Logic Neural Networks for Efficient FPGA Implementation

Iván Ramírez, Francisco J. Garcia-Espinosa, David Concha, and Luis Alberto Aranda

***Abstract*— Logic Neural Networks (LNNs) represent a new paradigm for implementing neural networks in hardware devices such as Field-Programmable Gate Arrays (FPGAs). These network architectures exhibit unique attributes that can leverage the inherent parallelism of FPGAs, enabling the development of networks characterized by low power consumption and fast inference capabilities. Despite their potential advantages, the relative novelty of LNNs poses a challenge, as there are currently no established guidelines for defining their architectures. In this paper, we present a comprehensive study of LNNs, aiming to address the existing gap in understanding and guide decision-making during the design phase. Through systematic experimentation and analysis, we explore various aspects of logic networks, including their impact on inference time, power consumption, and overall simplicity. The findings derived from these experiments provide valuable insights for the creation of improved networks, thereby paving the way for further advancements in this field.**

***Index Terms*— Field-programmable gate array (FPGA), hardware implementation, logic neural network (LNN), network architectures.**

I. Introduction

THE hardware of neural networks is currently designed to offer a maximum number of resources, primarily focusing on enabling a multitude of neurons that may potentially reach into the billions [1], [2]. While the development of neural network-specific hardware accelerators, such as Graphics Processing Units (GPUs) with their specialized tensor cores and Tensor Processing Units (TPUs), has enabled significant progress across a wide range of fields in Artificial Intelligence (AI), the design of these models often disregards the underlying hardware. The prevailing approach is to initially design a neural network and subsequently adapt the hardware platform if necessary [3]. Nowadays, AI attracts significant interest not only from the academic world but, more prominently, from the industry. Companies are eager to leverage neural networks to enhance their production processes effectively; however, a set of constraints arises when putting these models into production, including real-time processing requirements, low power consumption, and simplicity of network architecture. As a result, there is growing interest in developing reduced and efficient neural networks that address these challenges. Approaches to reduce inference costs and improve energy efficiency are being actively explored, including Binary Neural Networks (BNNs), Quantized Neural Networks (QNNs), pruning techniques, knowledge distillation, and low-rank factorization. These methods aim to simplify network architectures while maintaining performance, making them particularly suitable for deployment in resource-constrained environments.

In view of this, we investigate the design of Logic Neural Networks (LNNs), a renewed version of algebraic circuits [4] applied to machine learning tasks and programing logic [5], [6] which substantially differs from previous mentioned approaches. In fact, LNNs are a type of Weightless Neural Networks (WNNs) [7], contributing to a growing body of research that offers alternative models for improved efficiency.

In particular, LNNs leverage logic gates instead of using parameters, which makes them suitable for deployment on Field-Programmable Gate Array (FPGA)-like hardware.

In contrast to existing post-hoc strategies as proposed in frameworks as FINN [8], this co-design approach offers two key advantages due to the utilization of operational components native to the target device:

1) It avoids approximations that often lead to accuracy loss.
2) The resulting model is well-suited to the target device, enhancing its performance.

Additionally, in the specific case of interest, namely FPGAs, the target applications often involve processing tasks in extreme environments, such as space or radioactive settings, where power consumption and available resources are highly constrained, as well as scenarios where high-speed processing is crucial, such as in real-time autonomous vehicle control.

LNNs are constituted of neurons implemented by logic gates, each typically having two inputs and one output. Each neuron is, in fact, one logic gate selected optimally from an available set of gates. Due to the non-differentiable nature of such logic functions, we build upon the work of [9], where the authors proposed a differentiable version of the logic functions and a framework for end-to-end training of an LNN, however, offering only software-based implementations. In this work,

Received 10 July 2024; revised 27 September 2024; accepted 27 October 2024. Date of publication 7 November 2024; date of current version 1 July 2025. This work was supported in part by the Recovery and Resilience Facility Program from the NextGenerationEU Plan of European Union and Spanish Research Agency under Project TED2021-129162B-C22 and in part by Spanish Plan for Scientific and Technical Research and Innovation of the Spanish Research Agency under Project PID2021-128362OB-I00. This article was recommended by Associate Editor Y. Tang. *(Corresponding author: Iván Ramírez.)*

The authors are with the CAPO Research Group, Universidad Rey Juan Carlos, Móstoles, 28933 Madrid, Spain (e-mail: ivan.ramirez@urjc.es; franciscojose.garcia@urjc.es; david.concha@urjc.es; luis.aranda@urjc.es).

Digital Object Identifier 10.1109/TCSI.2024.3488119

we take an additional step by fully implementing a pre-trained differentiable LNN on an FPGA.

The main contributions of our work can be outlined as:

1) We describe how to implement a LNN into an FPGA. To our knowledge, this is the first attempt to accomplish this task.
2) We conducted an ablation study on the set of logic gates used in LNN models, aiming to elucidate their relative importance in achieving high accuracy scores.
3) We performed a series of experiments to bring clarity to the different trade-offs present in developing an LNN.

The paper is organized as follows: this Introduction sets the research problem and objectives. In Section II, existing literature is reviewed. Section III provides context on how LNN are conceived. Sections IV and V detail the main procedure and results of our study, examining both the model and hardware perspectives. Lastly, Sections VI and VII explore the findings and discuss their implications.

II. Related Work

Aligned with the objective of reducing the number of weights in neural networks, efforts have been made to compress these models while maintaining their performance. Some approaches focus on pruning methods [10], where it is crucial to discriminate between relevant and uninformative weights. The pruning techniques can be applied before the training stage [11], [12], during training (online) [13], [14], or after training [15], [16], with the latter being the most commonly used approach. Classic regularization techniques are often included in the optimization of neural networks, as the $L1$-regularization, well-known for promoting a spare set of optimal parameters, which can be seen as part of a pruning technique during training. Quantization of parameters has also been explored, leading to the development of Quantized Neural Networks (QNNs). QNNs offer reduced memory and computational requirements, outperforming pruning methods in many architectures [17], and making them suitable for development on resource-constrained devices, but may suffer from loss of accuracy compared to full-precision networks [18], [19]. Additionally, adapting the training stage to reduced precision has also been investigated to account for this loss of accuracy [20], [21].

