

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي
École Nationale Polytechnique



المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

**End-of-study project dissertation for obtaining the State Engineer's
degree in Electronics**

STUDY AND IMPLEMENTATION ON FPGA OF HUMAN RECOGNITION SYSTEM VIA IRIS BASED ON DEEP LEARNING

**NBRI Rihame
REKROUK Maroua**

Under the supervision of Latifa HAMAMI, Professor

Presented and defended on June 26th , 2023 before the members of jury :

President	M. Hicham	Bousbia-Salah	Prof.	ENP, Alger
Supervisor	Mme. Latifa	HAMAMI	Prof.	ENP, Alger
Examiner	M. Mohamed O.	TAGHI	MAA.	ENP, Alger

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي
École Nationale Polytechnique



المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

**End-of-study project dissertation for obtaining the State Engineer's
degree in Electronics**

STUDY AND IMPLEMENTATION ON FPGA OF HUMAN RECOGNITION SYSTEM VIA IRIS BASED ON DEEP LEARNING

**NBRI Rihame
REKROUK Maroua**

Under the supervision of Latifa HAMAMI, Professor

Presented and defended on June 26th , 2023 before the members of jury :

President	M. Hicham	Bousbia-Salah	Prof.	ENP, Alger
Supervisor	Mme. Latifa	HAMAMI	Prof.	ENP, Alger
Examiner	M. Mohamed O.	TAGHI	MAA.	ENP, Alger

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي والبحث العلمي
École nationale Polytechnique



المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Electronique

Mémoire de projet de fin d'études
Pour l'obtention du diplôme d'Ingénieur d'État en Électronique

ETUDE ET IMPLEMENTATION DE LA RECONNAISSANCE DES PERSONNES PAR L'IRIS VIA DEEP LEARNING

NBRI Rihame
REKROUK Maroua

Sous la direction de Pr.Latifa HAMAMI ENP, Alger

Présenté et soutenu publiquement le 26/06/2023 auprès des membres du jury :

Président	M. Hicham Bousbia-Salah	Prof.	ENP, Alger
Promotrice	Mme. Latifa HAMAMI	Prof.	ENP, Alger
Examineur	M. Mohamed O. TAGHI	MAA.	ENP, Alger

ENP 2023

Dedication

“

This work is dedicated foremost to my Mom, the source of all things. She has shown me, through her own vocation, how to transform empathy and the inability to accept the plight of the most disadvantaged in today's society into my own calling. I am immensely grateful for her unwavering gentleness, which is unique to her alone. Thank you for your constant support and advocacy. If I have achieved anything today, it is solely because of you and only you. I also want to express my gratitude to my Father for his patience. In his own way, he imparted to me all that he knew and all what he had and more, thank you.

This dedication extends to my friends and my brother Khalil. Each of you is a radiant sunbeam that brightens my mornings and makes every day worthwhile, despite the challenges. A special thanks goes to my friend Katia and rekrouk the sisters i didnt had, they helped me in the most hard moment i ever had in my whole life. they made the three years easier and happier, thank you my friends. I would like to express my appreciation to all my friends and classmates "ELN MATCH 2023" from the past three years at ENP.

I would like to take a moment to thank my dearest friend Maroua TOUTI and Mohamed BOUCHAKOUR, without whom I wouldn't be where I am today. Thank you for your love, support, patience, and for being the inseparable duo that we are.

”

- NBRI Rihame

Dedication

“

This work is dedicated foremost to my Mom, the source of all things. She has shown me, through her own vocation, how to transform empathy and the inability to accept the plight of the most disadvantaged in today's society into my own calling. I am immensely grateful for her unwavering gentleness, which is unique to her alone. Thank you for your constant support and advocacy. If I have achieved anything today, it is solely because of you and only you.

I also want to express my gratitude to my Father for his patience. In his own way, he imparted to me all that he knew and all what he had and more, thank you.

This dedication extends my sister and my two brothers and my friends. Each of you is a radiant sunbeam that brightens my mornings and makes every day worthwhile, despite the challenges.

I would like to express my appreciation to all my friends and classmates "ELN MATCH 2023" from the past three years at Ecole Nationale Polytechnique. Special thanks to Mohamed NENNOUCHE who helped me a lot during the past four years, thank you!

”

- REKROUK Maroua

Acknowledgments

We would like to express our heartfelt appreciation and gratitude to our project supervisor, Prof. HAMAMI Latifa. We are truly thankful for her unwavering guidance, support, and constant presence throughout our project. Her abundant knowledge, passion for Image processing and her kind and gentle nature have inspired us greatly.

