

Development of a Unified model for Classification, Object detection and Segmentation on FPGA

A thesis submitted in partial fulfillment of the requirements for
the award of the degree of

B.Tech.

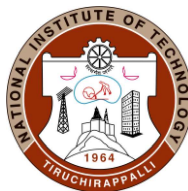
in

ELECTRONICS AND COMMUNICATION ENGINEERING

By

Nandini Kumawat (108119071)

Rakesh Mohan Patel (108119088)



**DEPARTMENT OF
ELECTRONICS AND COMMUNICATION
ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
TIRUCHIRAPPALLI-620015**

MAY 2023

BONAFIDE CERTIFICATE

This is to certify that the project titled **Development of a Unified model for Classification, Object detection and Segmentation on FPGA** is a bonafide record of the work done by

Nandini Kumawat (108119071)

Rakesh Mohan Patel (108119088)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **ELECTRONICS AND COMMUNICATION ENGINEERING** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**, during the year 2022-2023.

Dr. G. Lakshmi Narayanan

Project Guide

Dr. M. Bhaskar

Head of the Department

Project Viva-voce held on _____

Internal Examiner 1

Internal Examiner 2

ABSTRACT

In recent years, the industry has transitioned from glue logic to computational logic using field programmable gate array logic. The reconfigurability of FPGA has expanded the realm of parallel processing while retaining the programming and software reconfigurability features. This accelerated platform has surpassed traditional processors in image processing, surveillance, signal processing techniques, aerospace applications, communication systems, and video processing. With power and resource availability being major concerns for modern CPUs, maintaining high throughput is nearly impossible.

When compared to software computer systems, the fine-grain structure of FPGA presents programming problems to engineers. Object detection is important in optical remote sensing analysis on satellites or aeroplanes. Because of the limitations of traditional system architecture in terms of power consumption and performance, the focus has switched to embedded processors.

In this project we present the development of a unified model for classification, object detection, and segmentation on a Field-Programmable Gate Array (FPGA) platform. The proposed model combines state-of-the-art deep learning algorithms for each task and is optimized for efficient execution on the FPGA. The model's design is built upon a modular architecture, which permits effortless adaptation and implementation on various hardware platforms. Experimental results demonstrate that the unified model achieves high accuracy and real-time performance for all three tasks, making it a promising solution for a wide range of applications in computer vision and robotics.

ACKNOWLEDGEMENT

We would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

Dr. G. Lakshmi Narayanan, our project guide, for helping us and guiding us in the course of this project.

Dr. M. Bhaskar, the Head of the Department, ELECTRONICS AND COMMUNICATION ENGINEERING.

Our internal reviewers, **Dr. E.S. Gopi**, **Dr. S. Sudharsan**, **Dr. Srinivasalu Jogi**, **Dr. B. Naresh kumar Reddy** for their insight and advice provided during the review sessions.

We would also like to thank our individual parents and friends for their constant support.

TABLE OF CONTENTS

Title	Page No.
ABSTRACT.....	i
ACKNOWLEDGEMENT.....	ii
TABLE OF CONTENTS.....	iii
LIST OF FIGURES.....	iv
ABBREVIATIONS.....	v
CHAPTER 1 INTRODUCTION.....	1
1.1 Object Detection.....	2
1.2 Classification.....	3
1.3 Segmentation.....	4
1.4 PYNQ.....	4
1.5 LeNet-5.....	5
1.6 MobileNet.....	7
CHAPTER 2 LITERATURE SURVEY.....	8
CHAPTER 3 VITIS HLS AND VIVADO FLOW.....	12
3.1 Vitis HLS Flow.....	12
3.2 Vivado HLS Flow.....	13
CHAPTER 4 METHODOLOGY.....	15
CHAPTER 5 RESULTS.....	22
CHAPTER 6 CONCLUSION AND FUTURE SCOPE.....	24
6.1 CONCLUSION.....	24
6.2 FUTURE SCOPE.....	24
APPENDIX A CODE ATTACHMENT.....	25
A.1 LeNet – Vitis HLS source code.....	25
A.2 LeNet - Python code.....	29
A.3 BNN Python Code.....	31
REFERENCES.....	32

LIST OF FIGURES

Figure 1 Application of unified module.....	1
Figure 1.4.1 PYNQ Z2 Board.....	5
Figure 1.5: Architecture of LeNet-5.....	6
Figure 1.6: MobileNet Architecture.....	7
Figure 3.1: Vitis HLS flow.....	12
Figure 3.2: Xilinx vivado flow.....	13
Figure 4.1: LeNet IP (Developed in Vitis HLS)	15
Figure 4.2: 32-bit General Purpose axi slave interface enables.....	15
Figure 4.3: LeNet's Zynq block design (using General purpose Axis slave)	16
Figure 4.4: 32 bit High Performance axis slave interface enable	16

ABBREVIATIONS

CNN (Convolutional-Neural-Networks)

SSD (Single Shot Detection)

R-CNN (Region-based Convolutional Neural Network)

YOLO (You Only Look Once)

GPU (Graphics Processing Unit)

FPGA (Field-Programmable-Gate-Array)

RTL (Register Transfer Level)

HLS (High-Level Synthesis)

IP (Intellectual Property)

HDL (Hardware Description Language)

DDR (Double Data Rate)

DMA (Direct Memory Access)

AXI (Advanced eXtensible Interface)

CV (Computer Vision)

BNN (Binary Neural Network)

PIL (Python Imaging Library)

CHAPTER 1

INTRODUCTION

As Convolutional-Neural-Networks (CNN) technology advances, network formats emerge one after the other to overcome existing networks' performance and processing power restrictions.

Introduction of products such as Graphical Processing Units, the concept of machine learning and deep learning has become majorly applied in the study of object-detection. As a result, numerous deep learning-related object detection systems have been developed. Examples are SSD, R-CNN, and YOLO.

Despite the fact that GPUs are used in advanced algorithms like YOLO, they are inefficient in terms of obtaining lowest power usage and higher hits-per-second. Hardware-acceleration methods are used for optimising neural networks. FPGA (Field-Programmable-Gate-Array) is a low-power, high-performance, semiconductor that excels at accelerating neural network algorithms.

FPGA can provide great computational throughput while still being efficient. It also has a high degree of flexibility and is modifiable. Burning of the recorded configuration files is an option available to the developers and configure the wiring logic between the units to implement various arithmetic logic, and this can be repeated multiple times with the chip logic present to be subjected to change anytime.

The development of a unified model for classification, object detection, and segmentation on FPGA has become a topic of significant interest in the field of computer vision and deep learning. This is because a unified model can efficiently perform multiple tasks using a single network architecture, reducing computation time and resource utilization.



Figure 1 : Application of unified module

FPGAs are hardware devices that can be programmed to perform specific functions, and they are well-suited for implementing neural network models. The FPGA can be customized to accelerate the processing of large amounts of data and perform computations in parallel, making it an ideal platform for real-time computer vision applications.

The development of a unified model for classification, object detection, and segmentation on FPGA involves the creation of a single neural network architecture that can perform all three tasks. The model is trained using a large dataset of images, and it learns to identify objects, classify them, and segment them based on their visual features.

The classification task involves identifying the category of an object in an image, such as a car or a person. The object detection task involves identifying the location of objects in an image and drawing bounding boxes around them. The segmentation task involves identifying the pixels that belong to an object in an image and separating them from the background.

To develop a unified model for all three tasks, a common backbone architecture is used, which is responsible for feature extraction. This is followed by task-specific modules that perform classification, object detection, and segmentation. The backbone architecture can be a pre-trained convolutional neural network such as ResNet or VGG, which has been fine-tuned to perform all three tasks.

when the model is trained, it is optimized for deployment on an FPGA. This involves converting the model to a hardware-friendly format, such as Verilog or VHDL, and mapping the neural network to the FPGA hardware resources. The FPGA is then programmed to perform the desired tasks using the optimized neural network model.

The development of a unified model for classification, object detection, and segmentation on FPGA has several benefits. It enables real-time processing of images and videos with low latency and high accuracy, which is essential for applications such as autonomous vehicles and surveillance systems. Additionally, the use of FPGA hardware can significantly reduce power consumption and increase efficiency compared to traditional CPU or GPU-based implementations.

1.1 Object Detection

Object detection is a computer vision task which involves in identifying and localizing objects within an image or video stream. The main goal of object detection is to identify and locate objects accurately within an image or video stream, regardless of their size, orientation, or appearance. There are so many approaches to object detection, including the use of deep learning models such as Faster R-CNN, YOLO, and SSD. These models are basically trained on large datasets of labeled images and use convolutional neural networks (CNNs) to extract features from the input image.

In general, the object detection process involves several stages. The first stage is typically a pre-processing step, where the input image is resized or normalized to a fixed size. The second stage involves the application of a deep learning model to the input image, which extracts features that are relevant for object detection. The third stage involves the prediction of object bounding boxes and class labels based on the extracted features.

One common approach to object detection is the use of anchor-based methods, such as Faster R-CNN and SSD. These methods involve the use of a set of pre-defined anchor boxes that are placed at various positions and scales within the input image. The deep learning model is then trained to predict the offset and scale of each anchor box, as well as the probability of each anchor box containing an object.

Another approach to object detection is the use of anchor-free methods, such as CenterNet and FCOS. These methods do not use pre-defined anchor boxes and instead directly predict the center point and size of each object in the image. Once the object bounding boxes and class labels have been predicted, a post-processing step is typically used to refine the predictions and remove any false positives. This may involve the use of non-maximum suppression (NMS) to remove redundant bounding boxes or the application of additional heuristics or filters to remove false positives.

Object detection has many applications, including in autonomous vehicles, surveillance systems, and robotics. It is a challenging task that requires the use of advanced deep learning techniques and large datasets, but it has the potential to provide significant benefits in a wide range of domains.

1.2 Classification

Object classification is a computer vision task that involves assigning one or more class labels to an input image or video frame. The goal of object classification is to accurately identify the objects present in the image and assign them to the appropriate class labels.

Object classification can be approached using a variety of techniques, including traditional machine learning algorithms such as Support Vector Machines (SVMs), Random Forests, and Naive Bayes, as well as deep learning models such as Convolutional Neural Networks (CNNs). Generally, it involves several stages. The first stage is typically a pre-processing step, where the input image is resized or normalized to a fixed size. The second stage involves the application of a deep learning model or machine learning algorithm to the input image, which extracts features that are relevant for object classification. The third stage involves the prediction of class labels based on the extracted features.

Deep learning models such as CNNs have shown significant success in object classification tasks due to their ability to automatically learn features from the input data. CNNs typically consist of several layers of convolutional and pooling operations, which learn increasingly complex features from the input image. The output of the final layer is typically passed through one or more fully connected layers, which map the learned features to the output class labels.