Another approach involves Binary Neural Networks (BNNs), where activations and weights are represented as 1-bit values. This extreme compression allows for the reduction of matrix computations in each neural network layer to XNOR operations [22], [23], [24], [25]. BNNs can be seen as an extreme instance of QNNs and, as a consequence, the loss of accuracy is dramatically high. Hence, hybrid models integrating varying levels of quantization across layers have emerged in response to this challenge. A recent, slightly different but successful approach in the field of large language models (LLMs) is presented in [26], where parameters are not binary but ternary $\{-1, 0, 1\}$. Although the attempts with QNNs to simplify neural networks may achieve significant success, their effectiveness is still constrained by the inherent challenge of adapting hardware principles to software implementations.

For such a reason, the approach considered in this paper is based on Logic Neural Networks (LNNs), where, as mentioned previously, artificial multi-input neurons in a network are replaced by two-input logic gates. Unlike QNNs, BNNs and pruning methods, LNNs are weightless networks that consider specific hardware characteristics from the beginning, promoting a co-design strategy. In other words, whereas QNNs and BNNs work to compress standard neural networks, LNNs focus on algebraic circuits, thus, without relying on weights. As a result, they are particularly well-suited for implementation on FPGAs.

Nonetheless, LNNs face a challenge due to the non-differentiability of their logic gates, which inhibits their training with gradient-based algorithms like back-propagration. To address this issue, the authors in [9] introduce a modified differentiable variant of a set of logic gates, enabling end-to-end training of LNNs. In a broad sense, non-differentiability is highly significant in neural network research, especially concerning optimization methods such as gradient descent, which depend on derivative calculations. Extensively studied, this challenge deeply affects model convergence and training efficiency. Different approaches have been investigated to address non-differentiability, ranging from specialized optimization algorithms like subgradient methods, commonly employed in variational calculus, to utilizing differentiable approximations of activation functions to manage the inherent nonlinearities in neural networks. In the latter approach, certain studies utilize the Gumbel-max trick [27]. For instance, in [28], the authors use a Gumbel-softmax approach by introducing noise in the selection process of optimal functions from a given set. In [29], the authors propose a Gumbel-Max-based equation to learn optimal arithmetic functions within layers of neural networks.

The lack of differentiability is also at the root of the limitations of conventional neural networks. Despite their success in approximating tasks, neural networks fail when interpretable predictions are required. Recent efforts in this direction include differentiable deep neural logic networks [30], where the authors propose a new learning paradigm for inductive logic programming, aiming to enhance the explainability of neural network predictions.

III. Background

In this section, we provide a detailed background on Logic Neural Networks and recall the approach proposed in [9] to make them differentiable.

A. Logic Neural Networks

LNNs are essentially Directed Acyclic Graphs (DAGs) where each node, or neuron, operates as a binary logic gate. Although they share a similar structure with conventional neural networks, such as multi-layer perceptrons (MLPs), they differ significantly. In MLPs, multiple inputs are processed using nonlinear activation functions, multiplicative weights, and additive biases.However, in LNNs, inputs to logic gates are reduced to two, and traditional neural network components like activation functions and parameters are absent (see Fig. 1).

Consequently, while neural networks are parametric models capable of approximating any function, LNNs lack parameters

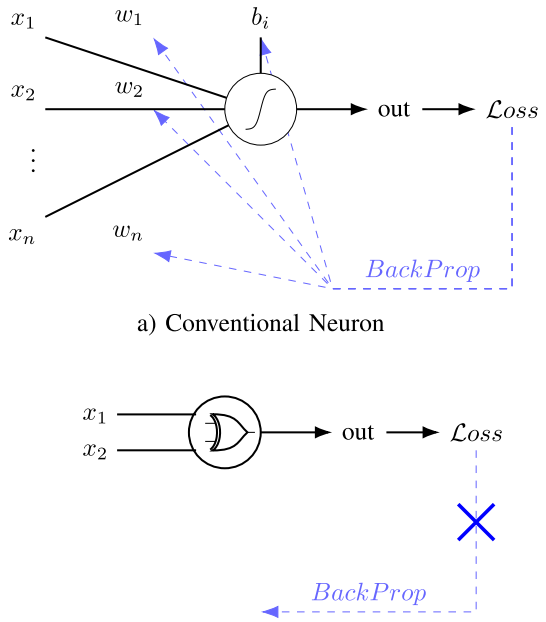

Fig. 1. Visualization of a conventional neuron (a) and a logic neuron (b) interconnected in a neural network. The dashed lines represent the back-propagration process for error minimization during training. Logic neurons differ from conventional ones due to the absence of differentiability and parameters.

to optimize and the logic functions implemented by the gates are non-differentiable. This non-differentiability poses a challenge for gradient-based optimization algorithms like back-propagration, making it unsuitable for direct application to LNNs.

Additionally, neural networks are universal approximators [31], particularly, given a sufficient number of neurons, they can approximate any continuous function up to an arbitrary error. Conversely, LNNs are equipped with a functionally complete set of logical gates, ensuring the ability to express all possible combinations, i.e., truth tables. More formally, while the domain and codomain/image of a neural network are dom(NN), img(NN) $\in \mathbb{R}$, LNNs operate with discrete and binary elements, dom(LNN), img(LNN) $\in \{0, 1\}$. This implies that both the input and output of an LNN are binary variables and, thus, for a given machine learning problem, LNNs must be appropriately adjusted based on whether they are used in a classification or regression task.

B. Differentiable LNNs

The non-differentiability of the logic gates is addressed by replacing them with differentiable functions that approximate their behavior. In [9], the authors suggest employing 16 approximating differential functions to convert binary values into the $[0, 1]$ range. These gates cover all two-input logic functions, each with 4 outputs, making a total of $2^4 = 16$ unique functions.

Furthermore and contrary to a neural network, to optimize the LNN means to find the optimal logic gates for a given neuron and architecture, i.e., to select from the 16 logic gates, the most appropriated to reduce training error. To this end, the output of a logic neuron is determined by calculating the expected value across all potential logic gates: each differentiable logic gate is multiplied by a probability derived from the softmax value of a weight vector. Let f_i be each logic gate's output. Then, the expected value of a logic neuron is given by:

$$\text{out} = \mathbb{E}_{\text{p}}\{f_i | i = 0, \ldots, 15\} \tag{1}$$

where

$$\text{p} = Softmax(\boldsymbol{\omega}) = \frac{e^{\boldsymbol{\omega}}}{\sum_{i=0}^{15} e^{\omega_i}}, \tag{2}$$

is a probability distribution function computed through $Softmax$. Note that, in order to provide a probabilistic interpretation of the relative importance of each differentiable logic gate, different normalizing functions may be considered.

