected by the user.

D. Deployment

We have tested the different implementations using the standard Jupyter interface provided by PYNQ. This interface allows having the training, the code generation for the HW implementation, and its deployment in Python. The framework calculates the accuracy percentage for validation once the output results are received from the FPGA. The advantage of our proposal is that the same operations are made in SW and HW. The framework changes the dataflow and the data representation, but finally, it returns the same result for both. That means there is no loss in accuracy when an ANN is implemented in HW concerning the corresponding SW.

VI. Experiments

This section shows the performance of the proposed method and architecture with different implementations. The framework has been tested using the MNIST [1], the Fashion MNIST [21], and the EMNIST [22] datasets. All the resources and implementation features correspond to the ZYNQ XC7Z020 FPGA included in the PYNQ-Z1 board.

A. *Experimental settings*

The experiments demonstrate the ability of the proposed architecture to achieve the state of the art accuracy with low resource utilization, not making use of scarce resources like BRAM or DSPs. Thus, the goal of our experiments is to show that the proposed architecture can to reduce hardware resources and power consumption with a minimum decrease in accuracy, which are key desirable features for IoT applications. In contrast to other proposals that target maximizing processing speed, our proposed architecture can work at 159.240 kfps. However, real IoT applications are usually limited by the sensor' speed, where a typical frame rate could be between 30 - 60 fps for cameras or Lidars.

The main issue of binarized MLPs is their limitation to deal with complex problems. However, MLPs approaches could stand out in efficiency and power consumption, as could be the case of Facebook for small tasks such as ads, news feed, and search [23]. Furthermore, novel sensors with ultra-low power consumption combine with MLP approaches to perform efficient tasks at the Edge. For example, in [24] ultrasound sensors are used to detect and recognize gestures, achieving between 84.18% and 92.87% accuracies with an MLP. Despite nowadays many approaches are moving towards more powerful ANNs topologies as CNNs (which sometimes still require some MLP's layers as output), applications such as the ones mentioned above get more profit on improving the efficiency of their deployment.

Thus, the framework has been validated with different datasets to demonstrate its low power consumption and resource-consuming capabilities. MNIST, Fashion MNIST, and EMNIST datasets have been selected because of their complexity, suitable for an MLP. Furthermore, most related works used MNIST as the benchmark, moving directly to CNNs for other datasets.

MNIST contains 28x28 grayscale images of handwritten digits (0 to 9). It has a set 60K images for training and another 10K for testing. Fashion MNIST is similar to MNIST, but it seeks to be a more significant challenge for the algorithms. It changes the digit images to 10 different types of clothes (T-shirts, trousers, sandals, etc.). It maintains the same dimension for the images at each set. In the same way, Letters EMNIST extends MNIST with 26 handwritten letters. It doubles the number of possible classification classes and the images for training, 124,8K, and for testing, 20,8K. These datasets have been tested with two different activation functions, whose accuracies converge with the increase of the network's size. As they use the same training, the comparison is made with the same trained weights:

- BNN: This topology keeps the ternary weights of the training (it is not a fully-binarized network). However, it uses the sign as activation function, which means only 1 bit is needed at the output. It is the one used for the HW implementations (Figure 5), as there are no significant differences with the ternary accuracy results while it presents a big improvement regarding resource utilization.
- TNN: It uses a ternary activation function similar to the one used in training, shown in Figure 2.

For the training, the dynamic threshold T is initialized to 1 (any integer value is allowed). The learning rate is set to 0.05, and 40 epochs are used to achieve the final test result. Our method does not use batch normalization prior to the activation function, as no significant improvement has been shown by its addition. The weights are initialized with a normal random distribution between -0.025 and 0.025 (during the training they will be gradually converted to their ternary version, as explained in Section III). These parameters' initialization is used for the experiments in Table II.

Additionally, equation 5 is used for MNIST and Fashion MNIST. However, its performance decreases with Letters EMNIST, where the experiments showed a reduced positive excitation of the neurons with respect to the previous datasets. For that reason, equation 6 was created (see Section III-B). This equation leads to improving the training results for Letters EMNIST, while it keeps working similarly to equation 5 for the others datasets.

TABLE II
PROPOSAL PERFORMANCE.