To train a deep learning model for object classification, a large dataset of labeled images is typically required. The images in the dataset are labeled with one or more class labels, and the model is trained to predict these labels based on the input image. When the model has been trained, it can be used to classify new images or video frames. The input image is passed through the model, and the output of the final layer is used to predict the class labels. In some cases, multiple class labels may be assigned to the input image, for example if the image contains multiple objects or if the objects in the image can belong to multiple categories.

It has many applications, including in image search engines, content-based image retrieval systems, and medical imaging. It is a challenging task that requires the use of advanced machine learning and deep learning techniques, but it has the potential to provide significant benefits in a wide range of domains.

1.3 Segmentation

Object segmentation is a computer vision task that involves dividing an input image into multiple segments, with each segment corresponding to a distinct object or region of interest within the image. The goal of object segmentation is to accurately separate and localize objects within the image, regardless of their size, orientation, or appearance. It can be approached using a variety of techniques, including traditional image processing methods such as thresholding, edge detection, and region growing, as well as deep learning models such as Fully Convolutional Networks (FCNs), U-Net, and Mask R-CNN.

Generally, it involves several stages. The first stage is typically a pre-processing step, where the input image is resized or normalized to a fixed size. The second stage involves the application of a deep learning model or image processing algorithm to the input image, which extracts features that are relevant for object segmentation. The third stage involves the prediction of object segments based on the extracted features. Deep learning models such as FCNs and U-Net have shown significant success in object segmentation tasks due to their ability to learn to associate each pixel in the image with a specific object segment. These models typically consist of several layers of convolutional and pooling operations, which learn increasingly complex features from the input image. The output of the final layer is a segmentation map, where each pixel is assigned, a label corresponding to the object segment to which it belongs.

More advanced models such as Mask R-CNN extend the segmentation approach to also identify the specific instance of an object and generate a mask that identifies the pixels belonging to that instance. This allows the model to differentiate between multiple instances of the same object class within the image. To train a deep learning model for object segmentation, a large dataset of labeled images is typically required. The images in the dataset are labeled with object segment masks, which indicate which pixels in the image belong to each object segment. When the model has been trained, it can be used to segment new images. The input image is passed through the model, and the output of the final layer is used to predict the object segments or instance masks.

Object segmentation has many applications, including in medical imaging, robotics, and autonomous vehicles. It is a challenging task that requires the use of advanced deep learning and image processing techniques, but it has the potential to provide significant benefits in a wide range of domains.

1.4 PYNQ

Pynq is an open-source project which aims to make it easy to program Xilinx Zynq Systems on Chip (SoCs) using Python. The project provides a software platform and a set of tools that allow users to develop and run custom hardware accelerators on the FPGA fabric of the Zynq SoC using high-level programming languages like Python. The Pynq platform includes a set of pre-built overlays, which are hardware-accelerated functions that can be used as building

blocks for custom designs. Additionally, Pynq provides a Jupyter Notebook environment, which allows users to develop and test their designs interactively using Python code snippets.

The Pynq project has gained popularity in the academic community for teaching FPGA design and accelerating algorithms in fields like machine learning, computer vision, and signal processing. Pynq has also been used in industry applications such as robotics, embedded systems, and wireless communication. Pynq is an open-source project hosted on GitHub and has an active community of developers and users who contribute to its development and support.

1.4.1 PYNQ Z2 BOARD FEATURES:

PYNQ-Z2 board integrates USB and Ethernet to connect to internet, MIC Input, HDMI Input/Output, Audio Output, Arduino interface, 2 Pmod, user LED, push buttons, switches, MicroSD Slot, Raspberry Pi interface, Power In port for direct power connection, a Jumper for power source selection and another jumper to select Boot Mode.

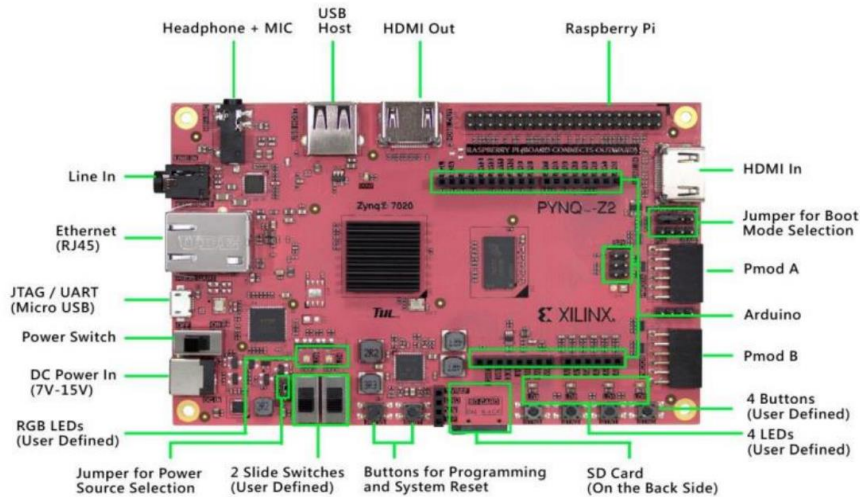


Figure 1.4.1: PYNQ Z2 Board

It is intended to be easily extensible with Pmod, Arduino, and peripherals, along with general purpose GPIO pins.

1.5 LeNet-5

LeNet is a simple convolutional neural network (CNN) which is designed for handwritten digit recognition tasks. The LeNet architecture was first proposed by Yann LeCun, Yoshua Bengio, Patrick Haffner and Leon Bottou in 1998, and it was one of the pioneering deep learning models that demonstrated the effectiveness of deep neural networks for image recognition tasks.

The LeNet architecture consists of seven layers: two convolutional layers, two subsampling (pooling) layers, and three fully connected layers. The input to the network is a 32x32 grayscale image, and the output is a probability distribution over ten possible digit classes (0-9).

The LeNet architecture has since been adapted and extended for various image recognition tasks, including object recognition, face recognition, and scene recognition. It is also commonly used as a benchmark model for evaluating new CNN architectures and optimization techniques.

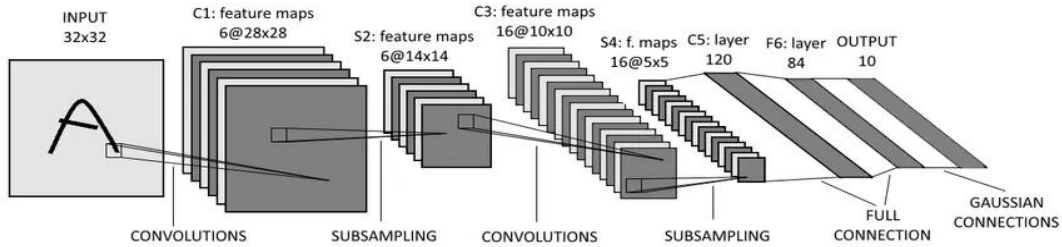


Figure 1.5: Architecture of LeNet-5

The MNIST dataset is used which images are 28x28 in size. The 28x28 images are padded such that the MNIST images' dimensions match those required by the input layer. The study paper's grayscale photos were normalised from pixel values of 0 to 255 to values between -0.1 and 1.175. The goal of normalisation is to ensure that the batch of images has a mean of 0 and a standard deviation of 1. This will result in a shorter training period. We'll be normalising the pixel values of the photos to take on values between 0 and 1 for the image classification with LeNet-5 example that follows.

The first convolution operation is applied, which have 6 filters with each of size 5X5. We get feature map that is 28X28X6 in size. Here, the number of applied filters is equal to the number of channels. When we use average pooling after the initial pooling procedure, the size of the feature map is cut in half. The following layer is a convolution layer with sixteen 5x5-inch filters. The 10X10X16 is once again modified feature map. Analogously, the output size is determined. Once more, we used an average pooling or subsampling layer, which cut the size of the feature map now 5X5X16 in half. The feature map is then reduced to size 1X1X120 by a final convolution layer of size 5X5 with 120 filters. 120 values are the consequence of flattening. Following that, we have a layer with 84 neurons that is fully connected layer. At last, there is an output layer with 10 neurons since the data have 10 classes.

1.6 MobileNet

MobileNet is a family of convolutional neural network (CNN) architectures designed for mobile and embedded devices with limited computational resources. It was first introduced in 2017 by Marco Andreetto, Menglong Zhu, Bo Chen, Andrew G. Howard, Dmitry Kalenichenko, Weijun Wang, Hartwig Adam, and Tobias Weyand. The motivation behind MobileNet is to enable on-device inference on low-power mobile and embedded devices, such as smartphones and IoT devices, without relying on cloud computing.

The key innovation of MobileNet is the use of depthwise separable convolutions, which consist of a depthwise convolution that filters the input channels and a pointwise convolution that combines the filtered channels. This approach significantly reduces the number of parameters and computations required while maintaining good accuracy performance. In

particular, MobileNet models are significantly faster and smaller than traditional CNN models without sacrificing accuracy.

MobileNet has been applied to various computer vision tasks, including object detection, image segmentation, and image classification. It has also been used as a backbone architecture in larger models, such as MobileNetV2 and MobileNetV3, which further improve the efficiency and accuracy of the original MobileNet. MobileNetV2, introduced in 2018, includes several improvements over the original MobileNet.

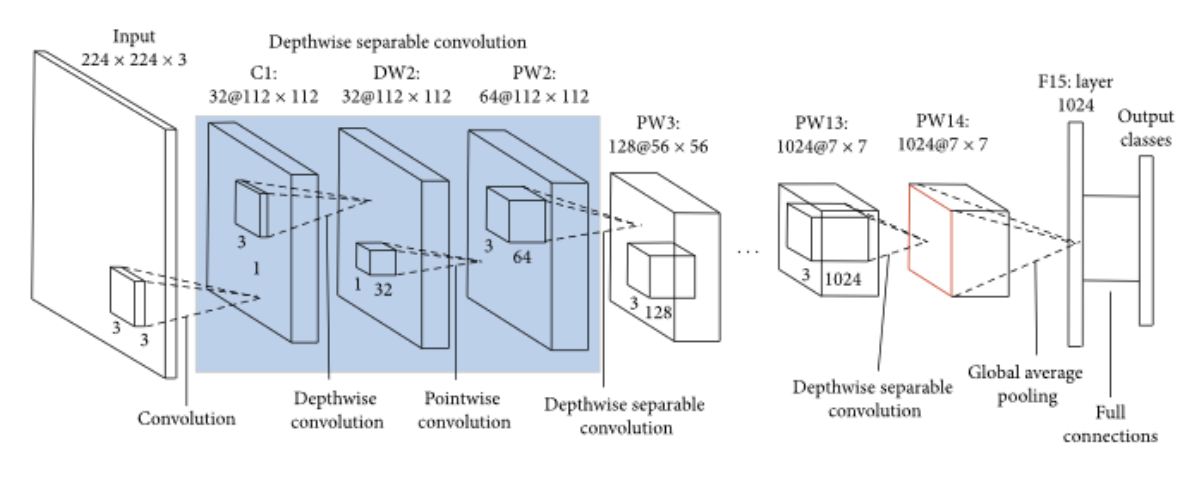


Figure 1.6: MobileNet Architecture

It uses a novel linear bottleneck design that improves the performance of depthwise separable convolutions. It also introduces a new feature called inverted residuals, which uses linear bottlenecks followed by non-linear activations to improve the accuracy of the model. MobileNetV2 achieves state-of-the-art accuracy on ImageNet while being 2x faster than MobileNetV1. MobileNetV3, introduced in 2019, further improves the efficiency and accuracy of MobileNetV2. It introduces a new search algorithm for designing neural network architectures that optimizes for both accuracy and latency. MobileNetV3 also introduces a new activation function called hard swish, which improves the non-linearity of the model while reducing computation.