During training, the $\boldsymbol{\omega}$ weights are optimized for a given loss function using back-propagation algorithm (see Fig. 2). Once the LNN is trained, inference is performed by replacing the approximating logic gate with the highest probability by its original logic function.

C. Network Architecture

To establish the final architecture of a LNN, it's necessary to predefine the connections between layers. This is because each node is restricted to having only two inputs, making fully connected LNNs unattainable. The authors, [9], proposed two manners for the selection of gates connections: 1) a random selection and 2) a deterministic algorithm that provides a set of connections for a given number of logic neurons. In this work, we have considered the first selection methods to initialize each logic neural layer, for the same reason given by the authors: simplicity and no significant differences with respect to the second method are found when tested over several datasets common datasets as MNIST.

Furthermore, the authors set 16 logic gates with their differentiable approximations (see Table I), enabling back-propagation during training. As previously mentioned, unlike the standard training of MLPs, training LNNs is significantly more expensive due to the necessity of maintaining 16 logic gates for each neuron. During inference, only one logic gate - with the highest probability p_i - is computed through the actual logic gate (rather than its differentiable version), which substantially alters the final architecture.

IV. Network Architecture Analysis

This section presents the systematic experimentation and analysis conducted to investigate the logic network architecture described in the previous section. Two types of experiments are outlined in this section: 1) a neuron analysis that involves the step-by-step removal of individual logic functions within the network to identify their relative importance; and 2) an architecture analysis focused on varying the number of neurons and layers employed in the architecture. In particular, the architecture analysis is subdivided into three sets of experiments, wherein three distinct architectures are

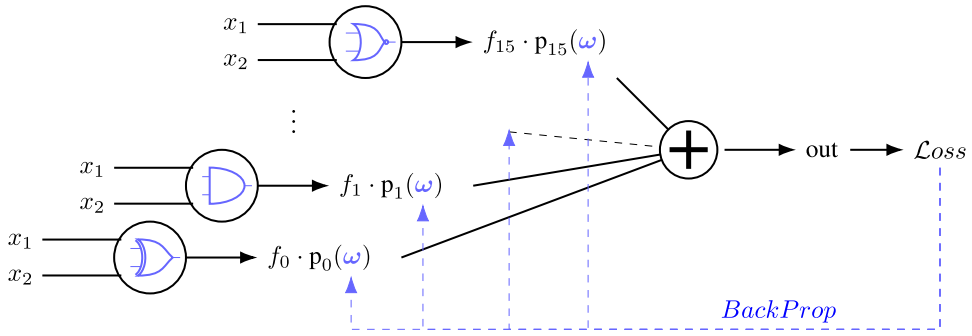

Fig. 2. Diagram depicting a differentiable logic neuron incorporating back-propagration for training. It is noteworthy that, unlike conventional logic neurons, this model enables weight learning allowing selection of differentiable logic gates. Thus, each approximating out is aggregated in a single output that computes the expected value of the neuron. At inference time, only the logic differentiable gate (blue) with highest probability (p_i^*) is replaced by the original one (Fig. 1).

TABLE I
LIST OF 16 REAL-VALUED BINARY LOGIC GATES PROPOSED BY [9]

ID	Logic Gate	Real-valued
0	'0'	0
1	$A \wedge B$	$A \cdot B$
2	$A \wedge \overline{B}$	$A - AB$
3	A	A
4	$\overline{A} \wedge B$	$B - AB$
5	B	B
6	$A \oplus B$	$A + B - 2AB$
7	$A \vee B$	$A + B - AB$
8	$\overline{A \vee B}$	$1 - (A + B - AB)$
9	$\overline{A \oplus B}$	$1 - (A + B - 2AB)$
10	$\overline{B}$	$1 - B$
11	$A \vee \overline{B}$	$1 - B + AB$
12	$\overline{A}$	$1 - A$
13	$\overline{A} \vee B$	$1 - A + AB$
14	$\overline{A \wedge B}$	$1 - AB$
15	'1'	1

tested: the baseline architecture comprising the 16 logic functions outlined by [9], an architecture incorporating half of the original logic functions, and an architecture composed of the two most significant logic functions identified in the neuron analysis.

A. Neuron Analysis

As previously stated, the baseline model for the MNIST dataset consists of 16 logic functions, ranging from a simple NOT operation ($F = \overline{A}$) to a more complex two-input logic function ($F = A \cdot \overline{B}$). The objective of this first study is to systematically eliminate, one by one, the logic functions that contribute the least to the overall test accuracy of the network. In each iteration, the logic function causing the smallest decrease in overall accuracy is removed. Subsequently, the network is retrained using the remaining functions, and the process is repeated until only two functions remain. The procedure stops at this point since the training of the model with only one logic function to choose from would become trivial. The detailed steps of this procedure are outlined in Procedure 1 for clarity.

Following this procedure, the results presented in Table II are obtained.

Procedure 1 Logic Function Removal Procedure

```
model ← init_model()
for i = 1 to N_func − 2 do
    func_to_remove ← find_min_impact(model)
    model.remove_func(func_to_remove)
    model.train()
    update_accuracy(model)
end for
model.display_accuracy_in_each_step()
```

Results in Table II indicate that the NOT, NOR and NAND logic functions are the most significant contributors to the overall accuracy of the network. The removal of these logic functions causes the loss of crucial functionalities within the network. This observation is consistent with the fact that NAND and NOR functions are universal logic gates, meaning that any logic function can be represented using only one of these functions (including the NOT function). From the industrial point-of-view, these findings offer an additional perspective, as an Application-Specific Integrated Circuit (ASIC) containing the architecture of a LNN based solely on NOR gates, for example, could be manufactured. This approach could potentially increase circuit integration, and reduce complexity and costs by using homogeneous gates.