We are also grateful to Prof. HAMAMI for giving us the opportunity to work within the Estimation, Modeling, and Analysis Group under her supervision. It has been a remarkable experience for us, not only as a novice researcher but also as an individual. Working under her guidance has been a highlight of our academic journey. Furthermore, we extend our sincere gratitude to Mme.ALLAM , who is a researcher at Ecole Nationale Polytechnique. She was one of the first to introduce us to the field of Biomedical Engineering, she has provided invaluable assistance, knowledge, and unwavering kindness. We are truly grateful for his guidance and flexibility.

We would also like to acknowledge the President of the jury, Prof. Hicham Bousbia-Salah, who is a Professor at Ecole Nationale Polytechnique, and the Examiner, Mr. Mohamed Oussaid Taghi, who is a Teacher-Researcher at the same institution. We deeply appreciate their willingness to be part of the reading committee and for their constructive analysis of our work. Their time, teaching, and support over the past three years have been instrumental in our growth and development.

We are immensely thankful to Ecole Nationale Polytechnique for funding this project. Without their support, this contribution would not have been possible. We would also like to express our sincerest thanks to all the professors of Ecole Nationale Polytechnique for facilitating the process in a seamless manner.

Lastly, we would like to express our gratitude to all the teachers and professors who have crossed our path since elementary school and our early years. Each and every one of you has played a significant role in shaping the persons we are today, and we deeply appreciate their contribution to our education and growth.

ملخص

نظراً للسمات المميزة والثابتة الموجودة في قزحية العين البشرية، فقد أثارت أنظمة التعرف على القزحية اهتماماً كبيراً في مجال التحقق والتعرف البيومتري. يقدم هذا المشروع تحقيقاً شاملاً وتنفيذاً عملياً على متحكم FPGA لنظام التعرف على القزحية، بهدف الوصول إلى حل للمصادقة البيومترية فعال وموثوق. تقدم الدراسة المجراة في هذا المشروع مساهمات ملحوظة في مجال التعرف على القزحية. فهي تقدم نهجاً مبتكراً لتحسين أداء النظام ومواجهة التحديات الحاسمة المرتبطة بهذه التكنولوجيا. يتمتع النظام المقترح بإمكانات واعدة في تطبيقات متنوعة، بما في ذلك التحكم في الوصول والمراقبة والتحقق من الهوية. من المتوقع أن تدفع الأفكار القيمة التي تم الحصول عليها من هذا المشروع للتقدم في تكنولوجيا التعرف على القزحية في المستقبل. بالإضافة إلى ذلك، تلعب هذه الأفكار دوراً حاسماً في تطوير أنظمة المصادقة البيومترية الأكثر أماناً وموثوقية.

كلمات مفتاحية : المصادقة البيومترية, أنظمة التعرف على القزحية , FPGA

Résumé

En raison des caractéristiques distinctes et cohérentes présentes dans l'iris humain, les systèmes de reconnaissance de l'iris ont suscité une attention considérable pour l'identification biométrique et l'authentification. Ce projet présente une enquête approfondie et une mise en œuvre pratique sur FPGA (Field-Programmable Gate Array) d'un système de reconnaissance de l'iris, ayant pour objectif principal de créer une solution d'authentification biométrique à la fois efficace et fiable. L'étude menée dans ce projet apporte des contributions notables au domaine de la reconnaissance de l'iris. Elle introduit une approche novatrice qui améliore non seulement les performances du système, mais qui aborde également les défis critiques associés à cette technologie. Le système proposé présente un potentiel prometteur pour diverses applications, telles que le contrôle d'accès, la surveillance et la vérification d'identité. Les connaissances précieuses obtenues grâce à ce projet devraient stimuler les avancées futures dans la technologie de reconnaissance de l'iris. De plus, elles jouent un rôle crucial dans le développement continu de systèmes d'authentification biométrique plus sécurisés et fiables.

Mots clés : FPGA, Les systèmes de reconnaissance de l'iris, L'authentification bio-métriques.

Abstract

Due to the distinct and consistent traits found in the human iris, iris recognition systems have garnered significant attention for biometric identification and authentication purposes. This project presents an extensive investigation and practical implementation on FPGA of an iris recognition system, with the primary objective of creating a biometric authentication solution that is both efficient and dependable. The study conducted in this project makes notable contributions to the field of iris recognition. It introduces an innovative approach that not only improves system performance but also tackles critical challenges associated with the technology. The proposed system holds promising potential for diverse applications, including access control, surveillance, and identity verification. The valuable insights obtained from this project are expected to drive future advancements in iris recognition technology. Additionally, they play a crucial role in the ongoing development of more secure and trustworthy biometric authentication systems.

Keywords : FPGA, Iris recognition systems, Biometric authentication.