Dataset	Initial size	Final size	Hidden Layers reduction (%)	Bias' bits per layer	LUTs	FF	Estimation error (%)	Accurancy BNN (%)	Accurancy TNN (%)
MNIST	784-10	784-10	0	10	212	206	32.72	80.33	83.36
	784-100-10	784-76-10	24	11-7	1,862	1,046	5.99	94.32	95.34
	784-200-10	784-149-10	26	11-8	3,613	1,983	2.31	95.53	96.23
	784-400-10	784-255-10	36	11-9	5,550	3,072	2.16	95.83	96.48
	784-250-250-250-10	784-235-130-48-10	45	11-9-9-6	8,430	4,841	3.93	96.37	96.67
Fashion MNIST	784-10	784-10	0	10	218	207	32	77.53	79.84
	784-100-10	784-67-10	33	11-7	1,695	941	5.08	84.89	85.96
	784-200-10	784-79-10	61	11-8	1,975	1,086	5.21	85.73	86.50
	784-400-10	784-213-10	47	11-9	5,167	2,794	1.71	85.89	86.54
	784-250-250-250-10	784-132-109-111-10	53	11-9-8-8	5,973	3,811	4.67	85.50	86.18
Letters EMNIST	784-26	784-26	0	11	567	318	11.77	49.60	54.41
	784-100-26	784-76-26	24	11-6	1940	1156	5.18	68.24	72.56
	784-200-26	784-154-26	23	11-8	3881	2196	1.76	75.24	78.45
	784-400-26	784-313-26	22	11-10	7497	4254	1.94	80.14	83.04
	784-250-250-250-26	784-232-202-54-26	35	11-9-9-6	9775	5703	3.49	79.32	82.49

B. Results and discussion

The performance of our proposal is summarized in Table II, with several experiments in each dataset to show how the pruning procedure affects the final ANN size and its implementation features. In Column 2 the initial sizes of the ANN to be trained are given for each dataset, MNIST, Fashion MNIST, and Letters EMNIST. Then, Columns 3 and 4 list the pruning results: final network architecture and percentage of reduction in model's size, respectively. The remaining columns 5, 6 and 7 show the number of bits used in each layer for bias and subtractors, LUTs, and FFs. The estimation error in Column 8 represents the mean relative error between the implemented resources (LUTs and FFs from the previous columns) and the ones obtained through the HW estimator (using the equations explained in section V-B). For deep implementations, excellent results were achieved since the error does not surpass the 6%. The last columns describe the ANN accuracy for the Binary (BNN) and Ternary (TNN) representation.

A clock of 125 MHz is used for these experiments, which corresponds to the maximum frequency of the slowest implementation. It has been set as a fixed parameter to improve the comparison between the performed experiments. However, this frequency can be particularly optimized for each implementation, as done in Tables IV and VII.

The following subsections focus on particular aspects of these results, extended with Tables III and IV.

TABLE III
PRUNING PERFORMANCE.

Dataset	Percentage x (%)	Accuracy (%) (20 epochs)	Accuracy (%) (40 epochs)	Final size
MNIST	100	95.30	95.57	784-168-10
	99	95.25	95.56	784-156-10
	95	95.14	95.53	784-149-10
	90	94.91	95.17	784-108-10
Fashion MNIST	100	84.85	85.95	784-151-10
	99	83.93	85.76	784-112-10
	95	84.48	85.73	784-79-10
	90	83.74	85.62	784-67-10
Letters EMNIST	100	74.48	75.66	784-187-26
	99	74.48	75.19	784-173-26
	95	74.42	75.24	784-154-26
	90	73.00	75.08	784-146-26

1) Network size optimization: The experiments in Table II achieve between 22% and 60.5% of reduction at the hidden layers. For these experiments, the pruning technique is performed after 20 epochs of training to allow the ANN enough time to ternarize most weights. Despite having the same initial size, the results change between each dataset, providing better results for Fashion MNIST.

