ISSN: 2395-1680 (ONLINE)
DOI: 10.21917/ijme.2026.0373

RECONFIGURABLE NEURAL NETWORK ON FPGA

Shyam Peraka, Venkatesh Mone, Sri Valli Gaddam and Manogna Annangi
Department of Electronics and Communication Engineering, Rajiv Gandhi University of Knowledge Technologies, India

Abstract

This paper presents a comprehensive methodology for transferring a four-layer feed-forward neural network, trained on the MNIST dataset, to a Xilinx Zynq-7000 System on Chip (SoC). The pretrained parameters are transformed into custom hardware modules optimized for on-chip memory using the Zynet framework. A lightweight software routine oversees AXI-DMA transfers and the collection of results through interrupts. The synthesized network layers and data transfer engines are integrated within the programmable logic fabric, while the ARM Cortex-A9 core manages task sequencing, data validation, and user interaction. Hardware-in-the-loop testing conducted on a Zed board demonstrates that the hardware implementation achieves classification accuracy comparable to software references, rapid inference speed, and minimal processor overhead. The real-time serial output of predictions against ground-truth labels facilitates immediate verification and effective debugging. This paper exemplifies the effectiveness of hardware–software co-design in creating compact and energy-efficient neural inference systems.

Keywords:

Feed-Forward Neural Network, FPGA Deployment, Zynet Framework, Hardware–Software Co-Design

1. INTRODUCTION

The recent acceleration of deep learning has significantly altered how machines learn to identify and react to intricate data. Among its many accomplishments, automatic handwriting character recognition stands out as a practical and educational benchmark in pattern recognition research [1]. [2]Handwritten digit recognition not only aids in tasks such as document scanning, postal-code sorting, and form processing, but also serves as a compact platform for exploring network architectures, training techniques, and deployment strategies. However, transferring a trained neural model from a high-level software development environment to a resource-constrained, real-time hardware platform such as an FPGA poses significant challenges regarding data accuracy, memory architecture, and computational parallelism.

The Modified National Institute of Standards and Technology (MNIST) dataset is one of the most widely used collections for evaluating handwritten digit classification algorithms. [3], [4]Comprising 60,000 training images and 10,000 test images, each represented as a 28×28 grayscale pixel matrix, MNIST offers a well-established, canonical test that balances simplicity with enough variation to challenge both accuracy and generalization. By normalizing pixel values and providing well-defined samples for ten digit classes (0 to 9), MNIST enables rapid prototyping of network architectures and facilitates comparisons among different quantization methods, activation functions, and layer configurations, all within a unified experimental framework.

Zynet is a framework based on Python designed for hardware generation, aimed at bridging the gap between high-level specifications of neural networks and synthesizable designs for FPGAs. It automates the generation of parameterized IP cores by utilizing pretrained weight and bias files, converting arithmetic operations into fixed-point formats, and instantiating the required BRAMs and processing elements within a Vivado project. The framework's built-in support for layerwise modularization, hardmax functions, and AXI-Lite/AXI-Stream interfaces greatly reduces the necessity for low-level HDL coding, enabling designers to focus on optimizing the network instead of dealing with intricate hardware details. This makes Zynet ideally suited for our project, as it streamlines the entire process from Python-based training to the generation of FPGA bitstreams.

Our project aimed to demonstrate that a moderately deep neural network, trained on a standard benchmark like MNIST, can be effectively mapped to a mid-range Zynq-7000 FPGA without sacrificing accuracy or resource constraints [5]. By leveraging Zynet's IP-core generation alongside Vivado's synthesis capabilities, we aimed to implement a four-layer feed-forward network (with neuron counts of 784, 30, 20, and 10) using fixed-point arithmetic, ensuring functional correctness through simulation and hardware-in-the-loop testing, and gathering real-world performance metrics such as classification accuracy, resource usage, operating frequency, and power consumption. Ultimately, this initiative illustrates how reconfigurable platforms can support compact, high-throughput neural inference engines suitable for embedded and edge computing applications.