Contents

List of Tables

List of Figures

List of Equations

Abrviations

General Introduction	18
1 State of The Art - Literature Review	20
1.1 Introduction	21
1.2 Iris Recognition System - Related Work	21
1.2.1 Image Preprocessing	21
1.2.2 Edge Detection	22
1.2.3 Segmentation	22
1.2.4 Normalization	22
1.2.5 Features Extraction	23
1.2.6 Classification	23
1.2.7 Performances	24
1.3 Implementation - Related Work	24
1.3.1 CNN Optimization Techniques	25
1.3.2 CNN Accelerator Architectures	25
1.3.3 Implementation Methods	25
1.3.4 Performances	27
1.4 Proposed Method	28
1.4.1 Proposed Model	28
1.4.2 Implementation	30
1.5 Conclusion	31

2	Terminology	32
2.1	Introduction	33
2.2	Preprocessing and Segmentation	33
2.2.1	Filtering	33
2.2.2	Histogram Equalization	38
2.2.3	Contrast	39
2.2.4	Gamma Correction (Power Law transformation)	39
2.2.5	Thresholding	40
2.2.6	Hough Transform	40
2.3	Features Extraction	41
2.3.1	Gabor Filter	41
2.3.2	Wavelet Transform	41
2.3.3	Local Binary Patterns	42
2.3.4	Gray Level Co-occurrence Matrix	43
2.4	Convolutional Neural Network	45
2.4.1	Convolution Filters	45
2.4.2	Pooling	46
2.4.3	Flatten Layer	47
2.4.4	Stride and Padding	47
2.4.5	Dropout	48
2.4.6	Adam Optimizer	48
2.4.7	Sparse Categorical Cross Entropy Loss	48
2.5	Machine Learning Methodes	49
2.5.1	Support Vector Machines	49
2.5.2	Random Forest Classifier	50
2.5.3	Accuracy	50
2.5.4	Epochs	51
2.6	Implementation	51
2.6.1	IP Block	51
2.6.2	Function Call Graph	51
2.6.3	AXI Protocol	52
2.6.4	Netlist	52
2.7	Conclusion	53
3	Proposed Approach and Results	54

Contents

3.1	Introduction	55
3.2	Iris Detection	55
3.2.1	Preprocessing - Reflections Reduction	55
3.2.2	Edge Detection	56
3.2.3	Iris Isolation	57
3.2.4	Normalization	59
3.2.5	Preprocessing - Enhancement	60
3.2.6	Iris Detection Evaluation	61
3.2.7	Dataset Preparation	61
3.2.8	Data Augmentation	62
3.3	Features Extraction	63
3.3.1	Deep Learning based Methods	63
3.3.2	Hybrid Method (Features Fusion)	68
3.3.3	Results of Feature Extraction methods	74
3.4	Classification	75
3.4.1	Deep Neural Network Classifier	75
3.4.2	Machine Learning Based Classifier	76
3.4.3	Support Vector Machine	76
3.4.4	Random Forest Classifier	77
3.5	Proposed Model	78
3.5.1	Proposed Architecture	78
3.5.2	Training of the Architecture	79
3.6	Results of the proposed architecture	80
3.7	Conclusion	81
4	Implementation	83
4.1	Introduction	84
4.2	Optimization of the model : Pruning and Quantization	84
4.2.1	Pruning	85
4.2.2	Quantization	86
4.2.3	Model Optimization Results	86
4.3	High Level Synthesis Approach	87
4.3.1	Subsystem in C language	88
4.3.2	Vitis HLS	89
4.3.3	IP Block Interfacing - Data Transfer	93

4.3.4	Block Design of the Architecture	97
4.3.5	CPU Programming	105
4.3.6	Programming Approach	106
4.3.7	Results	110
4.4	Matlab appraoch	111
4.4.1	Quantisation for FPGA Target	112
4.4.2	Prototype Deep Learning Networks on FPGA and SoC Devices	115
4.4.3	Results and evaluation	123
4.5	Results Comparison	123
4.6	Conclusion	125
	Conclusion and Perspectives	126
	Bibliography	129
	Webography	131

List of Tables

- 1.1 Summary of the different models and their performances 24
- 1.2 Summary of previous implementation results 27

- 3.1 Our datasets before and after proceeding with the substage described earlier. . . 61
- 3.2 Our datasets before and after proceeding data preparation 62
- 3.3 Our datasets after proceeding data augmentation 63
- 3.4 Results of Features Extraction Methods 74
- 3.5 Comparaison of Deep Learning based Methods 76
- 3.6 Results of the different tests using the proposed architecture 82

- 4.1 Results of Pruning and Quantization of the model 86
- 4.2 AXI protocol according to the variables types 96
- 4.3 Resources Consumption of the Features Extraction sub-stage on PYNQ Z1 . . 111
- 4.4 Resources utilization comparision between the two appraochs 124