These results depend on the training execution and the selected pruning percentage value x (see section III-C1). Some examples of the relation between this percentage x (in the range 90 - 100 %) and the pruning technique's performance are shown in Table III. Keeping all the configurable parameters static and with an initial model size of 784-200-10, the pruning is applied from 20 epochs of training to 20 epochs more (until 40 epochs). The accuracy post-pruning does not change in the case x = 100 % (used as a reference). Besides it removes between 13 and 49 neurons in each dataset. An accuracy loss can be observed for other values of x after the first pruning (column 3). However, these differences will be decreased with the training (column 4). The model's final size reduction is between 16 % - 46% for MNIST, 24.5 % - 66% for Fashion MNIST, and 6.5 % - 27% for Letters EMNIST. The pruning performance decreases significantly in Letters EMNIST due to the higher number of possible classifications, while the same layer structure is used to better compare datasets.

Therefore, the observed accuracy loss is relatively low compared to the achieved reduction in the hidden layers. A percentage of x = 95 % has been used in Table II, as it keeps a good trade-off for all tested models.

2) MLP baseline: To visualize the improvement of our quantized proposal with respect to a full-precision solution, Table IV compiles three different experiments for each dataset, based on using a hidden layer with 200 neurons:

- Full-precision (16/16 in Table IV): These experiments implement an MLP with 16 precision bits (for inputs, weights, and activation function). They use the same structure and dataflow as Figure 4, equations 1-4 for training, and the ReLU as activation function. The ReLU was selected instead of the Sign to take advantage of its higher number of precision bits while keeping a similar implementation in the FPGA (a multiplexer driven by the sign bit).

TABLE IV
MLP BASELINE.

Dataset	Precision bits (W/A)	Activation function	Network size	Clock (MHz)	LUTs		FFs		DSP		Power (W)	Throughput (kfps)	Efficiency (kfps/W)	Accurancy (%)
					Total	%	Total	%	Total	%				
MNIST	16/16	ReLU	784-200-10	32	44,335	83	8,636	8	210	95	0.265	39.81	150.02	98.26
	8/8				24,101	45	5,409	5	0	0	0.151	39.81	263.64	98.12
	2/1	Sign	784-149-10	167	3,615	7	1,984	2	0	0	0.120	212.31	1,769.29	95.53
Fashion MNIST	16/16	ReLU	784-200-10	32	42,246	79	8,670	8	210	95	0.264	39.81	150.80	88.92
	8/8				23,134	43	5,303	5	0	0	0.124	39.81	321.05	88.73
	2/1	Sign	784-79-10	125	1,975	4	1,086	1	0	0	0.113	159.24	1,409.17	85.73
Letters EMNIST	16/16	ReLU	784-200-26	32	44,350	83	8,673	8	210	95	0.264	39.81	150.80	89.23
	8/8				24,205	45	5,375	5	0	0	0.124	39.81	321.05	88.31
	2/1	Sign	784-154-26	167	3,643	7	2,045	2	0	0	0.118	212.31	1,799.27	75.24

- Half-precision (8/8 in Table IV): The same as Full-precision but with 8 precision bits.
- This proposal (2/1 in Table IV): Using the experiments from Table II (BNN topology with Sign as activation function).

The experiments have been implemented in the PYNQ-Z1 board using the maximum possible frequency. Thus, while the experiments with 16 and 8 precision bits do not surpass a frequency of 32 MHz, our proposal lets to magnify it by four (125 - 167 MHz). This improvement is due to shorter interconnections and reduced complexity of operations. As mentioned before, it is appreciated that for MNIST and Letters EMNIST the frequency has been optimized with respect to Table II.

Considering that the PYNQ-Z1 has 53,200 LUTs, 106,400 FFs and 220 DSPs, columns 6-8 show the absolute value and the utilization percentage of the required resources for each experiment. The experiments with 16 precision bits need the use of DSPs due to the lack of enough LUTs (their implementation would require more than 100,000 LUTs, doubling the resources available on the board). Furthermore, these experiments used most available LUTs (∼80%) and DSPs (∼95%) on this board. In contrast, the implementation of our proposal requires minimum usage of the available LUTs (∼7%).

As is shown in the last column of Table IV, the accuracy achieved by the proposed method is close to the vanilla MLP (1-3% of difference [21]) for MNIST and Fashion MNIST, while the distance increases until ∼10% for Letters EMNIST. At the expense of this accuracy loss, our proposal outperforms the results obtained in power, throughput, and therefore, efficiency (columns 9-11). As can be observed, its efficiency (kfps/W) is between 4 and 12 times higher than the full and half-precision experiments. These results indicate that it is possible to use our model to deal with different problems, achieving a reduced and efficient deployment of the networks suitable for the Edge.