CHAPTER 2

LITERATURE SURVEY

1. Implementation of CNN on Zynq based FPGA for Real-time Object Detection

The goal of this project is to implement a Convolutional Neural Network (CNN) using a Python framework on a Xilinx Zynq based Field Programmable Gate Array (FPGA). The objective is to address the challenging task of real-time object detection in computer vision. To achieve this, pre-trained CNN models are utilized, which are implemented using the TensorFlow Application Programming Interface (API). These models are then deployed on a Xilinx Zynq based FPGA using the Python productivity for Zynq (PYNQ) framework.

The versatility of this approach is tested on four state-of-the-art object detectors based on different classifiers and meta-architectures, namely MobileNet V1, Inception V2, SSD, and Faster R-CNN. These detectors have been trained on the MS-COCO dataset. The functionality of the system is compared based on system latency, detection accuracy, and ease of implementation on ARM embedded mobile platforms. The results and real-time frame rates show that the SSD with Inception V2 model is suitable for the intended application of real-time object detection. This hardware-software co-design approach forms the basis of FPGA-based hardware accelerators.

The paper also provides a comprehensive overview of the PYNQ architecture used for the FPGA design, highlighting the benefits of using high-level synthesis (HLS) design tools. It discusses the challenges of implementing CNNs on resource-constrained embedded systems and emphasizes the need for application-specific CNN architectures tailored to embedded platforms. Overall, this project contributes to the field of real-time object detection by showcasing the successful implementation of CNN models on FPGA-based hardware accelerators, enabling fast and efficient processing for computer vision applications.

2. Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS

The paper discusses the implementation of a Convolutional Neural Network (CNN) on an FPGA for image recognition tasks. CNNs are widely used in image recognition due to their ability to mimic the behavior of biological visual nerves and achieve high recognition accuracy. However, traditional processors often struggle to efficiently implement CNNs due to their unique calculation requirements.

To address this problem, the authors propose implementing CNNs on Field-Programmable Gate Arrays (FPGAs) and optimizing the convolution operation to improve computing parallelism, data throughput, and energy efficiency compared to general processors. They specifically implement the Lenet-5 model of CNN on the ZYBOZ7 FPGA board and compare its performance with a traditional processor.

The results demonstrate that the FPGA implementation achieves fast recognition of images at a frequency of 100 MHz with Direct Memory Access (DMA) control. The data throughput of the FPGA is more than four times higher than that of the general processor, while consuming

only 1.8W of power, which is significantly lower than the power consumption of the general processor.

Overall, the paper highlights the advantages of using FPGAs for CNN acceleration, including improved performance, efficiency, development cycle, and flexibility compared to traditional processors. The presented results support the feasibility and benefits of implementing CNNs on FPGA platforms for image recognition tasks.

3. PYNQ FPGA Hardware implementation of LeNet-5-Based Traffic Sign Recognition Application

The paper begins by highlighting the importance of computer vision and embedded systems, especially in applications such as advanced driver-assistance systems. It mentions the increasing computational complexity of deep learning models like CNNs and the inefficiency of implementing them on general-purpose processors. To address this, the authors propose using FPGAs, which are reconfigurable and power-efficient devices capable of parallelizing computation operations.

The authors utilize the Xilinx PYNQ framework, which combines hardware libraries on configurable FPGA designs with Python programming for communication with the processing system (PS) part of the embedded system. They describe previous works that have implemented CNN accelerators on FPGAs using various techniques and methodologies.

In their work, the authors design and implement a hardware accelerator based on the LeNet-5 model for traffic sign recognition on the PYNQ Z1 platform. They train the application on a GPU and then follow the high-level synthesis (HLS) flow paradigm to create the hardware accelerator and its interfaces for communication with the PS. They employ Vivado IP integrator to create a co-design by defining the interface between the PS and programmable logic (PL) to transfer data and control. Once the design is validated, they generate a bitstream file and use the overlay interface to implement it on PYNQ.

The paper also discusses the literature review of the LeNet-5 model, its architecture, and its usage in traffic sign recognition. They explain the GPU implementation of the LeNet-5 model for traffic sign recognition, including the preprocessing steps, training process, and performance evaluation using metrics like accuracy, precision, recall, and F1-score.

Finally, the authors present their proposed FPGA prototype for embedded traffic sign recognition based on the LeNet-5 model. They describe the creation of the LeNet-5 IP using Vivado HLS and the integration of the IP into the co-design using Vivado IP integrator. They explain the use of BRAM for storing the parameters of the LeNet-5 IP and Axis Interconnect for communication between the IP and the PS.

Overall, the paper focuses on the implementation of the LeNet-5 model on an embedded FPGA system for traffic sign recognition. It provides insights into the design and implementation process, as well as performance evaluation results.

4. A dynamic partial reconfigurable overlay concept for PYNQ

The authors introduce the concept of hybrid classes, which are Python classes that wrap the functionality of hardware accelerator modules implemented through multiple partial bitstreams. Each instantiation of a hybrid class is bound to a specific partial bitstream, and

when the object is deleted, the bitstream can be unloaded or overwritten. They also discuss the management of reconfigurable partitions (RPs) and the use of replacement strategies for RPs, such as least recently used (LRU) and first in first out (FIFO).

The "pynqpartial" package, developed by the authors, builds on top of the Xilinx pynq package and provides the "PartialReconf" class for managing partial bitstreams and RPs. The package includes modifications to the pynq package's functionality, such as updating bitstream information only when necessary and adding a dictionary for managing partial bitstreams to the "PL" class.

The paper also discusses related work in the field of dynamic partial reconfiguration, including approaches using Xilinx Vivado, SDSoC, FUSE, and other frameworks. It provides an overview of the PYNQ project and its overlay concept, which utilizes the programmable logic in Xilinx Zynq-7000-based devices.

In summary, the paper presents an extension to the Xilinx Python package "pynq" that enables the usage of partial reconfigurable bitstreams in embedded systems design. The authors demonstrate the concept of hybrid classes, the management of partial bitstreams and reconfigurable partitions, and the implementation of replacement strategies for RPs. They also provide the "pynqpartial" package, which integrates with the existing pynq package and simplifies the development of partial reconfigurable overlays.

5. Facial Expression Recognition Based on Improved LeNet-5 CNN

The paper proposes a modified LeNet-5 CNN model for facial expression recognition in the presence of occlusion. The traditional machine learning methods are not robust in such conditions due to the lack of image information and noise interference, resulting in poor recognition rates. The proposed model adds a convolution layer and a pooling layer to the LeNet-5 architecture, allowing for better feature extraction. Low-level features extracted from the network structure are combined with high-level features to construct the classifier. The trainable convolution kernel is used to extract implicit features, which are then reduced using the pooling layer. The SoftMax classifier is employed for classification and recognition.

The paper discusses the impact of occlusion on facial expression recognition and presents two categories of existing methods: the Gabor filtering method and the data reconstruction method. The Gabor filtering method focuses on extracting expression features from unconcluded areas using Gabor filters. The data reconstruction method involves constructing a calibration point model or an image matrix of the occluded area to remove noise and reconstruct occlusion area information. The proposed model combines the advantages of convolutional neural networks (CNNs) in feature extraction with the Gabor filtering method. The occlusion rule is divided into deterministic occlusion and uncertain occlusion. Deterministic occlusion refers to occlusion caused by head or hand movement, while uncertain occlusion involves occlusion by glasses, hair, masks, etc. The paper provides rules for occlusion simulation.

The experimental results section describes the experiments conducted on the CK+ expression database under deterministic and uncertain occlusion conditions. The improved LeNet-5 model is compared with the classic LeNet-5 model and traditional machine learning methods. The recognition performance of the models under different occlusion conditions is analyzed. After conducting an extensive literature review on the topics of object detection,

segmentation, and classification using Convolutional Neural Networks (CNNs), a promising research direction is to develop a unified CNN model that integrates these three tasks into a single framework. By combining the strengths of these individual tasks, the unified model can offer comprehensive and efficient solutions for various computer vision applications.

The proposed unified CNN model aims to address the limitations and challenges associated with separate models for object detection, segmentation, and classification. By leveraging shared features and intermediate representations, the model can achieve improved accuracy, reduce computational complexity, and enhance end-to-end performance.

The development of the unified CNN model would involve several key steps:

1. **Network Architecture Design:** Based on the insights gained from the literature review, a novel network architecture needs to be designed to seamlessly integrate object detection, segmentation, and classification components. This architecture should incorporate suitable feature extraction modules, fusion mechanisms, and prediction layers to handle the different aspects of these tasks effectively.
2. **Data Preparation:** A diverse dataset comprising annotated images that cover a wide range of objects, semantic regions, and classes needs to be curated. This dataset should include samples for object detection, segmentation, and classification, ensuring comprehensive coverage of real-world scenarios.
3. **Model Training:** The unified CNN model will undergo a rigorous training process using the curated dataset. Training strategies such as transfer learning, multi-task learning, or joint optimization techniques can be explored to leverage the shared representations among the tasks and enhance overall performance.
4. **Evaluation Metrics:** A comprehensive evaluation framework needs to be established to assess the performance of the unified CNN model. Evaluation metrics such as mean Average Precision (mAP), Intersection over Union (IoU), accuracy, precision, recall, and F1-score should be considered to measure the effectiveness of the model across object detection, segmentation, and classification tasks.
5. **Benchmarking and Comparison:** To validate the effectiveness and superiority of the proposed unified model, it should be benchmarked against existing state-of-the-art methods for each individual task. Performance comparisons in terms of accuracy, speed, memory consumption, and computational efficiency can provide insights into the advantages of the unified approach.
6. **Application and Deployment:** Finally, the developed unified CNN model can be applied and deployed in various computer vision applications such as autonomous driving, surveillance systems, medical image analysis, or robotics. Real-world testing and validation will provide practical insights into the model's performance and usability.