Upon examining in detail the accuracy values highlighted in bold in Table II, it can be observed that the network's accuracy experiences a substantial drop from iteration 13 to iteration 14, specifically by $96.98\% - 96.27\% = 0.71\%$. This drop is equivalent to the cumulative decrease observed in the preceding iterations, amounting to $97.71\% - 96.98\% = 0.73\%$. For better visualization, the variation in model accuracy is depicted in Fig. 3.

In Table II, at iteration 13, the remaining logic functions are NOR, NAND, and NOT. However, by iteration 14, eliminating the NAND function results in the mentioned accuracy drop of 0.71%. This result is interesting and can be explained as follows: if the NAND gate is removed, the NOT and the NOR functions must be utilized to achieve the lost functionality. A NAND gate can be synthesized using three levels of NOR gates, thus necessitating additional layers. Consequently, an architecture comprising NAND and NOR gates would

TABLE II

TEST ACCURACY (%) OF THE LNN AFTER REMOVING EACH LOGIC FUNCTION AT EVERY STEP. IN EACH ITERATION, THE LOGIC FUNCTION CAUSING THE SMALLEST DECREASE IN OVERALL ACCURACY IS REMOVED (IN BOLD). THE INITIAL LNN ACCURACY VALUE IS 97.71 %. SEE PROCEDURE 1

Iter.	'0'	$A \wedge B$	$A \wedge \overline{B}$	A	$\overline{A} \wedge B$	B	$A \oplus B$	$A \vee B$	$\overline{A \vee B}$	$\overline{A \oplus B}$	$\overline{B}$	$A \vee \overline{B}$	$\overline{A}$	$\overline{A} \vee B$	$\overline{A \wedge B}$	'1'
1	**97.63**	97.47	97.49	97.28	97.49	97.35	97.46	97.19	97.33	97.41	97.19	97.22	97.26	97.43	97.52	97.59
2		97.46	97.34	97.21	97.35	97.27	97.29	97.41	97.30	97.41	97.42	97.38	97.43	97.47	97.30	**97.57**
3		97.43	97.34	97.26	97.48	97.37	97.38	**97.56**	97.33	97.33	97.27	97.22	97.39	97.42	97.28	
4		97.48	97.46	97.47	97.40	97.32	97.48		97.42	**97.54**	97.27	97.45	97.34	97.40	97.43	
5		97.15	97.22	97.26	97.49	96.97	97.34		97.15		97.51	**97.60**	97.09	97.34	97.40	
6		97.51	97.41	97.30	**97.54**	97.33	97.28		97.19		97.24		97.16	97.41	97.26	
7		97.33	97.34	97.39		97.35	**97.39**		97.15		97.27		97.27	97.26	97.13	
8		97.22	**97.30**	97.02		97.14			96.85		97.00		97.13	97.05	97.13	
9		97.28		97.19		97.13			96.89		97.24		**97.35**	97.11	97.06	
10		96.90		96.94		**97.23**			96.73		96.78			96.92	97.05	
11		96.95		**97.12**					96.54		96.79			97.03	96.81	
12		96.89							96.44		96.63			**97.02**	96.86	
13		**96.98**							96.20		96.55				96.61	
14									95.55		96.00				**96.27**	

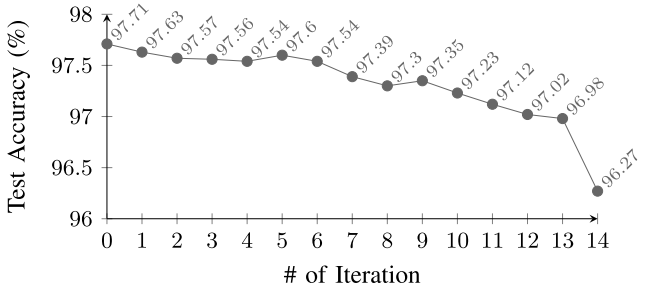

Fig. 3. Model accuracy variation (MNIST). In each iteration, the logic function causing the smallest decrease in overall accuracy is removed.

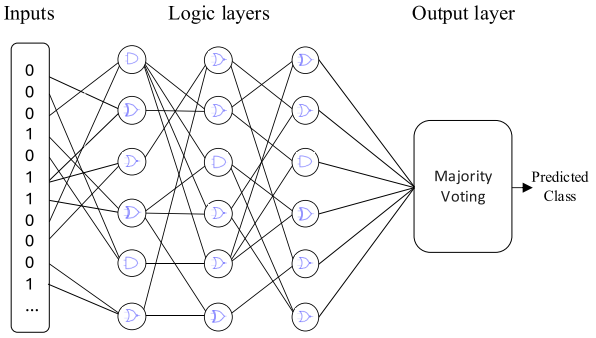

Fig. 4. Architecture of a logic neural network (LNN). The inputs are boolean vectors, with each element pairwise connected to logic gates. Multiple logic layers, potentially containing different numbers of logic neurons, are stacked. The final class prediction is determined through majority voting.

require fewer layers than an architecture composed of NOR and NOT gates, since implementing the remaining NAND function would pose greater difficulty for the network than implementing the NOT function. This observation will be further investigated in the next subsection during the architecture analysis.

B. Architecture Analysis

For the architectural analysis of LNNs, we trained various LNN architectures, systematically increasing the number of layers and neurons per layer to observe trends in network accuracy. A general scheme of the architecture used in all subsequent experiments is illustrated in Figure 4.

Let us first analyze the baseline LNN architecture comprising 16 logic functions. The test accuracy achieved by this model is presented in Table III.

TABLE III

ACCURACY (%) VARIATION WITH THE NUMBER OF LAYERS AND THE NUMBER OF NEURONS PER LAYER (BASELINE MODEL WITH 16 LOGIC FUNCTIONS)

	Neurons per layer			
Layers	2,000	4,000	6,000	8,000
1	86.31	91.37	92.09	93.85
2	90.99	94.46	95.05	96.00
3	92.99	95.65	96.51	96.82
4	93.62	96.05	96.72	97.14
5	94.50	96.14	97.04	97.23
6	94.74	96.61	96.92	97.34

TABLE IV

ACCURACY (%) VARIATION WITH THE NUMBER OF LAYERS AND THE NUMBER OF NEURONS PER LAYER (MODEL WITH 8 LOGIC FUNCTIONS)

	Neurons per layer			
Layers	2,000	4,000	6,000	8,000
1	82.80	89.94	91.55	93.20
2	89.32	93.52	94.57	95.70
3	91.90	94.84	95.74	96.48
4	92.86	95.45	96.27	96.95
5	93.20	95.82	96.70	97.28
6	93.36	95.90	96.86	97.38

As expected, the accuracy of the network increases with both the number of layers and the number of neurons per layer. However, the accuracy value experiences a greater increase when moving horizontally in Table III. This suggests that these logic gate-based architectures tend to prioritize width over depth. To further support this hypothesis, an architecture employing half of the initial 16 logic functions was trained using the same methodology. Table IV displays the results of this architecture.