List of Figures

1.1	Proposed Method	28
2.1	Top Hat Filter	35
2.2	Sobel Filter Kernels	36
2.3	Non-maximum Suppression	37
2.4	Dual-Threshold Segmentation	37
2.5	Edge Stitching with Hystheresis Thresholding	38
2.6	Histogram Equalization example	39
2.7	Curves of Gamma Correction	40
2.8	Examples of different types of wavelets	42
2.9	LBP Computation	43
2.10	GLCM Matrix Calculation	44
2.11	Convolution Operation	46
2.12	Pooling Operation	46
2.13	Pooling Operation	47
2.14	Padding effects on Output Image Size	48
2.15	SVM example	49
2.16	Random Forest Classifier	50
2.17	Call Graph Example	52
3.1	Comparison between an original image and image got after reflections reduction	55
3.2	Iris boundaries detection	56
3.3	Outer and Inner Boundaries Edge detection	57
3.4	CHT Output Example	58
3.5	Final Mask	58
3.6	Segmented iris of person 224 of IITD dataset	59
3.7	Normalized iris of person 224 of the IITD dataset	60
3.8	Comparison of before and after enhancement	60

3.9	Iris Detection Process	61
3.10	Different Data Augmentation Techniques	62
3.11	IRISNet Architecture	64
3.12	Residual learning : a building block	64
3.13	Details About ResNet Layers	65
3.14	AlexNet Architecture	65
3.15	Example of Siamese Architecture	66
3.16	CapsNet Architecture	67
3.17	Thresholding the center pixel neighbors	69
3.18	Collecting the threshold values	69
3.19	Replace the center pixel's value by the threshold values from the neighborhood	69
3.20	Example of LBP on Pheonix's image	70
3.21	GLCM's four angles	70
3.22	Wavelets filters	71
3.23	Haar, Biorthogonal, Dubechee and Discrete Mayor wavelet app on iris image .	72
3.24	Example of WT on Pheonix's image	72
3.25	Features of GF applied on IITD's image	73
3.26	Experimented Features Extraction Methods	74
3.27	Classifier - Explanatory Schema	75
3.28	Random Forest Algorithm	78
3.29	Proposed Architecture	79
4.1	Visualization of pruning weights/synapses vs nodes/neurons	85
4.2	Simplified Synoptic Schema of the HLS Approach	87
4.3	C code of CONV3	89
4.4	Vitis HLS Interface	90
4.5	Padding of an image having 5-by-5 dimension (pad=1)	90
4.6	Synthesis summary of the CONV3 layer	91
4.7	Call graph of the CONV3	92
4.8	Summary of RTL export and implementation	93
4.9	Interfacing Directives for CONV3	93
4.10	Channel Connections between Master and Slave Interfaces	94
4.11	Valid Transfer of Data	95
4.12	Data Flow of the Features Extraction Sub-stage	97
4.13	Vivado Interface	98

4.14 IP Repositories	98
4.15 Addition of the different IP Cores to the new Block Design	99
4.16 CONV3 IP Block	99
4.17 Block Design of the Features Extraction sub-stage	100
4.18 Address Editor in Vivado Design Suit	101
4.19 Utilization Report	102
4.20 Power Report	103
4.21 Resources Usage Report	104
4.22 Summary of Power Report	104
4.23 Architecture of the PYNQ-Z1 Board	105
4.24 Description of The Overlay created and the IP Blocks of this last	106
4.25 Details of the CONV1 Block	107
4.26 Mapping of CONV1 Block Using MMIO Class	107
4.27 Register Map of CONV1	107
4.28 Memory Allocation for all the IP blocks	108
4.29 Strating the computation's code	109
4.30 The Behavior of the Block-Level Handshake Signals	109
4.31 Output Buffer	110
4.32 Matlab solution for Deep learning on FPGA	111
4.33 Model saved	113
4.34 Architecture of our model imported to Matlab	113
4.35 Quantifiability of each layer	114
4.36 Dynamic Range of Calibration Layers	115
4.37 The process of deploying a network to a custom board	116
4.38 board registration function	117
4.39 Reference Design Registration Function	118
4.40 Calibration bitstream	118
4.41 Internal memory transaction latencies	119
4.42 Resource Estimation of original model	119
4.43 Resource Estimation of quantified model	119
4.44 Custom Bitstream for Custom Processor Configuration	120
4.45 Custom Bitstream for Custom Processor Configuration	121
4.46 Custom Bitstream for Custom Processor Configuration	122
4.47 Final result of the Matlab approach	123

List of Equations

2.1 Gaussian Filter formula	33
2.2 Median Filter equation	34
2.3 Erosion formula	34
2.4 Dilation formula	34
2.5 Top Filter formula	35
2.6 Bottom Hat Filter formula	35
2.7 Edge gradient	36
2.8 Direction for each pixel	36
2.9 Histogram Equalization formula	38
2.10 Gamma correction formula	39
2.11 Gabor Filter formula	41
2.12 Wavelet Transform Equations	41
2.13 Assignment of the pixels' new value	42
2.14 GLCM : Contrast formula	44
2.15 GLCM : Entropy formula	44
2.16 GLCM : Energy formula	44
2.17 GLCM : Homogeneity formula	44
2.18 ReLU function formula	45
2.19 Sparse Categorical Cross Entropy formula	48
2.20 Size of Pooling Layer Output formula	49
2.21 Accuracy	50
3.1 Normalization formula	59
3.2 Support Vector Machines	76
3.3 Number of classifiers of an SVM	77