By embarking on this research endeavor, the development of a unified CNN model for object detection, segmentation, and classification can significantly advance the field of computer vision. It can pave the way for more efficient and comprehensive solutions, eliminating the need for separate models and streamlining the development process for various computer vision applications.

CHAPTER 3

VITIS HLS AND VIVADO FLOW

3.1 Vitis HLS Flow

The Vitis HLS workflow involves several steps to convert a high-level language such as C, C++, or SystemC into an RTL (Register Transfer Level) implementation that can be synthesized and implemented on an FPGA. When we create the project in vitis HLS, first we need to specify the part name or FPGA board on which project will be implemented. There are following major steps which are involved:

1. **Design entry:** This step involves writing the design in a high-level language, like C or C++. The design is typically written as a function or set of functions that define the algorithm or operation that will be implemented.
2. **Verification:** Once the design is written, it must be verified to ensure that it meets the functional requirements and design constraints. Verification involves simulating the design using a testbench to ensure that it behaves correctly.
3. **High-level synthesis:** This is the process of converting the high-level language design into an RTL implementation. The HLS tool analyzes the high-level code and generates RTL code that can be synthesized for implementation on an FPGA.

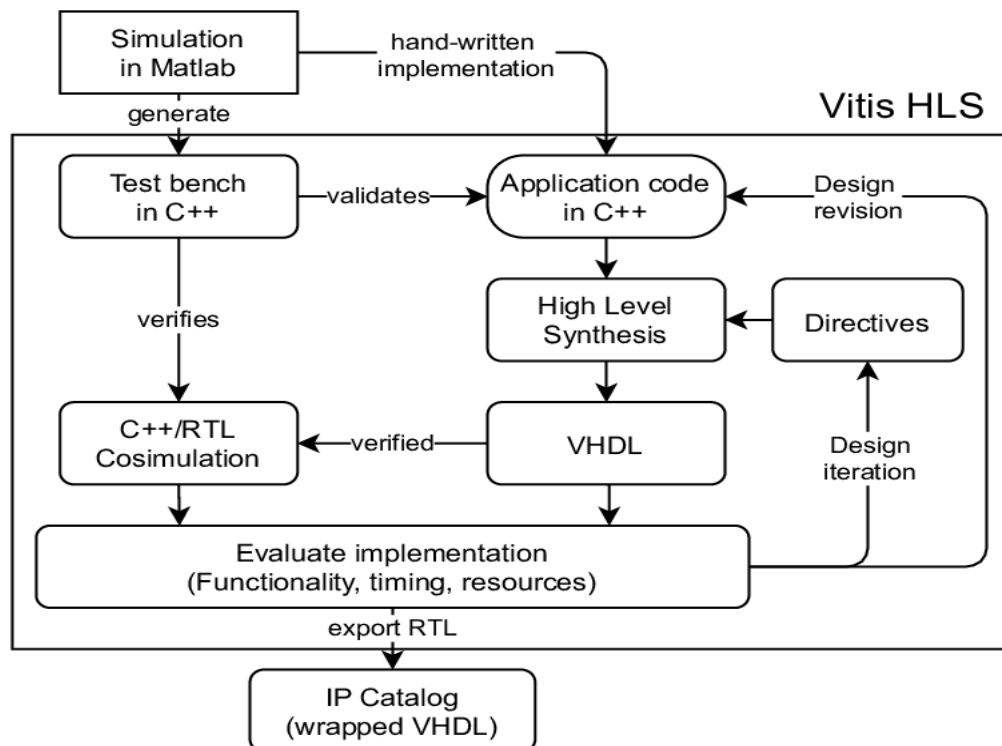


Figure 3.1: Vitis HLS flow

4. Optimization: After generating the RTL code, the HLS tool performs various optimizations to improve the performance, area utilization, and power consumption of the design. These optimizations may include pipelining, loop unrolling, and memory partitioning.
5. Verification: Once the RTL code has been generated and optimized, it must be verified again to make sure that it behaves correctly and also meets the design requirements.
6. Export: The final step is to export the RTL code and constraints to the FPGA implementation tool, such as Xilinx Vivado, for synthesis, implementation, and bitstream generation.

3.2 Vivado Flow

Vivado is a software suite which is provided by Xilinx that enables the development of FPGA designs. The Vivado design flow can be divided into so many stages, each of them performs a specific task in the FPGA design process. The following is a summary of the Vivado design flow:

1. Design entry: This stage involves creating or importing the design files, specifying the top-level module, and setting design parameters such as the target FPGA device. After that we open the IP catalog to create hardware block design. In IP catalog, Xilinx already provides so many default IP which can be further modified according to our requirement.

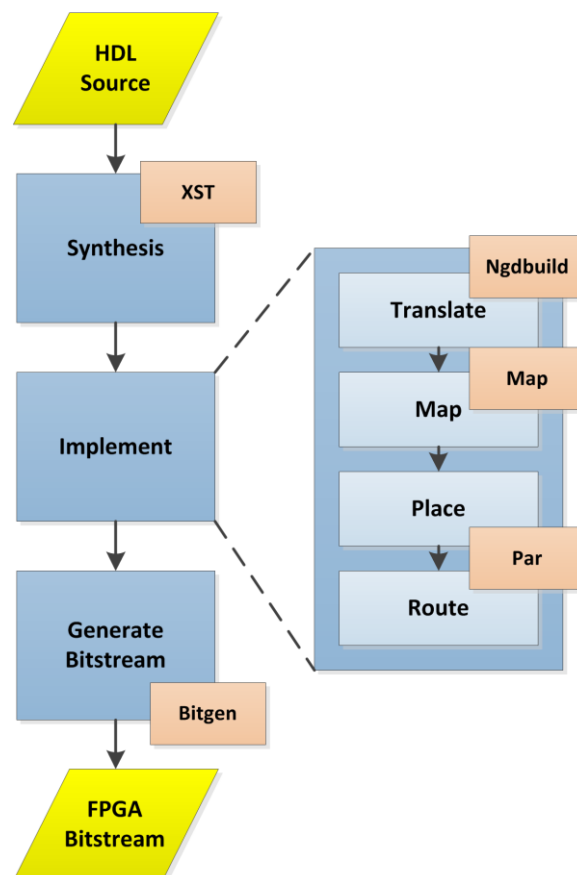


Figure 3.2: Xilinx vivado flow

2. To create block design, we need to add necessary IP and targeted processing system and other pre-defined IP module. Then we need to do some necessary modification in processing system and in predefined IP module. Then we need to do Run automation which is provided by Xilinx for user convenience.
3. After that we validate the block design so that we can find error or any port which is left unconnected. We also check the address editor for the port address where slave and master address will be checked and also we can look the assigned and unassigned address. Then we create the HDL wrapper to make design to work as single module and then will move to synthesis.
4. Synthesis: In this stage, the design is synthesized to create a netlist that describes the behavior of the design in terms of logic gates and flip-flops. Synthesis includes optimizations such as logic optimization, resource sharing, and technology mapping.
5. Simulation: Simulation is used to verify the functionality of the design before it is implemented on the target FPGA device. The design is simulated using a testbench, which provides inputs to the design and captures the outputs.
6. Design analysis: In this stage, the design is analyzed to ensure that it meets the requirements for the target device. This includes checking for timing violations, resource usage, and power consumption.
7. Implementation: In this stage, the design is mapped to the target FPGA device, and the necessary routing and placement information is generated. This includes physical optimization, such as floorplanning, placement, and routing.
8. Bitstream generation: The bitstream is the file that contains the configuration information for the target FPGA device. The bitstream is generated by the Vivado software suite, and it can be downloaded to the target device to configure it for the desired functionality.
9. Hardware validation: Once the bitstream is generated, the design can be loaded onto the target device. The design is then tested to ensure that it behaves correctly and meets the design requirements.

CHAPTER 4

METHODOLOGY

In our work we have presented object detection, classification and segmentation on fpga board. We have studied basics of various existing architecture for object detection, classification and segmentation. First we have shown the implementation of leNet-5 which is used for the the hand written and machine printed characters detection on PYNQ Z2 board using jupyter Notebook. Python programming language is used.

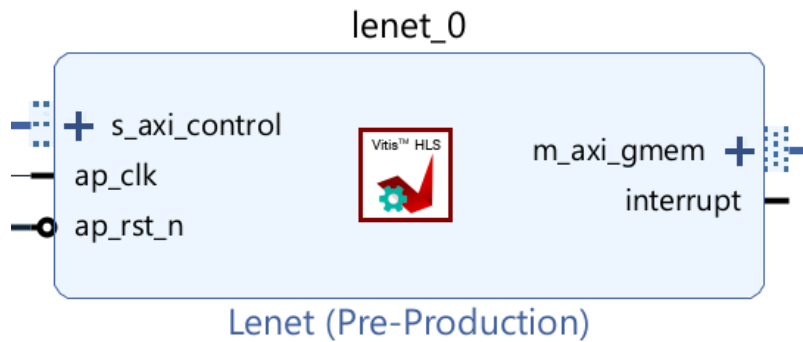


Figure 4.1: LeNet IP (Developed in Vitis HLS)

The above shown diagram is LeNet IP which is developed in Xilinx Vitis HLS. It is further used in Xilinx Vivado to integrate with Zynq processing to make hardware block design.

The hardware block design is created by using two ways, in first way we have used general purpose axis slave interface and in second way we have used high performance axis slave interface of 32 bit width.