It can be observed that the accuracy value again increases more noticeably when moving horizontally rather than vertically, suggesting the validity of the hypothesis. This observation aligns with the fact that neurons in the LNN are not fully connected, meaning they only have two inputs. Therefore, it seems that establishing more connections and making wider layers can enhance the network's functionalities.

However, the previous statement does not hold true when only two logic functions are employed in the architecture. Table V illustrates the accuracy results of architectures constructed solely from two logic functions. It can be seen in this table that the network exhibits better performance as the number of layers increases. This is because the network

TABLE V
ACCURACY (%) VARIATION WITH THE NUMBER OF LAYERS AND THE NUMBER OF NEURONS PER LAYER (MODELS WITH 2 LOGIC FUNCTIONS)

Layers	Neurons per layer 2,000	4,000	6,000	8,000
NOR-NOT architecture				
1	41.35	61.92	71.81	74.92
2	74.20	84.03	88.87	91.43
3	82.89	90.63	92.66	94.56
4	85.20	91.36	93.64	95.20
5	89.29	93.96	95.02	96.25
6	88.97	93.88	95.36	96.13
NAND-NOR architecture				
1	55.77	73.33	80.11	84.77
2	79.16	87.50	90.78	93.34
3	85.27	90.75	93.43	95.15
4	85.77	91.23	93.47	95.39
5	89.37	93.59	94.88	95.97
6	88.14	93.11	94.80	96.00

has limited functionalities so, to perform more complex logic operations, more logic gate levels are required. In the previous subsection, NOR and NOT logic functions were identified as the pivotal gates in the network. Nonetheless, it was also stated that an architecture comprising NAND and NOR gates would require fewer layers to achieve a similar accuracy level than an architecture composed of NOR and NOT gates. As can be seen in this table, with three or fewer layers, the NAND-NOR architecture surpasses the NOR-NOT one, but with more than three layers, the accuracy levels are similar, meaning that the lost functionality can be implemented by the network by combining the NOR-NOT gates. When the lost functionality is recovered (more than 3 layers) the width over depth hypothesis holds true again.

Finally, the accuracy variations with the number of layer for the 16, 8, and 2 (NAND-NOR) architectures are depicted together in Fig. 5 to compare trends.

As expected, the accuracy values for the architectures using fewer number of logic functions are usually lower than the other architectures. In fact, it can be seen that the accuracy levels tend to stabilize at 5-6 layers. This means that more neurons per layer are required to increase the overall accuracy of the network. For example, to achieve a 96% of accuracy the 16-functions architecture requires at least 4,000 neurons per layer and 4 layers. However, to achieve this value the 8-functions architecture requires 6,000 neurons per layer, while the 2-functions architecture necessitates 8,000 neurons per layer and 6 layers.

In the next section, we have implemented these designs on a hardware platform to compare resource usage, throughput, and power consumption, offering a different perspective for further discussions.

V. HARDWARE IMPLEMENTATION

The hardware platform used to measure the performance of the designs discussed in the previous section is the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit from Xilinx. The resource usage, throughput, and power consumption values have been estimated using Xilinx Vivado utilization, timing and power reports respectively. The detailed pipeline diagram followed for these experiments are depicted in Figure 6.

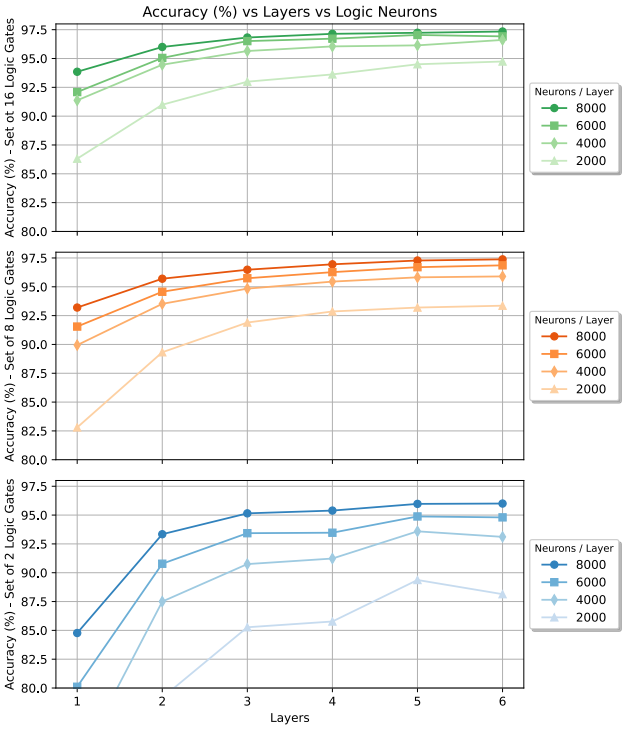

Fig. 5. Zoomed-in view of the accuracy variation with the number of layers for each architecture.

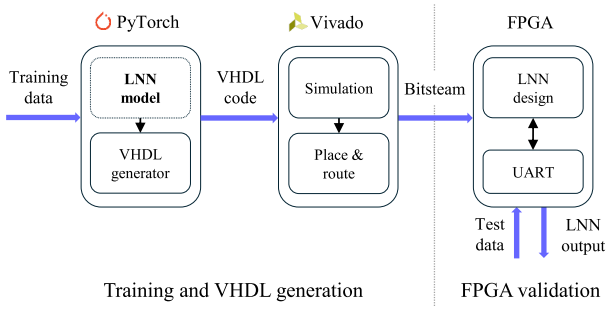

Fig. 6. The proposed pipeline is implemented as follows. First, an LNN model is trained on the training data using the differentiable approach outlined in [9], leveraging the PyTorch framework. Once trained, the optimized, weightless LNN is converted into VHDL code via the VHDL generator. This VHDL code is then processed on the Vivado platform for simulation and bitstream generation. Finally, the bitstream is deployed onto the FPGA, which is then ready for validation on the test data.