2. METHODOLOGY

2.1 WEIGHTS AND BIASES EXTRACTION

We employed the TensorFlow library to train a feed-forward neural network using the MNIST dataset, thereby automatically identifying its learned parameters [6], [7]. The procedure commences with the loading of the standard MNIST training and testing datasets, followed by a row-wise normalization that rescales pixel values to a range between 0 and 1. Subsequently, a Sequential model is established, beginning with a flatten layer, and is succeeded by two hidden dense layers containing 30 and 20 neurons, respectively, which may utilize either ReLU or sigmoid as the activation function [8], [9]. The model concludes with one dense output layer, comprising 10 neurons, employing ReLU or sigmoid activation to represent the ten digit classes [10]. After compiling the network with the Adam optimizer and sparse categorical cross-entropy loss, training is conducted for 20 epochs. Upon completion, the model's performance is evaluated on the test set, and the script iterates through each dense layer (excluding the flatten layer) to extract weight matrices and bias vectors. These parameters are then translated as necessary, serialized in JSON format, and written to an external text file for subsequent hardware mapping and analysis.

2.2 ZYNET-DRIVEN IP CORE

The specified network parameters are integrated into the Zynet Python platform to construct hardware systems [11]. The framework's model-building capability utilizes weights and biases sourced from an external parameter file, with the pretrained flag set to 'yes' by default, ensuring their utilization rather than random initialization (if absent or set to 'no,' the model will be constructed without the loaded parameters). Before executing the script, it is essential to accurately configure the path to the Xilinx Vivado installation directory within the system environment to ensure successful execution of project-creation commands. A hardmax operation is performed in the final stage to derive the ultimate class outputs in hardware, resulting in a one-hot encoding of the neuron outputs that streamlines digital decision logic and minimizes resource usage. When invoking the Zynet project-creation API, an FPGA device identifier is provided to ensure compatibility of the created hardware with the target platform's structure. Furthermore, a descriptive project name is submitted to Vivado, facilitating easy identification and management within the toolset. After the project setup, the IP-generation API is called to produce the necessary cores. Finally, the system-generation API assigns a user-defined name to the top-level block design, simplifying its integration into higher-level designs and subsequent maintenance.

Within the established Vivado project, each pre-trained parameter is supplied as an individual Memory Initialization File (MIF), facilitating the automatic population of on-chip memories during the configuration process. As shown in Fig.1, files prefixed with 'w' denote weight values, while those prefixed with 'b' indicate bias terms; the subsequent numeric identifiers correspond to the layer index and neuron index (both starting from zero), establishing a direct relationship between each MIF and its respective BRAM or distributed memory instance [12]. During synthesis and implementation, these MIFs are specified in the memory IP configuration dialogs, ensuring that when the FPGA bitstream is loaded, all weights and biases are positioned in their designated memory blocks without additional run-time transfers. This organized approach not only enhances parameter management by categorizing all trained values into descriptive files but also guarantees that the hardware is equipped with the exact numeric model acquired during the training phase.

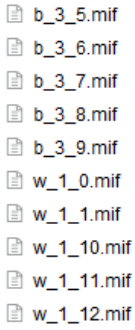

Fig.1. Memory Initialization Files

Executing this script also generates a Vivado project alongside an AXI-Lite wrapper interface; this wrapper serves as a low-latency, register-mapped connection between the neural network IP and the processing system, facilitating efficient parameter updates and status monitoring. Further, the implementation organizes all resulting design artifacts in a systematic folder structure, promoting effective management of files throughout the development and deployment processes.

2.3 VIVADO SIMULATION SETUP

A Python script is executed to generate fixed-point test vectors for each of the 10,000 images in the MNIST test set. The script quantizes each pixel value into two's-complement format at runtime and produces these integer values in a C-style header file for direct memory initialization within the hardware . Concurrently, two distinct plain-text files are created: one containing the visualized binary matrices for each digit, which is human-readable, and another containing the raw test data sequences intended for use in the Vivado project's testbench simulation.

We transition from project generation to functional verification through simulation within the Vivado environment. As a simulation folder does not exist at this stage, starting the simulation task in Vivado will automatically create a 'simulation' directory at the project's root. We place our raw test vectors—sequences of normalized MNIST-style pixel data formatted for the Vivado testbench—into this directory. Subsequently, we configure the testbench to receive these inputs and appropriately drive the neural-network IP core. After combining the stimuli, we execute the simulation, monitoring output waveforms and log files. The simulation log enumerates each test case by index and provides recognition accuracy along with the detected digit and its expected counterpart; entries such as 'Accuracy, Detected number, Expected' are displayed for each pattern. Ultimately, a summary line presents the overall accuracy across all test sequences. These comprehensive logs confirm that the synthesized hardware accurately reflects the trained network and assess its performance before hardware prototyping.