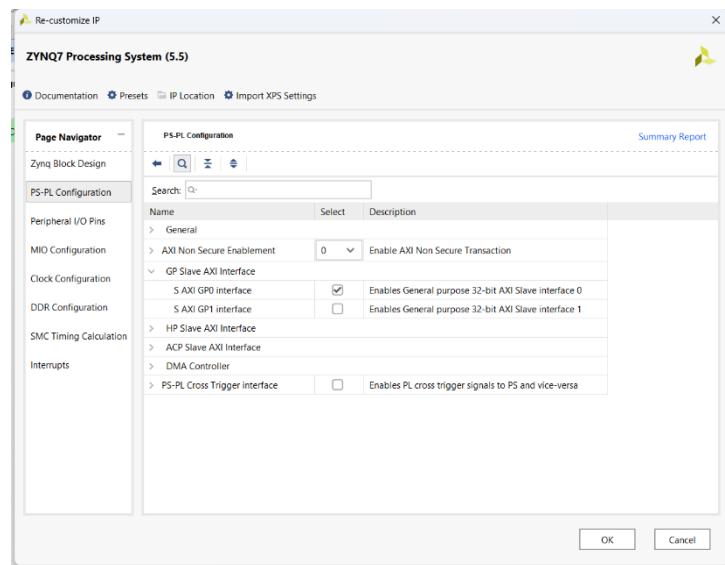


Figure 4.2: 32-bit General Purpose axi slave interface enable

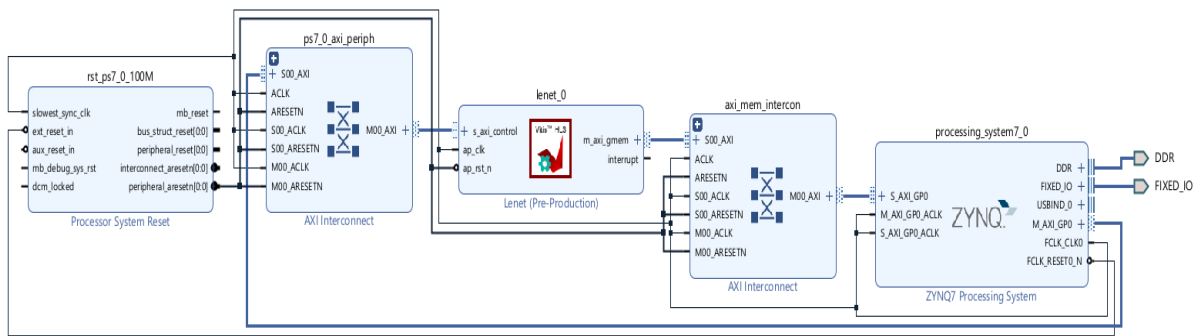


Figure 4.3: LeNet’s Zynq block design (using General purpose Axis slave)

The PS side of Zynq does not transmit the picture data directly to the Lenet IP core but opens up a piece of memory in the PS DDR to store the data of the input picture. Then, the PS side tells the IP core the address of the input picture in DDR through the Master GP interface of Zynq, and the IP core obtains the data of the picture from DDR through the Slave GP interface of Zynq. This is how the m_axi protocol (AXI Master) is used. If the axis protocol (that is, the AXI Stream protocol) is used, the DDR memory is accessed through DMA.

In addition, there is also the s_axilite protocol, which is suitable for the transmission of a small amount of data. It does not need to put the data in the memory first, but the PS side directly transmits the data to the IP core through the Slave GP interface of Zynq.

The PL side clock is provided by the PS side and can be modified in the Zynq IP core. The clock period used in this experiment is 10ns.

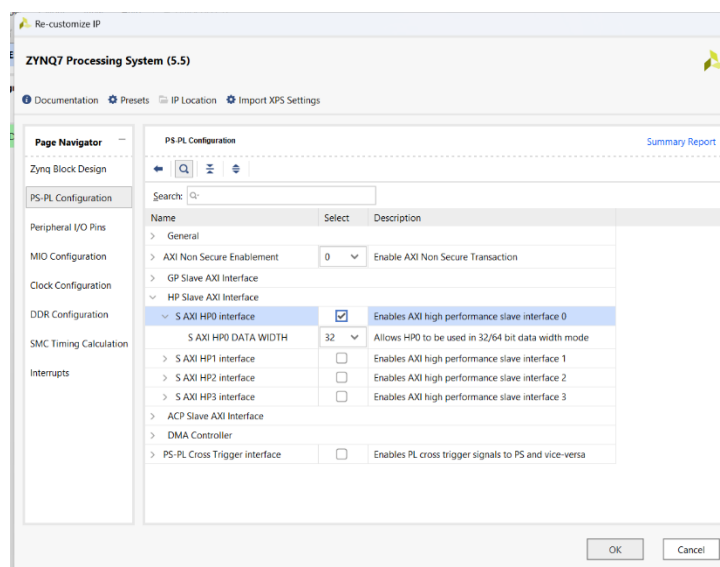


Figure 4.4: 32-bit High-Performance axis slave interface enable.

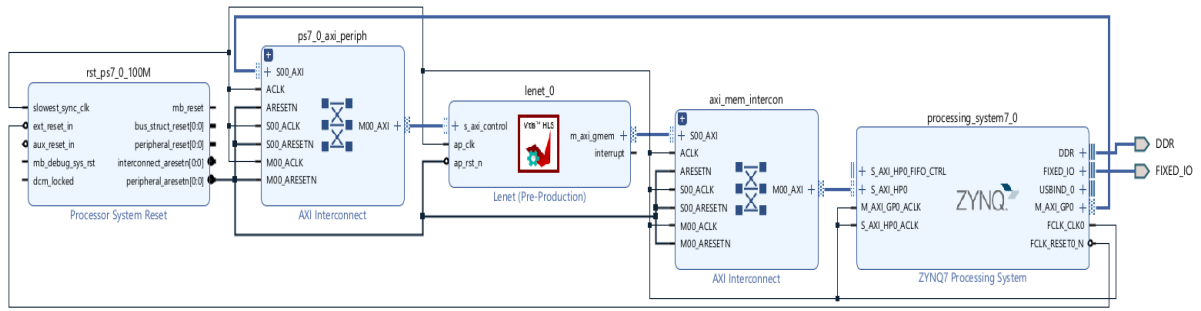


Figure 4.5: LeNet Zynq block design (using 32-bit High-performance Axis slave)

The above shown figure is Hardware block design of LeNet IP with Zynq processing system. After validation of the design, we created the HDL wrapper so that block design can work as single module. Then the design is synthesized to create a netlist that describes the behavior of the design in terms of logic gates and flip-flops. Synthesis includes optimizations such as logic optimization, resource sharing, and technology mapping. After that Simulation is used to verify the functionality of the design before it is implemented on the target FPGA device. The design is simulated using a testbench, which provides inputs to the design and captures the outputs. Next, the design is analyzed to ensure that it meets the requirements for the target device. This includes checking for timing violations, resource usage, and power consumption. the design is mapped to the target FPGA device, and the necessary routing and placement information is generated. This includes physical optimization, such as floorplanning, placement, and routing. The bitstream is generated by the Vivado software suite which contains the configuration information for the target FPGA device, and it can be downloaded to the target device to configure it for the desired functionality. After connecting FPGA board, we have dumped .bit file and .hwh file on FPGA using Jupyter Notebook.

In order to interface the jupyter Notebook with hardware design we will use Overlays. Overlays is the library of pynq that allows the use of bitstream overlays on a PYNQ board. An overlay is a hardware design that can be loaded onto an FPGA to implement custom logic circuits. Apart from that we used some other libraries also, allocate is a function used to allocate contiguous blocks of memory on the PYNQ board. DefaultIP is a class used to define custom IP (Intellectual Property) blocks in a bitstream overlay. CV2 is the Python interface for OpenCV, a popular computer vision library. Time is a standard Python module for measuring time.

MnistDataLoader class is used to load the MNIST dataset in Python. The class has an initialization method that takes input the file paths to the training and test data files, and sets them as instance variables. The class also has a method named read_images_labels, which reads in the image and label data from the given file paths. This method first opens the label file and reads in the magic number and the number of labels. It then reads in the labels using the "array" module and stores them in a list.

Next, the method opens the image file and reads in the magic number, the number of images, the number of rows, and the number of columns. It then reads in the image data using the "array" module and stores it in a list. The image data is then reshaped into a 28x28 NumPy

array and appended to a list of images. Finally, the method returns the list of images and the list of labels. The class also has a method named `load_data`, which calls the `read_images_labels` method to load the training and test data and returns them as tuples. Specifically, this method returns a tuple containing the training data and a tuple containing the test data.

`%matplotlib inline` magic command is used in Jupyter notebooks that allows for inline plotting of graphs. It instructs the notebook to display the plot on the same page below the code cell where the plot was created. Then `random` and `matplotlib.pyplot` modules are used for generating random numbers and creating plots, respectively.

The `show_images` function is used next, which takes in a list of images and a list of corresponding title texts. It uses the `plt` module to create a grid of images and titles. It has 5 columns and variable number of rows depending upon the number of images. It then iterates over the images and title texts using the `zip` function to create a tuple of each corresponding image and title. It displays each image using `plt.imshow` and sets the title using `plt.title`.

Then we define `images_50` which will contain the resized images. It is using a `for` loop to iterate through the first 50 images in the `x_test` array. For each image, it is first converting it to a NumPy array of unsigned 8-bit integers (`dtype='u1'`) using `np.array()`. Then, it is resizing the image using OpenCV's `cv2.resize()` function. The resized image will have a width and height of 32 pixels. The `interpolation` parameter is set to `cv2.INTER_LINEAR_EXACT` which specifies that the function should use bilinear interpolation. At last, the resized image is appended to the `images_50` list using the `append()` method.

Now we defines a custom driver class `LeNetDriver` for an IP core that implements the LeNet-5 neural network architecture. The driver class is derived from the `DefaultIP` class provided by the PYNQ library, which is a base class for AXI IP cores. The `bindto` attribute specifies the vendor, library, name, and version (VLNV) of the IP core that this driver class is intended to be bound to. In this case, it is set to `xilinx.com:hls:lenet:1.0`. The `__init__` method is the constructor of the class, and it initializes various attributes of the class, such as the address of the input image (`img_in`), the control signal (`ap_ctrl`), the return signal (`ap_return`), and the size of the input image (`img_size`). It also allocates two input image buffers (`img_in_buf_1` and `img_in_buf_2`) using the `allocate` function provided by the PYNQ library.

The `batch_predict` method is similar to the `predict` method, but it uses a double buffer implementation to improve the throughput for batch predictions. It takes a batch of input images (`img_in`) as input, writes the first image to the first buffer (`img_in_buf_1`), and then processes the remaining images in a loop. For each image, it writes the image to the buffer that is not currently being used for prediction, sets the control signal to 0x01, waits for the prediction to complete, and then reads the predicted label from the return signal. After all the images have been processed, it waits for the last prediction to complete, and then returns an array of predicted labels for the entire batch of input images.

Now, The trained LeNet model is loaded onto the FPGA using the `Overlay` function from the `pynq` library. `Overlay("lenet_training1.bit")` creates an overlay object for the specified bitstream file "lenet_training1.bit". Then the `batch_predict` method of the `lenet` object is called to perform inference on the 50 images in the `images_50` list. The results of the

predictions are stored in the `output` list. The `batch_predict` method is used to predict the labels of multiple images in parallel using double buffering, which can improve performance compared to predicting the labels of the images one at a time.

The predicted labels in `output` are then compared to the expected labels in the `standard` list to calculate the accuracy of the model. The `coincide` variable is incremented every time a predicted label matches the expected label. The final accuracy is calculated as the ratio of the number of coincidences to the total number of images in the test set. The accuracy is printed out to the console.