The resource usage reports generated for the different architectures indicate that only Look-Up Tables (LUTs) are utilized. This is due to the fact that logic neural networks are pure combinational circuits comprised of logic gates, thereby obviating the need for additional flip-flops, RAM, or digital signal processing blocks. By exclusively utilizing LUTs, LNNs can achieve inference times on the order of nanoseconds. Inference time or throughput for each design has been evaluated by estimating the worst total delay, which is calculated as the sum of the logic delay and the routing delay. Besides, the fact LUTs are used to implement logic gates, as LNNs do not require weights, helps alleviate storage limitations. The number of LUTs and the throughput of each model are summarized in Tables VI, VII, and VIII.

It is noticeable that the LUT count and throughput exhibit slight variations across different models. Interestingly, the model employing two logic functions appears to demand

TABLE VI

LUT COUNT AND THROUGHPUT (IN NANOSECONDS) VARIATIONS WITH THE NUMBER OF LAYERS AND THE NUMBER OF NEURONS PER LAYER (BASELINE MODEL WITH 16 LOGIC FUNCTIONS)

Layers	Neurons per layer 2,000	4,000	6,000	8,000
LUT count				
1	2,735	5,829	8,031	13,525
2	3,582	7,201	11,460	14,275
3	4,008	8,146	11,634	16,092
4	4,081	8,609	13,576	17,338
5	4,451	8,981	14,056	19,385
6	4,940	9,832	15,316	20,542
Throughput (in nanoseconds)				
1	54.5	62.2	65.6	67.0
2	56.9	63.3	67.5	71.7
3	57.0	65.6	71.6	77.2
4	57.7	67.4	76.0	81.7
5	59.1	68.8	78.2	84.3
6	61.9	72.9	81.5	90.7

TABLE VII

LUT COUNT AND THROUGHPUT (IN NANOSECONDS) VARIATIONS WITH THE NUMBER OF LAYERS AND THE NUMBER OF NEURONS PER LAYER (MODEL WITH 8 LOGIC FUNCTIONS)

Layers	Neurons per layer 2,000	4,000	6,000	8,000
LUT count				
1	2,784	5,809	8,023	13,377
2	3,384	6,949	10,735	14,008
3	3,857	7,698	11,858	15,581
4	3,885	8,028	12,697	17,305
5	3,984	8,065	13,335	18,231
6	4,277	8,784	13,628	18,696
Throughput (in nanoseconds)				
1	54.8	59.0	66.6	69.7
2	56.1	67.5	74.5	74.0
3	56.5	67.6	72.6	79.5
4	58.3	66.1	75.9	81.0
5	59.4	70.1	75.1	86.9
6	59.6	71.9	79.7	83.6

TABLE VIII

LUT COUNT AND THROUGHPUT (IN NANOSECONDS) VARIATIONS WITH THE NUMBER OF LAYERS AND THE NUMBER OF NEURONS PER LAYER (MODEL WITH 2 LOGIC FUNCTIONS)

Layers	Neurons per layer 2,000	4,000	6,000	8,000
LUT count				
1	2,639	5,540	8,029	12,411
2	2,717	5,488	8,384	13,474
3	3,419	6,697	11,696	13,545
4	3,851	7,559	12,355	15,714
5	4,037	8,100	12,553	17,480
6	4,372	8,570	13,616	19,452
Throughput (in nanoseconds)				
1	53.3	58.7	65.0	68.4
2	56.5	61.8	65.1	72.2
3	57.9	65.4	70.5	74.2
4	59.1	63.7	70.5	76.5
5	58.1	68.2	75.0	78.7
6	60.2	69.1	76.5	84.1

fewer resources while attaining superior throughput. This phenomenon could be due to the increased efficiency in processing such models through the Vivado place-and-route optimization algorithm, resulting in more optimized designs. However, it is worth noting that these disparities are not particularly significant, as the optimization strategies within Vivado can be adjusted to enhance the standard results outlined in our tables.

TABLE IX

ACCURACY AND INFERENCE TIME (THROUGHPUT) FOR DIFFERENT STATE-OF-THE-ART APPROACHES FOR MNIST DATASET. BEST AND SECOND BEST ARE DEPICTED IN BOLD AND ITALIC, RESPECTIVELY. *TIME REPORTED FOR THIS METHOD IS COMPUTED AS: #LAYERS × REPORTED LATENCY. THIS IS, $6 \times 16ns$

MNIST	Accuracy ↑	Throughput ↓
LBNN [33]	**99.2%**	152µs
BinaryEye [34]	*98.40%*	19.28µs
ReBNet [35]	98.29%	3µs
FINN [8] [implemented by [9]]	98.40%	641ns
FINN [8] [implemented by [32]]	96%	310ns
PolyLUT [32]	96%	*96ns**
hls4ml [36]	95%	190ns
LNN (our implementation)	97.34 %	**29.1**ns

In terms of power consumption, the designs show subtle variations once more. The estimated overall power consumption by Vivado falls within the range of 2 to 3 milliwatts. This indicates that these designs offer promising attributes for applications where power consumption is a significant concern.

Finally, during the behavioral simulation in Vivado and the deployment on the FPGA of the different network models, it was observed that there is no degradation of accuracy. This indicates that the accuracy achieved in testing (see Tables III, IV, and V) will remain consistent as the network model is implemented on the FPGA. This consistency arises from the fact that LNNs are comprised of logic gates, obviating the necessity for rounding, quantization, or similar techniques.

In summary, our proposed FPGA implementation of LNNs offers the following key benefits:

- **Resource Efficiency**: Utilizes only Look-Up Tables (LUTs), avoiding flip-flops, RAM, and DSP blocks.
- **High Throughput**: Achieves inference times in nanoseconds due to exclusive LUT usage.
- **Storage Optimization**: No weights required, alleviating storage limitations.
- **Low Power Consumption**: Power consumption ranges from 2 to 3 milliwatts.
- **No Accuracy Loss**: Test accuracy remains consistent during FPGA deployment.