We finalize the top-level block design by generating an HDL wrapper that exposes its I/O ports for use by subsequent tools. Utilizing Vivado's 'Create HDL Wrapper' command, we automatically produce a wrapper file around the previously developed user-named block design. Once the wrapper is in place, we proceed to synthesis, where the design is transformed into an optimized gate-level netlist tailored for our target FPGA. Following synthesis, the implementation phase executes the placement and routing processes, applying the netlist to the physical resources of the device. After routing is complete, Vivado generates the configuration bitstream that encompasses the fully routed design. The next step involves exporting the hardware definition, which includes the bitstream and hardware handoff files, to the Xilinx SDK. Finally, we launch the SDK, enabling software development to commence on the newly provisioned hardware platform, thereby facilitating embedded application integration and system-level validation.

2.4 PROGRAMMING IN XILINX SDK

A designated header file developed during Phase 3 is initially incorporated into the embedded application to define a standard MNIST test vector and its corresponding ground-truth label, as illustrated in Fig.2. This file specifies an array of 784 two's-complement integers, representing each pixel in the 28×28 image, and includes the expected classification outcome. By integrating this header directly into the C program, the system is capable of

performing an end-to-end inference on a single sample and directly comparing the hardware output to the known label without the necessity for external test harnesses.

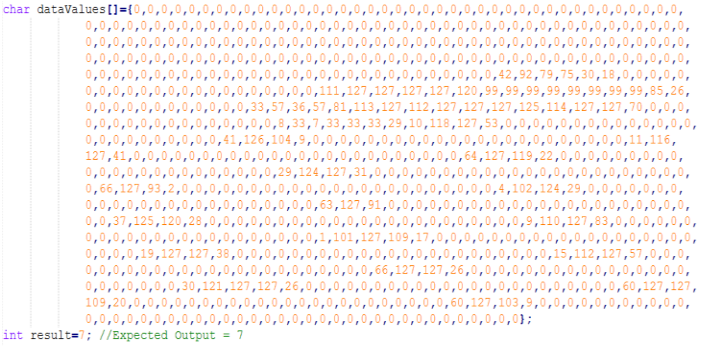

Fig.2. Header File

Inside the Xilinx SDK, an embedded C program is developed to test and validate the hardware-mapped network. As shown in Fig.3, the code initiates by invoking the AXI-DMA configuration API to set up the DMA engine for large-scale data transfer. Following this, the ARM interrupt controller is configured: its driver is initialized, the appropriate interrupt line for the neural-network IP is assigned a priority and trigger type, and a custom interrupt service routine is registered. Once the DMA and interrupts are configured, the main thread initiates a DMA transfer that streams the test-vector array to the neural-network IP core and awaits the ISR to signal completion. Upon transfer completion, the software retrieves the classification index via a memory-mapped register and outputs both the identified digit and the expected value to the console. The on-chip program employs interrupt-driven AXI-DMA to transfer input blocks, weight/bias streams, and output activations in large bursts without involving the processing system. After each transfer concludes, the interrupt controller invokes the registered callback function, enabling the software to queue subsequent transfers, verify results, or recover from errors without entering blocking loops. This architecture achieves optimal data movement throughput between programmable logic and the processing system while maintaining minimal processor overhead and latency.

```
myDmaConfig = XAxiDma_LookupConfigBaseAddr(XPAR_AXI_DMA_0_BASEADDR);
status = XAxiDma_CfgInitialize(&myDma, myDmaConfig);
if(status != XST_SUCCESS){
    print("DMA Initialization failed \n");
    return -1;
}

XScuGic_Config *IntcConfig;
IntcConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
status = XScuGic_CfgInitialize(&IntcInstance, IntcConfig, IntcConfig->CpuBaseAddress);
if(status != XST_SUCCESS){
    xil_printf("Interrupt Controller Initialization failed \n");
    return -1;
}

XScuGic_SetPriorityTriggerType(&IntcInstance, XPAR_FABRIC_ZYNET_0_INTR_INTR, 0xA0, 3);
status = XScuGic_Connect(&IntcInstance, XPAR_FABRIC_ZYNET_0_INTR_INTR, (Xil_InterruptHandler)nnISR, 0);
if(status != XST_SUCCESS){
        xil_printf("Interrupt Connection failed \n");
        return -1;
}
```