C. Comparison to related work

To fair compare our model with other techniques, Tables V and VI are made with other approaches also based on the MLP structure. Table V summarizes the main features used in each proposed model, while Table VI provides (if available) their FPGAs implementations results. Nevertheless, we have not found any other work which has published binarized MLP results for FPGA implementation on Fashion MNIST and Letters EMNIST. Two experiments are selected from Table II for further analysis in each dataset: The one with an initial size of 200 neurons (it reaches accuracy enough to be a good option) and the one with three hidden layers (network with a size similar to the rest of the Table).

Regarding resources, the lowest LUTs' utilization in Table VI is obtained in [15] by M. Blott et al. Using the same board proposed in this paper, their implementation requires 25,358 LUTs, achieving an accuracy of 97.69%. Comparing it with the best result in Table II (96.37%), it can be observed a difference of 1.32% in accuracy. However, three times fewer resources are needed with our model, showing a reduction of 66,76% in LUTs between both designs. In addition, no BRAMs or DSPs are required, which remain available for other possible implementations.

The experiments depicted in Table II were performed with power consumption between 0.11-0.13 W and a clock cycle of 125 MHz. With this frequency, the model achieved a throughput of 159.24 kfps when the maximum number of

TABLE V
METHODOLOGY AND ACCURACY COMPARISON.

Dataset	Model	Precision bits (W/A)	Optimizer	Activation function	Output	Network size	Accurancy (%)
MNIST	BNN [12]	1/1	Adam	Sign	L2-SVM	784-4096-4096-4096-10	99.04
	FP-BNN [7]	1/1	Adam	Htanh	-	784-1024-1024-1024-10	98.24
	FINN[8]	1/1	-	Sign	-	784-256-256-256-10	95.83
						784-1024-1024-1024-10	98.40
	FINN-R [15]	1/1	-	Sign	-	784-1024-1024-1024-10	97.69
	hls4ml [25]	1/1	-	Tanh	-	784-128-128-128-10	93.00
		2/2				784-128-128-128-10	95.00
	RebNet [16]	1/1	SGD	ReLU	Softmax	784-256-256-256-10	97.87
	STBNN [26]	1/1	SGD	ReLU	-	784-1000-1000-10	98.00
	TNN [14]	2/2	SGD	-	L2-SVM	784-750-750-750-10	98.33
	Park and Sung [27]	8/3	SGD	Sigmoid	-	784-1022-1022-1022-10	-
	Probabilistic [28]	32/32	Adam	ReLU	Linear	784-1000-1000-10	98.32
	Fashion [21]	32/32	-	ReLU	-	784-100-10	97.20
	This work	2/1	SGD	Sign	Max	784-149-10	95.53
						784-235-130-48-10	96.37
Fashion MNIST	STBNN [26]	1/1	SGD	ReLU	-	784-1000-1000-10	87.00
	Probabilistic [28]	32/32	Adam	ReLU	Linear	784-1000-1000-10	90.81
	Fashion [21]	32/32	-	ReLU	-	784-100-10	88.71
	This work	2/1	SGD	Sign	Max	784-79-10	85.73
						784-132-109-111-10	85.50
Letters EMNIST	OPIUM [22]	32/32	-	-	-	784-10,000-26	85.15
	MLP [29]	32/32	SGD	Sigmoid	-	1,024-100-26	89.47
	This work	2/1	SGD	Sign	Max	784-154-26	75.24
						784-232-202-54-26	79.32

TABLE VI
HARDWARE IMPLEMENTATIONS COMPARISON.