A. Comparison With Other Approaches

In the following, we present two sets of results, focusing on accuracy and inference time (throughput), for our proposed implementation of Logic Neural Networks (LNNs). We compare our approach against various state-of-the-art methods, including BNNs, QNNs, and LUT-based Neural Networks, using two datasets: MNIST and Jet Substructure Classification (JSC). Results are summarized in Tables IX and X. For fair comparisons, we adopted the same methodology as in [32], where the approach is implemented on a Virtex UltraScale+ xcvu9p-flgb2104-2-i FPGA, using Vivado's Flow PerfOptimized high strategy.

Our proposed LNN implementations demonstrate strong competitiveness across both datasets, outperforming other methods that focus on reducing conventional neural networks, rather than using a co-design strategy as employed in LNNs.

TABLE X
ACCURACY AND INFERENCE TIME (THROUGHPUT) FOR DIFFERENT STATE-OF-THE-ART APPROACHES FOR JSC DATASET. BEST AND SECOND BEST ARE DEPICTED IN BOLD AND ITALIC, RESPECTIVELY. *TIME REPORTED FOR THIS METHOD IS COMPUTED AS: #LAYERS × REPORTED LATENCY. THIS IS, $3 \times 5ns$

JSC	Accuracy ↑	Throughput ↓
PolyLUT [32]	72%	*15ns**
LogicNet [37]	72%	50ns
LNN (our implementation)	72%	**14.6ns**

VI. DISCUSSION

In this section, we give a brief summary of the findings from the previous software and hardware experiments, a scalability discussion of LNNs and some insights on other architectures.

A. Experimental Findings

- **Importance of certain logic functions**: From the complete set of 16 logic functions employed in LNNs, NOR and NAND are crucial for maintaining network accuracy because they form the basic building blocks of such architectures.
- **Reduced set of logic functions**: LNNs can be exclusivelly constructed utilizing NAND and NOR gates, simplifying the design and potentially reducing costs. This provides two key benefits:
  1) Speeds up the training time as the number of logic gates to process per neuron is reduced.
  2) It enables the creation of ASICs using only NAND or NOR technology.
- **Wider layers enhance performance**: Networks with wider layers (more neurons per layer) tend to perform better, as they can process more information simultaneously, improving overall network efficiency. A possible explanation is the limited fan-in of 2-input logic gates, unlike neural networks where each neuron processes all (fully-connected) inputs.
- **Architectures with fewer logic functions**: When fewer types of logic functions are used, more neurons per layer are needed to achieve similar accuracy levels, indicating a trade-off between the variety of logic functions and the number of neurons.

B. Scalability Issues

During training stage, each logic neuron needs 16 real-valued surrogate functions. This is because 2-input logic gates can implement up to $2^{2^2} = 16$ boolean functions. As fan-in increases, the number of surrogate functions grows double-expotentially at 2^{2^k} [9], making training impractical. Additionally, stacking many layers leads to vanishing gradients, which typically become problematic around 8-10 layers, similar to conventional neural networks.

C. Other Architectures

Exploring alternative architectures using LNNs is a future goal not covered in this work. However, we highlight some connections between logical convolutions [38] and LNNs. Specifically, an LNN with 2-input logic gates can be viewed as an adaptive convolution, functioning like a kernel with a 2-bit receptive field, moving across the data. While this is not a true convolution since the translation-invariant property is absent, which is fundamental for specific image processing applications, the reduced receptive field of an LNN provides locality properties suitable for data where information is inherently *local*.

VII. CONCLUSION

In this paper, we explored Logic Neural Networks (LNNs) as a novel paradigm for efficient FPGA implementation of neural networks.We have addressed the lack of established guidelines for implementing LNN architectures in FPGAs by conducting a comprehensive study. Our experiments investigated the impact of logic gates on inference time, power consumption, and simplicity of LNNs.

Key findings include the importance of certain logic functions, such as NOR and NAND, in maintaining network accuracy. We observed that wider layers tend to enhance network performance, with architectures prioritizing width over depth. However, architectures with fewer logic functions require more neurons per layer to achieve comparable accuracy levels. This fact also implies that LNNs can be created from a reduced set of logic functions such as NAND and NOR gates for example, enabling the development of ad-hoc Application-Specific Integrated Circuits (ASICs) exclusively using NAND or NOR technology.

Our hardware implementation on the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit revealed that LNNs utilize Look-Up Tables (LUTs) exclusively, resulting in fast inference times on the order of nanoseconds and minimal power consumption in the milliwatt range.

In summary, our study provides valuable insights into designing efficient LNN architectures for FPGA implementation, facilitating further advancements in this field. Future directions include the study and development of more complex LNN architectures to face more challenging problems.

REFERENCES

[1] M. Dehghani et al., "Scaling vision transformers to 22 billion parameters," in *Proc. 40th Int. Conf. Mach. Learn.*, vol. 202, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., Jul. 2023, pp. 7480–7512. [Online]. Available: https://proceedings.mlr.press/v202/dehghani23a.html

[2] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.

[3] N. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Architect.*, 2017, pp. 1–12.

[4] A. Darwiche and P. Marquis, "A knowledge compilation map," *J. Artif. Intell. Res.*, vol. 17, pp. 229–264, Sep. 2002.

[5] R. Riegel et al., "Logical neural networks," 2020, *arXiv:2006.13155*.

[6] P. Sen, B. W. de Carvalho, R. Riegel, and A. Gray, "Neuro-symbolic inductive logic programming with logical neural networks," in *Proc. AAAI Conf. Artif. Intell.*, 2022, vol. 36, no. 8, pp. 8212–8219.

[7] Z. Susskind et al., "Weightless neural networks for efficient edge inference," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, Oct. 2022, pp. 279–290.

[8] Y. Umuroglu et al., "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2017, pp. 65–74.

[9] F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, "Deep differentiable logic gate networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 2006–2018.

[10] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 11264–11272.

[11] Y. Wang et al., "Pruning from scratch," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2020, vol. 34, no. 7, pp. 12273–12280.

[12] Y. Bai, H. Wang, Z. Tao, K. Li, and Y. Fu, "Dual lottery ticket hypothesis," 2022, *arXiv:2203.04248*.