Second, we have implemented the image recognition with the help of (BNN). This network has 6 convolutional layers, 3 max pool layers and 3 fully connected layers. In this network, the weights and activations are binarized to only take values of +1 or -1. This binary representation reduces the memory and computation requirements of the network, making it more efficient in terms of speed, power consumption, and memory usage. Binary Neural Networks are particularly useful in resource-constrained environments, such as mobile devices and embedded systems, where conventional neural networks are too large to be deployed. BNNs can also provide more robustness to noisy inputs and prevent overfitting, which is a common problem in deep learning. There are several challenges associated with BNNs, such as the limited expressiveness of the binary representation, which can lead to reduced accuracy in certain tasks. Additionally, training BNNs can be more challenging than traditional neural networks because of the non-differentiability of the binary function. Researchers are actively working on developing new methods and techniques to overcome these challenges and improve the performance of BNNs. BNNs have shown promising results in various applications, including image classification, object detection, and speech recognition.

In this work, classifier is instantiated instead of overlay library. To instantiate the classifier, first we created the classifier which will download the correct bitstream on the device and also load the weights trained on the specified dataset. First of all we imported the BNN module, which allows for the creation and deployment of binary neural networks on hardware platforms such as FPGAs. Then it calls the `available_params` function with the argument `bnn.NETWORK_CNW1A1`. This function returns the available quantized weight parameters for the CNV network architecture, which is a convolutional neural network that is commonly used for image classification tasks.

After that `CnvClassifier` class instance is created from the BNN module. This class takes three arguments: the network architecture (in this case, `bnn.NETWORK_CNW1A1`), the name of the dataset to be used for training (in this case, `'road-signs'`), and the target runtime hardware (in this case, `bnn.RUNTIME_HW`). The `CnvClassifier` class is used to train and deploy binary neural networks on hardware platforms such as the Xilinx PYNQ-Z1 and PYNQ-Z2 development boards. The `road-signs` dataset is a dataset of road signs commonly used for benchmarking in computer vision research. The `RUNTIME_HW` argument specifies that the BNN should be deployed on hardware for acceleration, rather than running on the CPU. By calling `print(classifier.classes)`, the code is printing the list of classes.

The necessary modules is imported to work with images, specifically the Python Imaging Library (PIL), NumPy, and the display function from IPython. The `listdir` and `isfile` functions is also imported from the `os` module to work with the filesystem.

Next, an empty list called `images` is created. This list will be populated with the images loaded from the directory. Then for loop is used to iterate through each image in `imgList`. For each image, the code loads the image using the `Image.open` function from PIL and appends it to the `images` list. It then resizes the image to a height and width of 64 pixels using the `thumbnail` method of the PIL Image object, with the `Image.ANTIALIAS` filter to reduce the size of the image without losing too much detail. After that the thumbnail of the image is displayed using the `display` function from IPython.

```
['20 Km/h', '30 Km/h', '50 Km/h', '60 Km/h', '70 Km/h', '80 Km/h', 'End 80 Km/h', '100 Km/h', '120 Km/h', 'No overtaking', 'No overtaking for large trucks', 'Priority crossroad', 'Priority road', 'Give way', 'Stop', 'No vehicles', 'Prohibited for vehicles with a permitted gross weight over 3.5t including their trailers, and for tractors except passenger cars and buses', 'No entry for vehicular traffic', 'Danger Ahead', 'Bend to left', 'Bend to right', 'Double bend (first to left)', 'Uneven road', 'Road slippery when wet or dirty', 'Road narrows (right)', 'Road works', 'Traffic signals', 'Pedestrians in road ahead', 'Children crossing ahead', 'Bicycles prohibited', 'Risk of snow or ice', 'Wild animals', 'End of all speed and overtaking restrictions', 'Turn right ahead', 'Turn left ahead', 'Ahead only', 'Ahead or right only', 'Ahead or left only', 'Pass by on right', 'Pass by on left', 'Roundabout', 'End of no-overtaking zone', 'End of no-overtaking zone for vehicles with a permitted gross weight over 3.5t including their trailers, and for tractors except passenger cars and buses', 'Not a road sign']
```

`Classifier` class is used to classify a list of images, `images`. The `classify_images` method of the `CnnClassifier` class is called with the `images` list as an argument. This method processes each image in the list and returns a list of integers representing the index of the class that each image belongs to. The resulting list of integers is stored in the `results` variable. The identified classes is printed out using the `print` function, formatted using a string that includes the `results` variable. The `format` method of the string is used to insert the `results` variable into the string. A “for” loop is used to iterate over each index in the `results` list. For each index, the `class_name` method of the `CnnClassifier` object is called with the index as an argument. This method returns a string representing the name of the class that corresponds to the index. The code then prints out the identified class name using the `print` function, formatted using a string that includes the name of the class.

```
Inference took 915.00 microseconds, 305.00 usec per image
Classification rate: 3278.69 images per second
Identified classes: [41 27 14]
Identified class name: End of no-overtaking zone
Identified class name: Pedestrians in road ahead
Identified class name: Stop
```

Image module is imported from the PIL (Python Imaging Library) library and then loads an image file from the located address using the `Image.open()` method. The opened image is assigned to the variable `im`. Once the image is loaded, it can be manipulated using the various methods provided by the PIL library. This can include operations such as cropping, resizing, filtering, and more. A list of image patches, `images`, is created and a corresponding list of bounding boxes, `bounds`, for a given image, `im`.

First `images` and `bounds` is created to empty lists. It then iterates over two different stride values, 64 and 96, which correspond to the sizes of the image patches to be extracted. For each stride value, the code calculates the number of tiles that can fit along the width and height of the image, `x_tiles` and `y_tiles`, respectively.

Then looped over each tile by iterating over each row and column of the tile grid. For each tile, a bounding box is defined based on the current stride value, row, and column. The bounding box is defined as a tuple of four values: (left, upper, right, lower). The left and upper coordinates are determined by multiplying the stride value with the current column and row, respectively. The right and lower coordinates are calculated by adding the stride value to the left and upper coordinates. Then the bounding box is checked within the dimensions of the

original image. If so, the code crops the image using the bounding box coordinates and appends the resulting patch to the `images` list. The bounding box tuple is also appended to the `bounds` list. The number of patches is printed out that were extracted by printing the length of the `images` list.

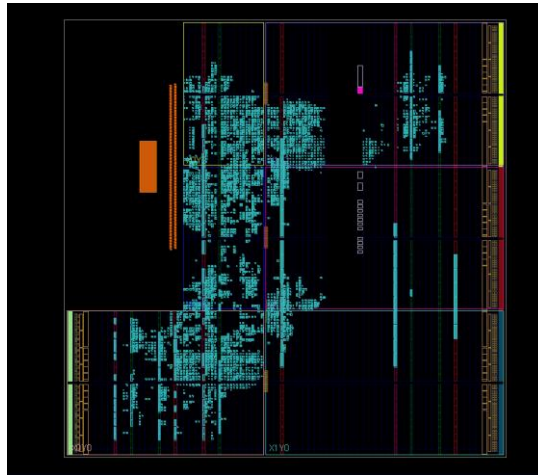
Object detection on the image patches is performed that were extracted in the previously. It uses a pre-trained BNN classifier, `classifier`, to classify each patch and identify the presence of a stop sign. The code first creates a boolean array, `stop`, by comparing the results array returned by `classifier.classify_images()` with the integer value of 14, which corresponds to the class label for stop signs. The `nonzero()` method is then used to find the indices of all elements in the `stop` array that are not equal to zero, which correspond to the indices of the patches that were classified as stop signs.

Then loads the original image file again using the `Image.open()` method and creates an `ImageDraw` object, `draw2`, to draw rectangles around the detected stop signs. The loops over the indices of the detected stop sign patches and draws a red rectangle around the corresponding region in the original image using the `draw.rectangle()` method. The coordinates of each rectangle are defined by the corresponding bounding box from the `bounds` list. Finally, the modified image with the detected stop signs is displayed by displaying the image using `im2.show()` or by returning the `im2` object.

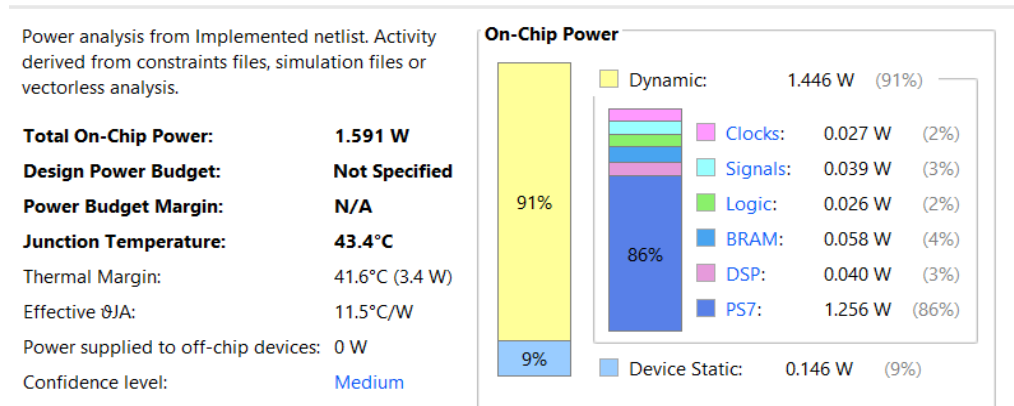
CHAPTER 5

RESULTS

Implemented Design on FPGA:



Power Consumption:



Output of LeNet:




Given below is the output of lenet model, When IP core obtains the data from DDR through the 32-bit general purpose axi slave interface of Zynq processing system is used. In that output we can see the accuracy and the total time taken by the model.

```
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 8 1 7 4 2 3 5 1 2 9 4]
Elapsed time: 116.85 ms
accuracy = 96.00 %
```

Given below is the output of lenet model, When IP core obtains the data from DDR through the 32-bit High Performance axi slave interface of Zynq processing system is used. In that output we can see the accuracy is decreased in comparison to above output but the total time taken by the model is drastically decreased.

```
[7 2 9 0 4 9 4 9 5 9 0 6 5 0 1 5 5 7 3 4 9 8 6 5 4 8 7 4 2 1 3 1 3 0 3 2 7
 1 3 3 1 7 4 2 3 5 3 2 9 4]
Elapsed time: 67.50 ms
accuracy = 70.00 %
```

Output of BNN model:

	Inference took 915.00 microseconds, 305.00 usec per image
	Classification rate: 3278.69 images per second
	Identified classes: [41 27 14]
	Identified class name: End of no-overtaking zone
	Identified class name: Pedestrians in road ahead
	Identified class name: Stop

Input Image

Output (object classified)



Input Image



Output Image (Object located)

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

In our Project, we have successfully implemented the CNN based LeNet-5 and BNN based Object detection and classification on FPGA board (Pynq Z2 board). We used the jupyter notebook to interface the hardware and to implement the project. Python programming language is used in jupyter notebook. BNN-based object detection has the potential to significantly improve the efficiency and accuracy of object detection systems, particularly in resource-constrained environments. By reducing the number of parameters and computations required, BNNs can enable the development of faster and more efficient object detection systems that can operate in real-time.