Fig.3. Setup Routine for AXI-DMA and PS–PL Interrupts

2.5 VERIFICATION ON FPGA

After building the application, we upload it to the FPGA using the debugger provided by the SDK. Under Run → Run Configurations…, we select System Debugger on Local. As illustrated in Fig.4, within the Target Setup tab, we select the following options to ensure that the programmable logic and processing system are correctly configured prior to executing the code: Program FPGA (PL), Run PS7 Init, and Run PS7 Post ConFig.Once these options are set, we navigate to the Application tab, select the ps7_cortexa9_0 entry, and click Run. This sequence programs the bitstream into the PL in a systematic manner, initializes the ARM cores at the system level, applies any necessary post-configuration register settings, and finally downloads and starts the test application.

To observe the inference output of the network, one can connect a serial terminal application such as Tera Term or access the Console view within the SDK. It is essential to ensure that the UART settings correspond with the board configuration (for instance, baud rate, data bits, etc.). As the application operates, the classification output is transmitted via the UART interface. For instance, output lines may appear as 'Detected Number: 7 Expected Number: 07,' which verifies that the hardware-accelerated neural network has accurately detected the test pattern, thereby confirming the functionality of the end-to-end system [13].

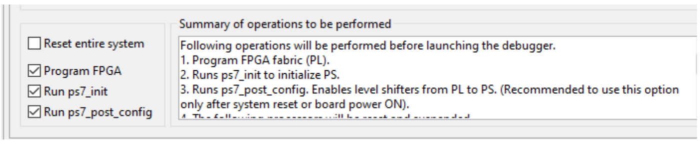

Fig.4. Xilinx SDK System Debugger Configuration Dialog

3. HARDWARE IMPLEMENTATION PLATFORM

3.1 TOP-LEVEL BLOCK DESIGN

Upon the completion of the Zynet-driven system using the Vivado project hierarchy, the top-level block design, along with its corresponding HDL wrapper, is incorporated as in Fig.5. The processing system, interconnect fabrics such as AXI interconnect and SMC, the DMA engine, and the custom neural-network IP are interconnected via AXI-Lite and AXI-Stream interfaces within the block design. Additionally, Vivado can generate an HDL wrapper module for this block design, encapsulating the entire hardware subsystem into a singular RTL entity. This wrapper facilitates the external ports of the block design, including clock, reset, and AXI control interfaces, enabling straightforward instantiation within higher-level designs or integration into comprehensive FPGA systems. By providing a structured, tool-generated wrapper, the system enhances the ease of hardware implementation, allowing for rapid integration, simulation, and synthesis with minimal user intervention.

Fig.5. Top-Level Block Design with HDL Wrapper

3.2 HIERARCHY

The Fig.6 depicts the source hierarchy of the Vivado project, which consists of a single top-level HDL wrapper and five distinct Verilog modules produced by Zynet. The AXI-Lite wrapper module offers a streamlined, register-mapped interface for managing the neural network and monitoring its status. Following this, there are three layer modules, each featuring a feed-forward stage: the first includes 30 processing elements, the second has 20, and the third, which serves as the output stage, contains 10 [14]. Additionally, a maxFinder module, designed to execute the hardmax function, evaluates the outputs from the layers and provides a one-hot encoded class decision. By compartmentalizing each component of the neural network into separate HDL files, Zynet enhances modular synthesis, simplifies timing closure tasks, and minimizes debugging efforts. This design approach also allows for the individual optimization of the arithmetic and activation logic for each layer, while the maxFinder module ensures efficient execution of the final classification task, all under the framework of the autogenerated top-level wrapper.

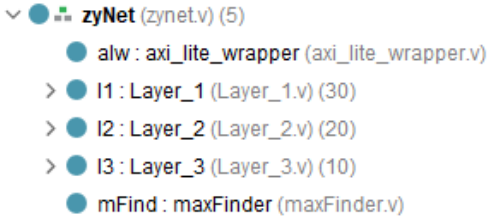

Fig.6. Hierarchy of Zynet-Produced Modules

3.3 HARDWARE VALIDATION

As demonstrated in Fig.7, the complete hardware platform was developed and validated using a Zed board development kit, which features the Xilinx Zynq-7000 SoC (device part number xc7z020clg484-1) [15]. The synthesized neural network intellectual property, comprising layer modules, AXI-DMA engines, and control wrappers, is integrated within the programmable logic fabric. Meanwhile, the ARM Cortex-A9 processing system runs the embedded application responsible for transferring test data and verifying results. The on-board DDR memory stores input vectors and intermediate activations, while the PS–PL interconnects (AXI-Lite for control and AXI-DMA for bulk data transfer) provide the necessary bandwidth for real-time inference. Testing on this platform confirmed the correct functional operation and performance characteristics within a standard FPGA environment [16], [17].