[13] S. Liu et al., "Sparse training via boosting pruning plasticity with neuroregeneration," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 34, 2021, pp. 9908–9922.

[14] S. Liu et al., "Deep ensembling with no overhead for either training or testing: The all-round blessings of dynamic sparsity," 2021, *arXiv:2106.14568*.

[15] X. Ma, G. Fang, and X. Wang, "LLM-Pruner: On the structural pruning of large language models," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, 2023, pp. 21702–21720.

[16] X. Men et al., "ShortGPT: Layers in large language models are more redundant than you expect," 2024, *arXiv:2403.03853*.

[17] A. Kuzmin, M. Nagel, M. Van Baalen, A. Behboodi, and T. Blankevoort, "Pruning vs quantization: Which is better?" in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, 2024, pp. 1–14.

[18] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4820–4828.

[19] I. Hubara, M. Courbariaux, and D. Soudry, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, pp. 1–30, Jan. 2018.

[20] Y. Guo, "A survey on methods and theories of quantized neural networks," 2018, *arXiv:1808.04752*.

[21] B. Moons, K. Goetschalckx, N. Van Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks," in *Proc. 51st Asilomar Conf. Signals, Syst., Comput.*, Oct. 2017, pp. 1921–1925.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I*. Springer, 2016, pp. 525–542. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-46493-0_32

[23] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1," 2016, *arXiv:1602.02830*.

[24] M. Kim and P. Smaragdis, "Bitwise neural networks," 2016, *arXiv:1601.06071*.

[25] C. Yuan and S. S. Agaian, "A comprehensive review of binary neural network," *Artif. Intell. Rev.*, vol. 56, no. 11, pp. 12949–13013, Nov. 2023.

[26] S. Ma et al., "The era of 1-bit LLMs: All large language models are in 1.58 bits," 2024, *arXiv:2402.17764*.

[27] I. A. M. Huijben, W. Kool, M. B. Paulus, and R. J. G. van Sloun, "A review of the Gumbel-max trick and its extensions for discrete stochasticity in machine learning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 2, pp. 1353–1371, Feb. 2023.

[28] Y. Kim, "Deep stochastic logic gate networks," *IEEE Access*, vol. 11, pp. 122488–122501, 2023.

[29] G. Chen, "Learning symbolic expressions via gumbel-max equation learner networks," 2020, *arXiv:2012.06921*.

[30] A. Payani and F. Fekri, "Inductive logic programming via differentiable deep neural logic networks," 2019, *arXiv:1906.03523*.

[31] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control, Signals, Syst.*, vol. 2, no. 4, pp. 303–314, Dec. 1989.

[32] M. Andronic and G. A. Constantinides, "PolyLUT: Learning piecewise polynomials for ultra-low latency FPGA LUT-based inference," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2023, pp. 60–68.

[33] J. Zhan, X. Zhou, and W. Jiang, "Field programmable gate array-based all-layer accelerator with quantization neural networks for sustainable cyber-physical systems," *Softw., Pract. Exper.*, vol. 51, no. 11, pp. 2203–2224, Nov. 2021.

[34] P. Jokic, S. Emery, and L. Benini, "BinaryEye: A 20 kfps streaming camera system on FPGA with real-time on-device image recognition using binary neural networks," in *Proc. IEEE 13th Int. Symp. Ind. Embedded Syst. (SIES)*, Jun. 2018, pp. 1–7.

[35] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "ReBNet: Residual binarized neural network," in *Proc. IEEE 26th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 57–64.

[36] J. Ngadiuba et al., "Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml," *Mach. Learn., Sci. Technol.*, vol. 2, no. 1, Dec. 2020, Art. no. 015001.

[37] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "LogicNets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proc. 30th Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2020, pp. 291–297.

[38] G. Robinson, "Logical convolution and discrete Walsh and Fourier power spectra," *IEEE Trans. Audio Electroacoust.*, vol. AU-20, no. 4, pp. 271–280, Oct. 1972.

Iván Ramírez received the B.Sc. degree in telecommunication engineering from Universidad de Sevilla, Seville, in 2014, and the M.Sc. and Ph.D. degrees in computer vision from Universidad Rey Juan Carlos, Madrid, in 2016 and 2019, respectively.

Following that, he was a Post-Doctoral Associate with the DAI-Laboratory, Massachusetts Institute of Technology (MIT), Boston, in 2021. Currently, he is an Assistant Professor with Rey Juan Carlos University, where he is associated with the CAPO Research Group. His research interests include machine learning for computer vision, deep learning, representation learning, and variational methods for image processing.

Francisco J. Garcia-Espinosa received the M.Sc. degree in computer vision from Universidad Rey Juan Carlos in 2016 and the Ph.D. degree in information technology from Universidad Rey Juan Carlos in 2022. He is currently a member of the CAPO Research Group, Department of Computer Science. His research interests include image processing, computer vision, and deep learning.

David Concha received the B.Sc. degree in computer engineering and the M.Sc. degree in computer vision and the Ph.D. degree (Hons.) in information technologies from Universidad Rey Juan Carlos, Spain, in 2011 and 2021, respectively. His research interests include high-performance computing, computer vision, machine learning, GPU computing, and hardware implementation.

Luis Alberto Aranda received the B.Sc. degree in industrial engineering and the M.Sc. degree in robotics from Universidad Carlos III de Madrid, Spain, in 2012 and 2015, respectively, and the Ph.D. degree (Hons.) in industrial engineering from Universidad Antonio de Nebrija, Madrid, Spain, in 2018.

From 2013 to 2014, he was a Project Engineer with Zeus Creative Technologies S.L., developing various computer vision projects. Subsequently, he was a Professor and a Researcher with Universidad Antonio de Nebrija from 2015 to 2021. Currently, he is with the CAPO Research Group, Universidad Rey Juan Carlos, Madrid. He has actively participated in several research projects in collaboration with the National Institute of Aerospace Technology (INTA), European Space Agency (ESA), Cobham Gaisler AB, and QinetiQ Space NV. He is the author of several technical publications both in journals and international conferences. His research interests include reliability in reconfigurable computing for space applications and hardware implementation of vision and artificial intelligence circuits.