However, BNNs also present some challenges, particularly in terms of training and optimization. BNNs require specialized training algorithms that can handle the non-differentiable nature of binary activations and weights. Additionally, BNNs are more sensitive to noise and quantization errors, which can affect their performance. Despite these challenges, the potential benefits of BNN-based object detection are significant, and ongoing research is focused on developing more effective training and optimization techniques to improve their performance.

6.2 Future Scope

BNN-based object detection has promising future potential due to the significant reduction in memory and computational requirements of BNNs. BNN-based object detection can lead to the development of faster, more efficient, and more accurate object detection systems that are well-suited for use in resource-constrained devices such as smartphones and embedded systems.

Currently, object detection systems based on deep learning techniques such as convolutional neural networks (CNNs) require large amounts of memory and computational resources, which can limit their application in devices with limited resources. BNN-based object detection can address this limitation by reducing the number of parameters and computations required, while maintaining high accuracy. BNNs can also be combined with other techniques such as pruning and quantization to further reduce the memory and computational requirements of object detection systems. This can lead to even faster and more efficient systems that can operate in real-time and in resource-constrained environments.

APPENDIX A

CODE ATTACHMENTS

A1. LeNet – Vitis HLS source code

A1.1 Net structure

```
1  |include "op.h"
2  |include "param.h"
3
4  //define TEST
5
6  #ifdef TEST
7  |include "tbutils.h"
8  #endif
9
10 using namespace mylenet;
11
12 int lenet(INPUT (&img)[32][32])
13 {
14     #pragma HLS INTERFACE s_axilite port=return
15     #pragma HLS INTERFACE m_axi port=img offset=slave depth=1024
16     // preprocess
17     INPUT img_preprocessed[1][32][32] = {0};
18     preprocess(img, img_preprocessed);
19
20     #ifdef TEST
21         arr2txt("img_preprocessed_0.txt", img_preprocessed[0]); //
22     #endif
23
24     // compute
25     // (1) conv1 and relu
26     BIAS conv1_relu_rst[6][28][28] = {0};
27     DATA pool1_in[6][28][28] = {0};
28     conv_relu(img_preprocessed, conv1_weights, conv1_bias, conv1_relu_rst);
29     scale(conv1_relu_rst, n_conv1, m_conv1, pool1_in);
30
31     #ifdef TEST
32         arr2txt("conv1_relu_rst_0.txt", conv1_relu_rst[0]);
33     #endif
34
35     // (2) pooling1
36     DATA pool1_rst[6][14][14] = {0};
37     maxpooling(pool1_in, 2, 2, pool1_rst);
38
39     // (3) conv2 and relu
40     BIAS conv2_relu_rst[16][10][10] = {0};
41     DATA pool2_in[16][10][10] = {0};
42     conv_relu(pool1_rst, conv2_weights, conv2_bias, conv2_relu_rst);
43     scale(conv2_relu_rst, n_conv2, m_conv2, pool2_in);
44
45     #ifdef TEST
46         arr2txt("conv2_relu_rst_0.txt", conv2_relu_rst[0]);
47     #endif
48     // (4) pool2
49     DATA pool2_rst[16][5][5] = {0};
50     maxpooling(pool2_in, 2, 2, pool2_rst);
51
52     // (5) flatten
53     DATA flattened[400] = {0};
54     flatten(pool2_rst, flattened);
55
56     // (6) fc1
57     BIAS fc1_relu_rst[120] = {0};
58     DATA fc2_in[120] = {0};
59     fc_relu(flattened, fc1_weights, fc1_bias, fc1_relu_rst);
60     scale(fc1_relu_rst, n_fc1, m_fc1, fc2_in);
61
62     #ifdef TEST
63         arr1d2txt("fc1_relu_rst.txt", fc1_relu_rst);
64     #endif
65
66     // (7) fc2
67     BIAS fc2_relu_rst[84] = {0};
68     DATA fc3_in[84] = {0};
69     fc_relu(fc2_in, fc2_weights, fc2_bias, fc2_relu_rst);
70     scale(fc2_relu_rst, n_fc2, m_fc2, fc3_in);
71
72     #ifdef TEST
73         arr1d2txt("fc2_relu_rst.txt", fc2_relu_rst);
74     #endif
75 }
```


A1.2 Net structure header

```
#ifndef _NET_H_
#define _NET_H_

#include "op.h"

int lenet(mylenet::INPUT (&img)[32][32]);

#endif
```

A1.3 Parameters

```
#ifndef _PARAM_
#define _PARAM_
#include "../op.h"
#include "../param/conv1_bias.h"
#include "../param/conv1_weights.h"
#include "../param/conv2_bias.h"
#include "../param/conv2_weights.h"
#include "../param/fc1_bias.h"
#include "../param/fc1_weights.h"
#include "../param/fc2_bias.h"
#include "../param/fc2_weights.h"
#include "../param/fc3_bias.h"
#include "../param/fc3_weights.h"

int n_conv1 = 17;
int n_conv2 = 16;
int n_fc1 = 15;
int n_fc2 = 14;

mylenet::BIAS m_conv1 = 129;
mylenet::BIAS m_conv2 = 69;
mylenet::BIAS m_fc1 = 31;
mylenet::BIAS m_fc2 = 39;

#endif
```

A1.4 Testbench header

```
#ifndef _TB_UTILS_H_
#define _TB_UTILS_H_

#include "op.h"
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>

template<typename T, size_t M, size_t N>
void disp_arr(T (&arr)[M][N])
{
    for (size_t i = 0; i < M; i++) {
        for (size_t j = 0; j < N; j++) {
            std::cout << arr[i][j] << ' ';
        }
        std::cout << std::endl;
    }
}

template<typename T, size_t M, size_t N>
void txt2arr(std::string filename, T (&arr)[M][N])
{
    std::ifstream infile;
    infile.open(filename);
    for (size_t i = 0; i < M; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            infile >> arr[i][j];
        }
    }
    infile.close();
}

template<typename D, typename S, size_t M, size_t N>
void copy_arr(D (dst)[M][N], S (&src)[M][N])
{
    for (size_t i = 0; i < M; i++) {
        for (size_t j = 0; j < N; j++) {
            dst[i][j] = src[i][j];
        }
    }
}

template<typename T, size_t M, size_t N>
void arr2txt(std::string filename, T (&arr)[M][N], int width=2, int precision=4)
{
    std::ofstream outfile;
    outfile.open(filename);
    outfile.precision(precision);
    for (size_t i = 0; i < M; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            outfile << std::setw(width) << arr[i][j] << " ";
        }
        outfile << std::endl;
    }
    outfile << std::endl;
    outfile.close();
}

template<typename T, size_t M>
void arr1d2txt(std::string filename, T (&arr)[M], int width=2, int precision=4)
{
    std::ofstream outfile;
    outfile.open(filename);
    outfile.precision(precision);
    for (size_t i = 0; i < M; i++)
    {
        outfile << std::setw(width) << arr[i] << std::endl;
    }
    outfile.close();
}
```

A1.5 Testbench

```
#include "tbutils.h"
#include "op.h"
#include "net.h"
#include <string>

#ifdef _WIN32
#include <windows.h>
DWORD st, et;
#endif

#ifdef __linux__
#include <sys/time.h>
timeval st, et;
#endif

int main()
{
    float _img[32][32] = {0};
    mylenet::INPUT img[32][32] = {0};
    int rst;
    std::string str = "./imgs/img_";
    std::string img_path = "./imgs/img_";

    #ifdef _WIN32
    DWORD elapsed_time = 0;
    #endif
    #ifdef __linux__
    double elapsed_time = 0;
    #endif

    int rst_vec[50];

    int tb_iter = 50;

    for (int i = 0; i < tb_iter; i++) {
        img_path = str + std::to_string(i) + ".txt";
        txt2arr(img_path, _img);
        copy_arr(img, _img);

        #ifdef _WIN32
        st = GetTickCount();
        #endif
        #ifdef __linux__
        gettimeofday(&st, NULL);
        #endif

        rst = lenet(img);
        rst_vec[i] = rst;

        #ifdef _WIN32
        et = GetTickCount();
        elapsed_time += et - st;
        #endif
        #ifdef __linux__
        gettimeofday(&et, NULL);
        elapsed_time += (et.tv_sec - st.tv_sec) * 1000 + (et.tv_usec - st.tv_usec) / 1000;
        #endif

        std::cout << rst << " ";
    }
    std::cout << std::endl;

    std::cout << "\nPrediction of 50 pictures takes time "<< elapsed_time << " ms\n";

    int gold[50] = {7, 2, 1, 0, 4, 1, 4, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4, 0, 7, 4, 0, 1, 3, 1, 3, 4, 7, 2, 7, 1, 2, 1, 1, 7, 4, 2, 3, 5, 1, 2, 4, 4};
    int coincide = 0;
    for (int i = 0; i < tb_iter; i++) {
        if (rst_vec[i] == gold[i]) {
            coincide++;
        }
    }

    std::cout << "Precision = " << (double) coincide / tb_iter * 100. << "%" << std::endl;

    return 0;
}
```

A2. LeNet Python code in Jupyter Notebook

A2.1 Import Packages

```
In [1]: from pynq import Overlay
from pynq import allocate
from pynq import DefaultIP
import cv2
import time
```

A2.2 Data Loader

```
In [2]: import numpy as np
import struct
from array import array
from os.path import join

class MnistDataLoader(object):
    def __init__(self, training_images_filepath, training_labels_filepath,
                  test_images_filepath, test_labels_filepath):
        self.training_images_filepath = training_images_filepath
        self.training_labels_filepath = training_labels_filepath
        self.test_images_filepath = test_images_filepath
        self.test_labels_filepath = test_labels_filepath

    def read_images_labels(self, images_filepath, labels_filepath):
        labels = []
        with open(labels_filepath, 'rb') as file:
            magic, size = struct.unpack(">II", file.read(8))
            if magic != 2049:
                raise ValueError('Magic number mismatch, expected 2049, got {}'.format(magic))
            labels = array("B", file.read())

        with open(images_filepath, 'rb') as file:
            magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
            if magic != 2051:
                raise ValueError('Magic number mismatch, expected 2051, got {}'.format(magic))
            image_data = array("B", file.read())
            images = []
            for i in range(size):
                images.append([0] * rows * cols)
            for i in range(size):
                img = np.array(image_data[i * rows * cols:(i + 1) * rows * cols])
                img = img.reshape(28, 28)
                images[i][:] = img

        return images, labels

    def load_data(self):
        x_train, y_train = self.read_images_labels(self.training_images_filepath, self.training_labels_filepath)
        x_test, y_test = self.read_images_labels(self.test_images_filepath, self.test_labels_filepath)
        return (x_train, y_train), (x_test, y_test)
```

```
In [3]: %matplotlib inline
import random
import matplotlib.pyplot as plt

input_path = './'
training_images_filepath = join(input_path, 'train-images-idx3-ubyte/train-images.idx3-ubyte')
training_labels_filepath = join(input_path, 'train-labels-idx1-ubyte/train-labels.idx1-ubyte')
test_images_filepath = join(input_path, 't10k-images-idx3-ubyte/t10k-images.idx3-ubyte')
test_labels_filepath = join(input_path, 't10k-labels-idx1-ubyte/t10k-labels.idx1-ubyte')

def show_images(images, title_texts):
    cols = 5
    rows = int(len(images)/cols) + 1
    plt.figure(figsize=(30,20))
    index = 1
    for x in zip(images, title_texts):
        image = x[0]
        title_text = x[1]
        plt.subplot(rows, cols, index)
        plt.imshow(image, cmap=plt.cm.gray)
        if (title_text != ''):
            plt.title(title_text, fontsize = 15);
        index += 1

mnist_dataloader = MnistDataLoader(training_images_filepath, training_labels_filepath, test_images_filepath, test_labels_filepath)
(x_train, y_train), (x_test, y_test) = mnist_dataloader.load_data()
```