List of Abreviation

AI *Artificial Intelligence.*

DL *Deep Learning.*

CNN *Convolutional Neural Network.*

DNN *Deep Neural Network.*

FCN *Fully Connected Layer.*

SVM *Support Vector Machine.*

ReLU *Rectified Linear Unit.*

LBP *Local Binary Patterns.*

GLCM *Gray Level Co-Occurrence Matrix.*

DWT *Discrete Wavelet Transform.*

FPGA *Feild Programmable Gate Array.*

PYNQ *Python Productivity for ZYNQ.*

WT *Wavelet Transform.*

GF *Gabor Filter.*

HD *Hammin Distance.*

ED *Euclidean Distance.*

VHDL *Very High Speed Integrated Circuit Fast Hardware Discription Language.*

HDL *Hardware Description Language.*

IP *Intellectual Property.*

HLS *High Level Synthesis.*

LUT *Look Up Table.*

BRAM *Block Read Access Memory.*

SCCE *Sparce Categorical Cross Entropy.*

2D *Two Dementions.*

3D *Three Dementions.*

AMBA *Advanced Microcontroller Bus Architecture*

AXI *Advanced extensible Interface.*

HE *Histogram Equalization.*

CHT *Circlar Hough Transform.*

General Introduction

Iris foot print refers to the distinctive patterns and characteristics found in an individual's eye iris. These patterns are utilized by iris recognition systems for biometric identification and authentication purposes. However, despite the widespread adoption and benefits of iris recognition technology, there are challenges and issues associated with utilizing them.

The algorithms used in iris recognition involve intricate image processing and pattern matching techniques. These algorithms can be computationally demanding, requiring significant processing and memory resources. Efficiently implementing these algorithms to achieve real-time performance is a challenging task. Optimization techniques like hardware acceleration, or algorithm simplification may be necessary to meet the computational requirements of iris foot print recognition. Developing algorithms that are both efficient and robust, capable of handling variations in iris patterns, occlusions, and noise, poses an additional challenge. Optimizing these algorithms for accuracy, speed, and scalability is crucial for achieving real-time performance and effective identification.

In summary, while iris recognition technology offers unique advantages, challenges remain in implementing and utilizing iris foot prints. These challenges include addressing the computational complexity of the algorithms, optimizing them for accuracy and scalability. Overcoming these challenges requires expertise in areas such as image processing, deep learning, algorithm optimization, and system design.

Motivation and objectives

The motivation behind conducting a study and implementing a human recognition system via iris based on deep learning on an FPGA stems from the increasing demand for accurate and reliable biometric identification solutions. Iris recognition has emerged as a highly secure and robust biometric modality, offering unique advantages such as uniqueness, stability, and resistance to tampering. Deep learning techniques have shown remarkable success in various computer vision tasks, including iris recognition. By implementing this system on an FPGA, we aim to leverage the parallel processing capabilities and hardware acceleration to achieve real-time performance and efficient deployment in resource-constrained environments.

The objectives of this project

- **Algorithm Development** : Design, test and develop efficient and accurate Image Processing and Deep Learning algorithms for iris recognition. Explore different architectures and techniques, such as segmentation, convolutional neural networks (CNNs) to extract features from eye images and identify the person.
- **Hardware-Software Co-design** : Optimize the developed Deep Learning algorithms for Field Programmable Gate Array (FPGA) implementation. Explore hardware acceleration techniques and algorithmic optimizations to minimise the utilization of FPGA resources.
- **Deployment and Validation** : Validate the implemented system using a large-scale iris database and real-world scenarios. Assess its effectiveness in accurately recognizing and identifying individuals.

Work structure

The problematic is tackled following the steps :

- literature Overview :
 - Conduct an in-depth review of the existing literature on iris recognition, deep learning techniques, and FPGA implementation.
 - Identify key concepts, algorithms, and methodologies relevant to the study.
 - Analyze the strengths and limitations of previous works to establish a foundation for the project.
- Preprocessing and Segmentation :
 - Preprocess the dataset to remove noise, normalize the images, eliminate reflections and handle any artifacts or variations in image quality.
 - Edge detection and segmentation of the pre-processed dataset.
- Features extraction and Classification :
 - Study the suitable Deep Learning and statistical methods for textural features extraction such as Local Binary Patterns (LBP).
 - Study the suitable deep learning architectures for texture recognition, such as Convolutional Neural Networks (CNNs).
 - Implement and train the deep learning models using the acquired dataset, test the different architectures.
 - Design a better and a new architecture.
- System Requirements and Design :
 - Define the specific requirements and objectives of the project, considering factors such as recognition accuracy, real-time performance, power efficiency, and resource utilization.
 - Design the overall architecture of the system, including the hardware components (FPGA, interface, memory, etc.).
 - Determine the interfaces and protocols for seamless integration with other system components.