Fig.7. Zed Board Hardware Platform Featuring Zynq-7000

In real-time operation, the FPGA design transmits each quantized test vector to the neural-network IP and relays the classification result through a serial interface to a terminal emulator. As demonstrated in Fig.8, Tera Term shows that the system displays both the predicted digit and the actual ground-truth label, such as 'Detected Number 7 Expected Number 7', for each inference cycle. This immediate feedback guarantees that the hardware accurately replicates the actions of the trained model, while the printed output facilitates straightforward validation and rapid identification of misclassifications.

Fig.8. Real-Time Classification Output via Serial Terminal

4. RESULTS

The proposed neural network, after undergoing training for 20 epochs on the MNIST dataset, achieved a testing accuracy of 96.69% in its software implementation. The architecture comprises four layers: an input layer containing 784 neurons for the 28×28 pixel input images, two hidden layers with 30 and 20 neurons respectively, and an output layer with 10 neurons corresponding to digit classes 0-9. A hardmax function is employed at the output layer to determine the class with the highest activation. The sigmoid activation function is utilized across all layers due to its compatibility with low-resource hardware and its ability to facilitate smooth gradient flow.

The Fig.9 depicts a timing waveform captured during the on-chip validation of the neural network intellectual property. After the reset signal is released, the clk and in_valid signals initiate a burst of 8-bit input samples (displayed on in[7:0]), which are retrieved from the on-chip memory (in_mem). The AXI-Stream interfaces (s_axi_awvalid, s_axi_wvalid, s_axi_wdata) demonstrate the handshaking process for each write operation, while the start flag alternates to indicate the commencement of each inference cycle. As the layerNo counter iterates from 1 to 3, the testDataCount register increments with each completed classification, and the right/wrong indicators briefly pulse to signify correct or incorrect predictions. Collectively, these traces confirm that the hardware effectively processes each test vector, updates internal counters, and delivers classification results in real time [18], [19].

In terms of hardware resource utilization, the design was efficient and adhered to the limitations of the target FPGA. The Fig.10 represents the resource utilization metrics of a neural network comprising two hidden layers, with 30 and 20 neurons respectively, employing the sigmoid activation function and utilizing an 8-bit data width. The usage of slice Look-Up Tables (LUTs) was approximately 14% of the available resources, while the consumption of Flip-Flops (FFs) was minimal at around 4%. The Block RAM (BRAM), which stored intermediate data and parameters, utilized nearly 19% of its capacity. Input/Output (IO)

pins were the most utilized, accounting for nearly 49% of the available pins, indicating a significant need for data exchange with external devices. Clocking resources such as BUFGs were used minimally, occupying only about 3%.