Dataset	Model	Platform	Clock (MHz)	LUTs	DSPs	ALM	BRAM	FFs	Latency (us)	Power (W)	Throughput (kfps)	Efficiency (kfps/W)	Accuracy (%)
MNIST	FP-BNN [7]	Stratix-V	150	0	20	182,301	2,210	130,237	-	26.2	-	-	98.24
	FINN [8]	ZC706	200	91,131	0	0	4.5	0	0.31	21.2	12,361	583.07	95.83
				82,988	0	0	396	0	2.44	22.6	1,561	69.07	98.40
	FINN-R [15]	Ultra96	300	38,205	0	0	417	0	-	11.8	∼851.67[a]	∼72.18	97.69
		PYNQ-Z1	100	25,358	0	0	220	0	-	2.5	∼162.33	∼64.9	97.69
	hls4ml [25]	Virtex-9+	200	212,804	0	0	346	165,514	0.2	-	-	-	93.00
				260,093	0	0	346	165,514	0.19	-	-	-	95.00
	RebNet [16]	Spartan XC7S50	200	32,600	150	0	120	65,200	-	-	640	-	97.87
	TNN [14]	Sakura-X	200	-	-	-	-	-	20.5	3.8	195	51.31	98.33
	Park and Sung [27]	ZC706	172	124,862	0	0	323	130,237	-	4.98	70,4	14.14	-
	This work	PYNQ-Z1	125	3,613	0	0	0	1,983	12.56	0.116	159.24	1372.72	96.23
				8,430	0	0	0	4,841	25.12	0.125	159.24	1274.89	96.67
Fashion MNIST	**This work**	PYNQ-Z1	125	1,975	0	0	0	1,086	12.56	0.113	159.24	1409.17	86.50
				5,973	0	0	0	3811	25.12	0.122	159.24	1,305.21	86.18
Letters EMNIST	**This work**	PYNQ-Z1	125	3,881	0	0	0	2,196	12.56	0.114	159.24	1,397.84	78.45
				9,775	0	0	0	5,703	25.12	0.126	159.24	1,263.81	82.49

[a]Values with ∼ are estimated by the authors.

TABLE VII
IMPLEMENTATION OF MNIST IN DIFFERENT PLATFORMS.

Platform	Clock (MHz)	LUTs	FF	Power (W)	Throughput (kfps)	Efficiency (kfps/W)
PYNQ-Z1	167	3,615	1,984	0.120	212.31	1769.29
ZC706	300	3,614	1,991	0.231	424.63	2,011.39
Ultra96	500	3,617	1,993	0.271	636.94	2,350.33

neurons per layer is 784. Aggregating both features, we achieve an efficiency between 1,225-1,448 kfps/W, doubling the best result in Table VI [8]. This efficiency comes after focusing on achieving minimal power consumption, which is 21 times lower than the smallest in Table VI (obtained in [15]).

Additionally to Table VI, Table VII provides a comparison of the MNIST experiment with an initial size of 200 neurons implemented in three different platforms: the PYNQ-Z1, the ZC706 (used in [8] and [27]), and the Ultra96 (used in [15]). To take the most of their particular features, we have set the clock frequency to the maximum allowed for each platform for this specific experiment. While the hardware resources required by this small example do not exhibit significant changes among the three platforms, the results in power and throughput lead to modifying the final efficiency. These results prove that the hardware estimator (section V-B) can be applied to different platforms despite being based on the PYNQ-Z1. This platform also presents the worst performance in efficiency (kfps/W). Nevertheless, for Edge applications, we obtained throughputs that surpassed the ones of the possible input flow, as discussed in Section VI-A. From this perspective, the lower power of PYNQ-Z1 will be preferable, which stands out with the best relation hardware resources / power consumption.

To sum up, the main differences found between our work and related works are:

- We have shown our proposal performance in three different datasets. Thus, we can conclude that the results achieved on MNIST cannot be extrapolated to more complex scenarios, as is Letters EMNIST. Furthermore, the lack of additional datasets from other approaches complicates measuring the actual performance difference between models.
- Batch normalization, a technique typically used for BNN, is not required before the activation function. However, most works in Table V use it.
- This work focuses on having independent HW for each neuron (increasing robustness) with the lowest possible resources utilization. In general, research in this field has mainly focused on throughput, targeting acceleration instead of resource and energy-constrained devices.
- The proposed pruning method effectively reduces the network size and, therefore, to automatically optimizes the final hardware implementation. To the best of our knowledge, no other approach has tackled both problems simultaneously.
- Although achieving a slightly lower accuracy than other works, the proposed approach stands out from the rest in resource utilization, energy consumption, and efficiency. Furthermore, no BRAMs are used, nor DSPs.