Chapter 1

State of The Art - Literature Review

1.1 Introduction

This chapter focuses on three main aspects : the typical structure of an iris recognition system, a review of significant contributions in each stage of the system and a summary of the proposed method.

Firstly, we will discuss the commonly employed structure of an iris recognition system, highlighting its key components and their respective functions.

Secondly, we will provide an overview of noteworthy research conducted in each stage of the system. This includes advancements made in preprocessing techniques to enhance iris images, features extraction algorithms to capture distinctive iris features, template generation methods for creating compact representations of these features, matching algorithms to compare iris templates, and decision-making strategies based on similarity scores.

Lastly, we will summarize the proposed method put forward in this study. This summary will encompass any novel approaches or enhancements introduced in comparison to existing techniques, as well as a brief explanation of the evaluation methodology and performance metrics used to assess the proposed method.

By exploring these aspects, this chapter aims to provide a comprehensive understanding of the structure and advancements in iris recognition systems, while also presenting a concise summary of the proposed method.

1.2 Iris Recognition System - Related Work

Similar to other recognition systems, the iris recognition system comprises two main stages : iris detection and iris identification (matching). The initial stage includes preprocessing, edge detection, segmentation and normalization. The subsequent stage involves features extraction and classification. Extensive research and advancements have been carried out in each sub-stage of the system and a collection of these studies is outlined below.

1.2.1 Image Preprocessing

As mentioned earlier, the first sub-stage of detection is preprocessing, which involves various operations aimed at eliminating noise, improving contrast and serving other segmentation purposes. This phase enhances the image quality, making it more informative and better prepared for the features extraction phase.

In this stage, notable works have been conducted. For instance, *Kumar and al.* [1] focused on image enhancement techniques to improve system accuracy. They proposed a novel approach that combines both Top and Bottom Hat filters for image enhancement.

Another interesting work by *Chang and al.* [2] aimed to isolate the pupil and locate the inner boundaries of the iris. They proposed a preprocessing method consisting of three stages : Gaussian filtering with a sigma value of 0.9, binary thresholding with a threshold of 0.18 and a subsequent Gaussian filter with a sigma value of 2. This method effectively detected the pupil

and isolated it from the rest of the eye.

In the work of Ming Liu [3], a Fuzzy method was employed to enhance images. Unlike other approaches, this method focused on addressing the image non-linearities that are commonly ignored by commonly used filters such as Median, Gaussian, and others. These filters typically assume the image characteristics to be Gaussian, which is often not the case.

1.2.2 Edge Detection

Following the preprocessing phase, which ensures the quality of the images, the subsequent stage of edge detection becomes crucial in highlighting the boundaries of the iris. Several researchers, including Tahir [4], Omran [5] and Samitha [6], have utilized the Canny edge detector for this purpose, given its efficiency and effectiveness. The Canny edge detector has been widely adopted by these researchers in iris recognition systems to detect and enhance the edges of the iris region.

1.2.3 Segmentation

Once the edges of the image are detected, the next stage is segmentation, where the localization of iris boundaries takes place, retaining only the iris region (the region of interest) by eliminating other areas. Various approaches have been employed for this purpose, as mentioned in the review [7], including K-Means Clustering, Active Contour Models, Edgeless Active Contour, Gradient Vector Flow Snake, Statistical Learning Methods, Least Median of Square Differential Operator, and Linear Basis Function.

In particular, Yinyin Wei [8] and Guo Qiaoli [9] utilized the Integro-Differential Operator to detect the inner boundaries of the iris, while the Hough Transform was employed to detect the outer boundaries. These methods have demonstrated effective segmentation results.

However, the most commonly used method is the Circular Hough Transform, which has been extensively employed by numerous researchers in various works, including [10], [11], [12], [13]. This method assumes that the inner and outer boundaries of the iris are circular, enabling accurate detection and segmentation.

1.2.4 Normalization

Normalization plays a crucial role in iris recognition as it transforms the circular iris region into a rectangular shape, making it suitable for further processing. Among the most prominent and widely used methods for iris normalization, Daugman's Rubber Sheet model stands out. Despite being an older technique, it remains highly efficient and continues to be employed in current research.

Numerous recent articles, including [1], [2], [3], [4], rely on Daugman's Rubber Sheet model for iris normalization. This model offers a reliable and effective approach to transform the iris region into a rectangular representation, ensuring compatibility with subsequent feature extraction and matching algorithms. Despite the emergence of newer techniques, Daugman's

Rubber Sheet model is still favored due to its robustness and well-established performance in iris recognition systems.