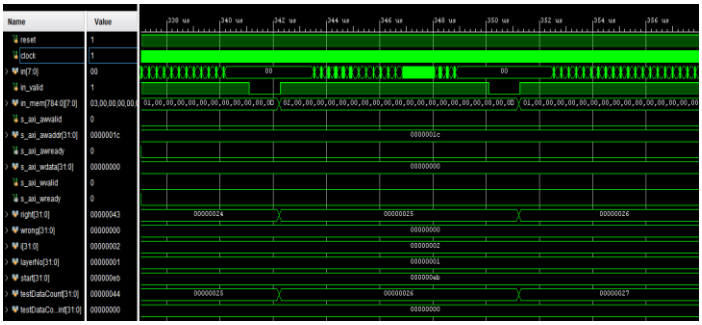

Fig.9. Simulation Waveforms

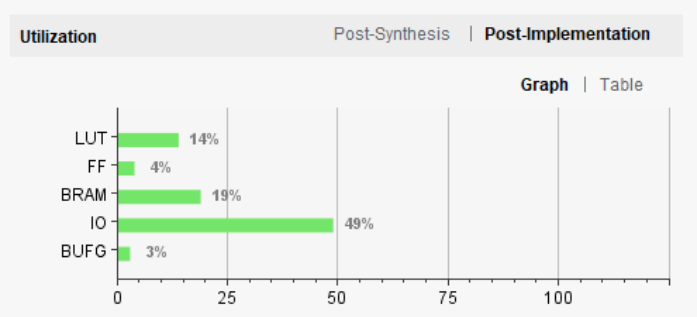

Fig.10. FPGA Resource Utilization

Table.1. Classification Performance Overview

Activation Function	Data Width	Frequency (MHz)	Power (mW)	Accuracy (%)
Sigmoid	8	119	257	99
	16	112	389	99
	24	104	510	99
	32	119	257	99
Relu	8	126	183	80
	16	106	345	87
	24	101	404	87
	32	80	706	88

The Table.1 presents a summary of the classification performance for all combinations of quantization schemes and activation functions evaluated on the MNIST test set. [20]Table 2 details the FPGA resource utilization, including LUTs, flip-flops, BRAM, IO, and BUFG, associated with each neural network configuration under examination [21].

The trained model was successfully deployed on a Zed board featuring an xc7z020clg484-1 SoC, utilizing the Xilinx Vivado 2019.1 design suite for synthesis, implementation, and bitstream generation. The design achieved a maximum operating frequency of 119 MHz. In summary, the hardware implementation not only achieved a commendable classification accuracy of 99% on test inputs but also operated with low power consumption, recorded at approximately 0.257 watts [22]. These outcomes underscore the design's effectiveness in terms of performance, efficiency, and its appropriateness for real-time embedded applications.

Table.2. Resource Usage per Network Configuration

Activation Function	Data Width	Resource Utilization				
		LUT	FF	BRAM	IO	BUFG
Sigmoid	8	7337	3766	27	97	1
	16	4691	4113	30	105	1
	24	9292	6667	60	113	1
	32	7337	3766	27	97	1
Relu	8	7485	3929	1	97	1
	16	6133	5120	15	105	1
	24	11398	8101	26	113	1
	32	23519	11849	30	121	1

5. CONCLUSION

This research has employed a comprehensive strategy to implement a moderately deep neural network featuring a fully connected layer on an Xilinx Zynq-7000 FPGA. It began with training in Python using the MNIST dataset, progressed to automated IP-core generation through Zynet [23], and concluded with real-time hardware-in-the-loop verification and integration of hardware and software. By utilizing fixed-point representations for network parameters and activations, along with leveraging the on-chip memory management and AXI interconnect features of Zynet, the implemented four-layer feed-forward architecture achieved impressive classification accuracy both in simulation and during on-board inference. Interrupt-driven AXI-DMA transfers alleviated the processor's workload, facilitating seamless streaming of pixel data and one-hot outputs via a serial console. Observations regarding throughput, energy consumption, and overall hardware performance validate that this methodology strikes an effective balance between computational speed, efficiency, and resource utilization, underscoring the promise of reconfigurable logic for high-throughput neural inference in embedded and edge applications.

REFERENCES

[1] S. Agatonovic-Kustrin and R. Beresford, "Basic Concepts of Artificial Neural Network (ANN) Modeling and its Application in Pharmaceutical Research", *Journal of Pharmaceutical and Biomedical Analysis*, pp. 717-727, 2000.

[2] S. Cuomo, V.S. Di Cola, Giampaolo, "Scientific Machine Learning through Physics-Informed Neural Networks: Where We are and What's Next", *Journal of Scientific Computing*, Vol. 92, No. 88, pp. 1-7, 2022.

[3] Y. LeCun, "The MNIST Database of Handwritten Digits", Available at https://yann.lecun.org/exdb/mnist/index.html, Accessed in 1998.

[4] A. Baldominos, Y. Saez and P. Isasi, "A Survey of Handwritten Character Recognition with MNIST and EMNIST", *Applied Sciences*, Vol. 9, No. 15, pp. 1-7, 2019.

[5] R. Kaibou and M.S. Azzaz, "FPGA Implementation of Mixed Robust Chaos based Digital Color Image Watermarking", *Proceedings of International Conference on Networking and Advanced Systems*, Vol. 2, pp. 1-7, 2021.