A2.3 Upsample to 32 bit

```
In [5]: images_50 = []
        for img in x_test[0:50]:
            img = np.array(img, dtype='u1')
            images_50.append(cv2.resize(img, dsize=(32, 32), interpolation=cv2.INTER_LINEAR_EXACT))
```

A2.4 IP Driver

```
In [5]: class LeNetDriver(DefaultIP):
        bindto = ["xilinx.com:hls:lenet:1.0"]
        def __init__(self, description):
            super().__init__(description=description)
            self.img_in = 0x18
            self.ap_ctrl = 0x00
            self.ap_return = 0x10
            self.img_size = 32
            self.img_in_buf_1 = allocate(shape=(self.img_size, self.img_size), dtype='u1')
            self.img_in_buf_2 = allocate(shape=(self.img_size, self.img_size), dtype='u1')

        def predict(self, img_in):
            self.write(self.img_in, self.img_in_buf_1.physical_address)
            np.copyto(self.img_in_buf_1, np.uint8(img_in))

            self.write(self.ap_ctrl, 0x01)
            while self.read(self.ap_ctrl) == 0x01:
                pass

            return self.read(self.ap_return)

        # double buffer implementation for batch prediction
        def batch_predict(self, img_in):
            flag = False
            output = np.zeros(len(img_in), dtype=int)

            self.write(self.img_in, self.img_in_buf_1.physical_address)
            np.copyto(self.img_in_buf_1, np.uint8(img_in[0]))

            for i in range(1, len(img_in)):
                self.write(self.ap_ctrl, 0x01)
                current_buf = self.img_in_buf_1 if flag else self.img_in_buf_2
                self.write(self.img_in, current_buf.physical_address)
                np.copyto(current_buf, np.uint8(img_in[i]))
                while self.read(self.ap_ctrl) == 0x01:
                    pass
                output[i-1] = self.read(self.ap_return)
                flag = not flag

            self.write(self.ap_ctrl, 0x01)
            while self.read(self.ap_ctrl) == 0x01:
                pass
            output[len(img_in)-1] = self.read(self.ap_return)

            return output
```

A2.5 IP Call

```
overlay = Overlay("lenet_training1.bit")
lenet = overlay.lenet_0

st = time.time()
output = lenet.batch_predict(images_50)
et = time.time()
print(output)
print("Elapsed time: %.2f ms" % ((et - st) * 1000))

standard = [7, 2, 1, 5, 7, 1, 6, 9, 5, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4, 0, 7, 4, 0, 1, 3, 1, 3, 4,
coincide = 0
for i in range(50):
    if (standard[i] == output[i]):
        coincide += 1
accuracy = coincide / 50
print("accuracy = %.2f %" % (accuracy * 100))
```

A.3 BNN Python Code in Jupyter Notebook

```
In [1]: import sys
import os, platform
import json
import numpy as np
import cv2
import ctypes

from PIL import Image
from datetime import datetime

import qnn
from qnn import TinierYolo
from qnn import utils
sys.path.append("/opt/darknet/python/")
from darknet import *

from matplotlib import pyplot as plt
%matplotlib inline
```

```
In [3]: from PIL import Image
import numpy as np
from os import listdir
from os.path import isfile, join
from IPython.display import display

imgList = [f for f in listdir("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/") if isfile(join("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/", f))]

images = []

for imgFile in imgList:
    img = Image.open("/home/xilinx/jupyter_notebooks/bnn/pictures/road_signs/" + imgFile)
    images.append(img)
    img.thumbnail((64, 64), Image.ANTIALIAS)
    display(img)
```

```
In [4]: results = classifier.classify_images(images)
print("Identified classes: {}".format(results))
for index in results:
    print("Identified class name: {}".format(classifier.class_name(index)))
```

```
In [5]: sw_class = bnn.CnnClassifier(bnn.NETWORK_CNVM1A1, "road-signs", bnn.RUNTIME_SW)

results = sw_class.classify_images(images)
print("Identified classes: {}".format(results))
for index in results:
    print("Identified class name: {}".format(classifier.class_name(index)))
```

```
In [6]: from PIL import Image
image_file = "/home/xilinx/jupyter_notebooks/bnn/pictures/street_with_stop.JPG"
im = Image.open(image_file)
im
```

```
In [7]: images = []
bounds = []
for s in [64, 96]:
    stride = s // 4
    x_tiles = im.width // stride
    y_tiles = im.height // stride

    for j in range(y_tiles):
        for i in range(x_tiles):
            bound = (stride * i, stride * j, stride * i + s, stride * j + s)
            if bound[2] <= im.width and bound[3] < im.height:
                c = im.crop(bound)
                images.append(c)
                bounds.append(bound)

print(len(images))
```

```
In [8]: results = classifier.classify_images(images)
stop = results == 14
indicies = []
indicies = stop.nonzero()[0]
from PIL import ImageDraw
im2 = Image.open(image_file)
draw2 = ImageDraw.Draw(im2)
for i in indicies:
    draw2.rectangle(bounds[i], outline='red')

im2
```

REFERENCES

- [1]. M K Aparna Nair, Police Manoj Kumar Reddy, Y.L. Abijith, Venkatesh Rajagopalan, and J Soumya, "Hardware Implementation of Network Interface Architecture for RISC-V based NoC-MPSoC Framework".
- [2]. J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoc Platform Using the PYNQ Framework," in IEEE Access, vol. 8, pp. 129012-129031, 2020, doi: 10.1109/ACCESS.2020.3008954.
- [3]. "Implementation of Lenet-5 Convolution Neural Network Based on FPGAwith HLS"
Dai Rongshi School of Electronic Science and Engineering Southeast University
Nanjing, China Tang Yongming School of Electronic Science and Engineering
Southeast University Nanjing, China.
- [4]. G. Sahu, A. Seal, A. Yazidi and O. Krejcar, "A Dual-Channel Dehaze-Net for Single Image Dehazing in Visual Internet of Things Using PYNQ-Z2 Board," in IEEE Transactions on Automation Science and Engineering, 2022, doi: 10.1109/TASE.2022.3217801.
- [5]. A. Maraoui, S. Messaoud, S. Bouaafia, A. C. Ammari, L. Khriji and M. Machhout, "PYNQ FPGA Hardware implementation of LeNet-5-Based Traffic Sign Recognition Application," 2021 18th International Multi-Conference on Systems, Signals & Devices (SSD), Monastir, Tunisia, 2021, pp. 1004-1009, doi: 10.1109/SSD52085.2021.9429480.
- [6]. B. Janßen, P. Zimprich and M. Hübner, "A dynamic partial reconfigurable overlay concept for PYNQ," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 2017, pp. 1-4, doi: 10.23919/FPL.2017.8056786.
- [7]. Convolutional Neural Network and Maxpooling Architecture on Zynq SoC FPGA Y.A. Bachtiar University Center of Excellence on Microelectronics Bandung Institute of Technology Bandung, Indonesia T.Adiono University Center of Excellence on Microelectronics Bandung Institute of Technology Bandung, Indonesia.
- [8]. Implementing Lightweight Image/Video Encryption Cores on Xilinx Zynq Sh.M. Sadaghiani, Department of ECE, University of Tabriz, Tabriz, Iran M.Zolfy Department of ECE, University of Tabriz, Tabriz, Iran.

- [9]. F. Kästner, B. Janßen, F. Kautz, M. Hübner and G. Corradi, "Hardware/Software Codesign for Convolutional Neural Networks Exploiting Dynamic Partial Reconfiguration on PYNQ," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 2018, pp. 154-161, doi: 10.1109/IPDPSW.2018.00031.
- [10]. Accelerating Deep Neural Networks Using FPGAs and ZYNQ Han Sung Lee, Jae Wook Jeon Dept. of Electrical and Computer Engineering Sungkyunkwan University Suwon, Republic of Korea.
- [11]. Analysis of the Programmable Soft IP-cores Implementation for FPGAs Vasilii M. Khvatov, Daniil A. Zheleznikov, Sergey V. Gavrilov Institute for Design Problems in Microelectronics of Russian Academy of Sciences (IPPM RAS) Moscow, Russia.
- [12]. Classification of Garments from Fashion MNIST Dataset Using CNN LeNet-5 Architecture, Mohammed Kayed Faculty of Computers and Artificial Intelligence, Ahmed Anter Faculty of Computers and Artificial Intelligence, Hadeer Mohamed Faculty of Science, Beni-Suef University, DOI: 10.1109/ITCE48509.2020.9047776.
- [13]. Facial Expression Recognition Based on Improved LeNet-5 CNN Guan Wang, Jun Gong College of Information Science and Engineering, Northeastern University, Shenyang110819, DOI: 10.1109/CCDC.2019.8832535.
- [14]. Aman Sharma, Vijander Singh, Asha Rani, "Implementation of CNN on Zynq based FPGA for Real-time Object Detection" Netaji Subhas Institute of Technology University of Delhi New Delhi, India, DOI: 10.1109/ICCCNT45670.2019.8944792.
- [15]. Ruizhe Zhao, Xinyu Niu, Yajie Wu, Wayne Luk, and Qiang Liu. "Optimizing CNN-Based Object Detection Algorithms on Embedded FPGA Platforms". In: Applied Reconfigurable Computing. Ed. by Stephan Wong, Antonio Carlos Beck, Koen Bertels, and Luigi Carro. Cham: Springer International Publishing, 2017, pp. 255–267. ISBN: 978-3-319-56258-2
- [16]. Runze Huang, Jintao Wang School of Information and Communication Engineering, Communication University of China, Beijing, China, "A design of object detection system in fog" IEEE 6th Information Technology and Mechatronics Engineering Conference (ITOEC), DOI: 10.1109/ITOEC53115.2022.9734584