1.2.5 Features Extraction

After normalizing the image, we proceed to the next step, which involves extracting the most important features from the resulting rectangle obtained during the normalization sub-stage. It is important to note that both features extraction and features selection are dimension reduction methods employed for this purpose.

Features extraction aims to identify and extract the most relevant features from the input data while reducing its dimensionality. On the other hand, features selection involves choosing a subset of the original features to decrease dimensionality. Both techniques aim to enhance the performance of the Machine Learning model by reducing the complexity of the data.

In this discussion, we will primarily focus on Features Extraction, which takes a mixture of independent components as input data. Its objective is to accurately identify each component by eliminating any unnecessary noise, thus generating a reduced dataset that is more manageable for processing and manipulation without sacrificing accuracy.

To achieve this, various mathematical algorithms have been implemented for feature extraction, including Principal Component Analysis based on Discrete Wavelet, Haar and Biorthogonal Wavelet Transform, Gabor and Log Gabor filters, Local Binary Patterns, Discrete Cosine Transform, Gray Level Co-occurrence Matrix [10], Residual Pooling layer introduced in the Convolutional Neural Networks [14], Joint Bayesian Formulation and Supervised Discrete Hashing [15].

1.2.6 Classification

Once the most significant features are extracted, we proceed to the final stage of our system, which is the Classification. In iris classification, a variety of approaches are utilized, including classical methods and Machine/Deep Learning-based techniques.

The Hamming Distance, a classical method widely employed in several articles such as [13], [16] and [17], is frequently used for classification.

Convolutional Neural Networks, as seen in [3], [5], [8], [15] and [18], K-Fold Cross-Validation, K-Nearest Neighbor [18] and Support Vector Machine, utilized in [8], [9], [10] and [14], are examples of newer Machine/Deep Learning-based approaches for classification.

In addition, a hybrid approach is mentioned in [19], where both the Hamming Distance and a Neural Network are employed for the matching process

1.2.7 Performances

We present here a recapitulating table of the performances of the methods seen in the state of art in Table 1.1 :

Article	Dataset	Seg. Method	Feat.Ext Method	Classif Method	Accuracy %
[1]	IITD	/	DWT-DCT	ED	94.59
[3]	Casia-Iris-Thousand	HT	CNN - CapsNet		83,10
	IITD				86,80
	ATVS-Flr				88,4
[4]	Casia-V1	/	HD		96.48
	Casia-V4-Lamp				95.10
	SDUMLA-HMT				93.60
[5]	IITD	HT	IRISNet		96.43
[8]	Casia-V4-Distance	HT	ResNet		10.83
	ND-LG4000				61.41
	Casia-V4-Lamp				49.78
	Casia-M1-S2				36.65
[9]	Casia-V4-Thousand	HT	WT	HD	97.80
[18]	IITD	HT	DWT	SVM	98.92
[19]	Casia-V4	HT	DWT	SVM	95.40
[12]	JluIrisV3.1	/	DNN based on CapsNet		99.37
	JluIrisV4				98.88
	Casia-V4-Lamp				92.27
[13]	Casia-V4-Thousand	HT	DenseNet-201		97.30
[16]	Casia-V4-Interval	HT	Box Counting	KNN	92.63

Table 1.1 : Summary of the different models and their performances

1.3 Implementation - Related Work

Convolutional Neural Networks (CNNs) have garnered significant attention among researchers due to their impressive accuracy in various cognitive tasks. However, CNNs typically require substantial computational resources, prompting the need for improved performance through custom hardware accelerators. In this context, FPGA (Field-Programmable Gate Array) platforms have emerged as a promising solution, offering advantages such as high energy efficiency, powerful computing capabilities, and reconfigurability. These features make FPGAs well-suited for accelerating CNNs, driving increased interest in their utilization for efficient CNN processing.

Numerous studies and advancements have been conducted in each specific stage of the system, we will now highlight a few notable examples.

1.3.1 CNN Optimization Techniques

CNNs contain significant redundancy and are used for error-tolerant applications [20]. These facts allow significant simplification of CNN model which reduces its hardware implementation overhead.

Several techniques for reducing hardware overhead of CNNs were discussed in [21], such as Pruning, Quantization, Data Compression, Encoding and Binarized CNNs.

For instance, *Page et al.* [22] present five CNN sparsification techniques for boosting CNN accelerator efficiency. First, they divide convolutional layers with large filters into multiple layers with smaller filters, which effectively maintains the same receptive field region. For instance, a layer with 5x9x5 filters can be replaced with two successive layers, each with 3x9x3 layers. This reduces the number of parameters from 25 to 18, without much loss in accuracy.

Moreover, *Qiu et al.* [23] present a dynamic quantization approach and a data organization schema for improving CNN efficiency. To boost Fully Connected layers, they use singular value decomposition which reduces the number of weights. They use dynamic quantization schema where fractional length is different for different layers and features maps (fmaps) but fixed in a single layer to avoid rounding-off errors.