[6] R.R. Kumbhar, P. Radhika and D. Mane, "Design and Optimization of an On-Chip Artificial Neural Network on FPGA for Recognizing Handwritten Digits", *Proceedings of International Conference on Recent Advances in Electrical, Electronics, Ubiquitous Communication and Computational Intelligence*, Vol. 1, pp. 1-10, 2023.

[7] S.S. Lingala, S. Bedekar, P. Tyagi, P. Saha and P. Shahane, "FPGA based Implementation of Neural Network", *Proceedings of International Conference on Advances in Computing, Communication and Applied Informatics*, pp. 1-5, 2022.

[8] F. Ortega-Zamorano, J.M. Jerez, D. Urda Munoz, R.M. Luque-Baena and L. Franco, "Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers", *IEEE Transactions on Neural Networks and Learning Systems*, Vol. 27 No. 9, pp. 1840-1850, 2016.

[9] Nwankpa Chigozie, W. Ijomah, Gachagan Anthony and Marshall Stephen, "Activation Functions: Comparison of Trends in Practice and Research for Deep Learning", *Proceedings of International Conference on Computer Vision and Pattern Recognition*, pp. 1-20, 2020.

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

[11] R. Kaibou, M.S. Azzaz, B. Madani, A. Kadir and H. Hamil, "Performances Analysis of DNN Accelerator in a HW/SW Co-Design FPGA Implementation for Image Classification", *Proceedings of International Conference on Advanced Electrical Engineering*, pp. 1-6, 2024.

[12] P. Jokic, S. Emery and L. Benini, "Improving Memory Utilization in Convolutional Neural Network Accelerators", *IEEE Embedded Systems Letters*, Vol. 13, No. 3, pp. 77-80, 2021.

[13] J. Si and S.L. Harris, "Handwritten Digit Recognition System on an FPGA", *Proceedings of International Conference on Computing and Communication*, pp. 402-407, 2018.

[14] K. Vipin, "ZyNet: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms", *Proceedings of International Conference on Field Programmable Technology*, pp. 1-8, 2019.

[15] V.A. Sumayyabeevi, J.J. Poovely, N.Aswathy and S. Chinnu, "A New Hardware Architecture for FPGA Implementation of Feed Forward Neural Networks", *Proceedings of International Conference on Advances in Computing, Communication, Embedded and Secure Systems*, pp. 107-111, 2021.

[16] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou and L. Wang, "A High Performance FPGA-based Accelerator for Large-Scale CNNS", *Proceedings of International Conference on Field Programmable Logic and Applications*, pp. 1-9, 2016.

[17] S.K. Venkataramanaiah, X. Du, Z. Li, S. Yin, Y. Cao and J.S. Seo, "Efficient and Modularized Training on FPGA for Real-Time Applications", *Proceedings of International Conference on Artificial Intelligence*, pp. 5237-5239, 2021.

[18] R. Kaibou, M.S. Azzaz, M. Benssalah, D. Teguig, H. Hamil, A. Merah and M.T. Akrour, "Real-Time FPGA Implementation of a Secure Chaos based Digital Crypto-Watermarking System in the DWT Domain using Co Design Approach", *Journal of Real-Time Image Processing*, Vol. 18, No. 6, pp. 2009-2025, 2021.

[19] S. Qiao, Z. Lin, J. Zhang and A.L. Yuille, "Neural Rejuvenation: Improving Deep Network Training by Enhancing Computational Resource Utilization", *Proceedings of International Conference on Computer Vision and Pattern Recognition*, pp. 61-71, 2019.

[20] K. Khalil, B. Dey, M. Abdelrehim, A. Kumar and M. Bayoumi, "An Efficient Reconfigurable Neural Network on Chip", *Proceedings of International Conference on Electronics, Circuits and Systems*, pp. 1-4, 2021.

[21] K. Khalil, O. Eldash, A. Kumar and M. Bayoumi, "N2OC: Neural-Network-on-Chip Architecture", *Proceedings of International Conference on System-on-Chip*, pp. 272-277, 2019.

[22] Y.H. Chen, T. Krishna, J.S. Emer and V. Sze, "Eyeriss: An Energy Efficient Reconfigurable Accelerator for DNNS", *IEEE Journal of Solid-State Circuits*, Vol. 52, No. 1, pp. 127-138, 2016.

[23] "ZyNet Git Repository", Available at https://github.com/dsdnu/zynet, Accessed in 2024.