To conclude with this analysis, and based on the information provided from Tables II, III, IV, V, VI, and VII, the proposed framework, thanks to the chosen ternarization and pruning approaches, has demonstrated to be highly qualified as a sustainable solution for small device applications, where power consumption and resource utilization are the biggest constraints.

VII. CONCLUSION

This paper proposes a framework for FPGA implementation of ternarized ANNs, using a dynamic threshold parameter estimation. The tool automatically provides an implementable hardware model, and optimizes resources using pruning techniques. In contrast with previous works in the literature, the proposed framework can drastically improve hardware resource requirements for implementation and reduce power

consumption, which is 21 times lower than the best previous result. Both lead to reach a power efficiency in the range of 1,225-1,448 kfps/W, making it suitable for IoT and Edge computing applications. Future work will explore resource sharing between neurons, looking for a trade-off between performance, power and resources through a new balance parameter.

ACKNOWLEDGMENT

This work was partially performed in the ADACORSA project which has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876019. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Netherlands, Austria, Romania, France, Sweden, Cyprus, Greece, Lithuania, Portugal, Italy, Finland, and Turkey.

The authors would like to thanks Infineon Technologies for its support by contracts OTRI - 3770-05 and OTRI - 3770-04.

M. Lopez-Vallejo was funded by the Spanish Ministry of Science and Innovation through the NEUROWARE grant (PGC2018-097339).

REFERENCES

[1] Y. LeCun, "The MNIST database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[2] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[4] A. Page, A. Jafari, C. Shea, and T. Mohsenin, "Sparcnet: A hardware accelerator for efficient deployment of sparse convolutional networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–32, 2017.

[5] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[6] A. Rozantsev, M. Salzmann, and P. Fua, "Beyond sharing weights for deep domain adaptation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 4, pp. 801–814, 2018.

[7] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231217315655

[8] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[9] X. Sun, S. Yin, X. Peng, R. Liu, J. Seo, and S. Yu, "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1423–1428.

[10] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 5–14.

[11] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural computation*, vol. 2, no. 2, pp. 198–209, 1990.

[12] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016, pp. 4114–4122.

[13] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural computing and applications*, vol. 32, no. 4, pp. 1109–1139, 2020.

[14] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," in *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 2547–2554.

[15] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[16] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "ReBNet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 57–64.

[17] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[18] M. Augasta and T. Kathirvalavakumar, "Pruning algorithms of neural networks—a comparative study," *Open Computer Science*, vol. 3, no. 3, pp. 105–115, 2013.

[19] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *arXiv preprint arXiv:2003.03033*, 2020.

[20] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.

[21] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[22] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "Emnist: an extension of mnist to handwritten letters," 2017.

[23] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[24] B. Saez, J. Mendez, M. Molina, E. Castillo, M. Pegalajar, and D. P. Morales, "Gesture recognition with ultrasounds and edge computing," *IEEE Access*, vol. 9, pp. 38 999–39 008, 2021.

[25] J. Ngadiuba, V. Loncar, M. Pierini, S. Summers, G. Di Guglielmo, J. Duarte, P. Harris, D. Rankin, S. Jindariani, M. Liu *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 1, p. 015001, 2020.

[26] G. Qiao, S. Hu, T. Chen, L. Rong, N. Ning, Q. Yu, and Y. Liu, "STBNN: Hardware-friendly spatio-temporal binary neural network with high pattern recognition accuracy," *Neurocomputing*, vol. 409, pp. 351–360, 2020.

[27] J. Park and W. Sung, "FPGA based implementation of deep neural networks using on-chip memory only," in *2016 IEEE International conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2016, pp. 1011–1015.

[28] R. Peharz, A. Vergari, K. Stelzner, A. Molina, M. Trapp, K. Kersting, and Z. Ghahramani, "Probabilistic deep learning using random sum-product networks," *arXiv preprint arXiv:1806.01910*, 2018.

[29] A. Botalb, M. Moinuddin, U. M. Al-Saggaf, and S. S. A. Ali, "Contrasting convolutional neural network (cnn) with multi-layer perceptron (mlp) for big data analysis," in *2018 International Conference on Intelligent and Advanced System (ICIAS)*, 2018, pp. 1–5.