1.3.2 CNN Accelerator Architectures

Several CNN Accelerator architectures were used in the literature of implementation. Firstly, *Alwani and al.* [24] note that processing the neighboring CNN layers one after another leads to high amount of off-chip transfers. They propose changing the computation pattern such that multiple convolutional layers are computed together, which provides better opportunity of caching the intermediate feature maps and avoids their off-chip transfer.

In the other hand, in the study conducted by *Moss and al.* [25], they aimed to speed up a Binary Neural Network on a platform that combines a CPU and an FPGA. They used a design called a systolic array, which consists of multiple Processing Elements that perform multiplication operations using XNOR (Exclusive NOR) and bit-count operations.

1.3.3 Implementation Methods

The implementation of a Convolutional Neural Network model on an FPGA is indeed a popular approach for accelerating deep learning inference tasks so we present in this section the commonly used methods to.

A. Using High-Level Synthesis Method

One of the initial approaches we encountered involves the conversion of the model into a High-Level Language such as C/C++ before implementing it in an FPGA.

Bing Liu and al. employed the HLS (High-Level Synthesis) technique as described in their work [26]. The researchers specifically focused on the utilization of a Hybrid CNN-

SVM architecture on the Zynq-7020 accelerator. Additionally, they introduced a universal deployment methodology that automates the selection of accelerator design parameters based on the target platform and algorithm requirements.

The HLS technique is a process that transforms High-Level Code into Hardware Description Language (HDL) code, which can be used to program the FPGA. The procedure involved in this approach is as follows :

- Initially, the functions to be deployed on the FPGA are programmed using the C or C++ programming languages.
- Subsequently, these functions are synthesized into VHDL (or Verilog) to generate an IP Block specifically tailored for those functions.
- Finally, the entire system is designed by creating a block design that incorporates the previously generated IP blocks of the required functions. This block design is then synthesized and implemented in the FPGA to complete the process.

B. Using Matlab

One other interesting recent approach is Matlab. Using the Deep Learning Toolbox [27] and Deep Learning HDL Toolbox [28] which are part of the MATLAB environment, can aid in the design and deployment of CNN models on FPGAs.

Here's a general overview of the process involved in implementing a CNN model on an FPGA based on their official documentation [29] :

1. CNN Model Design or importing from python,
2. Fixed-Point Conversion[30],
3. Code Generation [31],
4. IP Core Generation [32],
5. Bitstream file Generation [33],
6. FPGA Deployment and Inference [34].

C. Using HDL from scratch

Another common approach is using HDL directly for coding Pre-trained Deep Learning applications (training off-chip). *Hossam and al.* [35] present a design methodology related to the implementation of a Multiply-Accumulate (MAC) unit using VHDL. This MAC unit is specifically designed to support Deep Learning Networks on FPGA platforms. The authors may discuss the concurrent nature of the MAC unit, highlighting its ability to perform multiple multiply-accumulate operations simultaneously.

D. Using Vitis AI

Another new approach we came across is to use Vitis AI where *Wang and al.*[36] discussed the implementation of an Object Detection Accelerator on an FPGA using the Vitis-AI development

framework. The authors may present their methodology, design, and optimization techniques employed to accelerate object detection algorithms on FPGA.

The focus of the paper is on leveraging the Vitis-AI framework, which provides a High-Level development environment for AI applications targeting FPGAs. The authors discuss the advantages and challenges of using this framework for implementing Object Detection algorithms and provide insights into the performance, efficiency, and accuracy of their FPGA-based accelerator.

1.3.4 Performances

We present here a recapitulating table of the performance of the methods seen in the State of The Art in Table 1.2 :

Article	Arch.	FPGA	Frequency (MHz)	Optimization techniques	Implementation techniques	Data precision	Metric	Results
[21]	/	/	/	Pruning - Quantization - Data compression - Encoding - Binarized CNNs	HLS	/	/	/
[37]	VGG16	Altera Stratix V	200	Winograd convolution engine	/	32 bits fixed	Overall Latency (ms)	142.30
[38]	Alexnet	Virtex7 VX485T	100	Loop Pipelining - Binarized CNNs	HLS	32bits float	LUT	186251
[39]	VGG	Arria-10 GX 1150	150	Loop Pipelining	/	/	Overall Latency (ms)	47.97
[39]	/	Stratix-V GXA7	100	/	HLS	8-16 bits fixed	Logic Utilization	112K

Table 1.2 : Summary of previous implementation results

1.4 Proposed Method

After conducting extensive research and thorough testing of various methods, we have made a deliberate decision to incorporate the following techniques in each sub-stage of the iris recognition system. Moreover, we have introduced our own contributions that aim to improve the overall performance of the system presented in the Figure 1.1 :

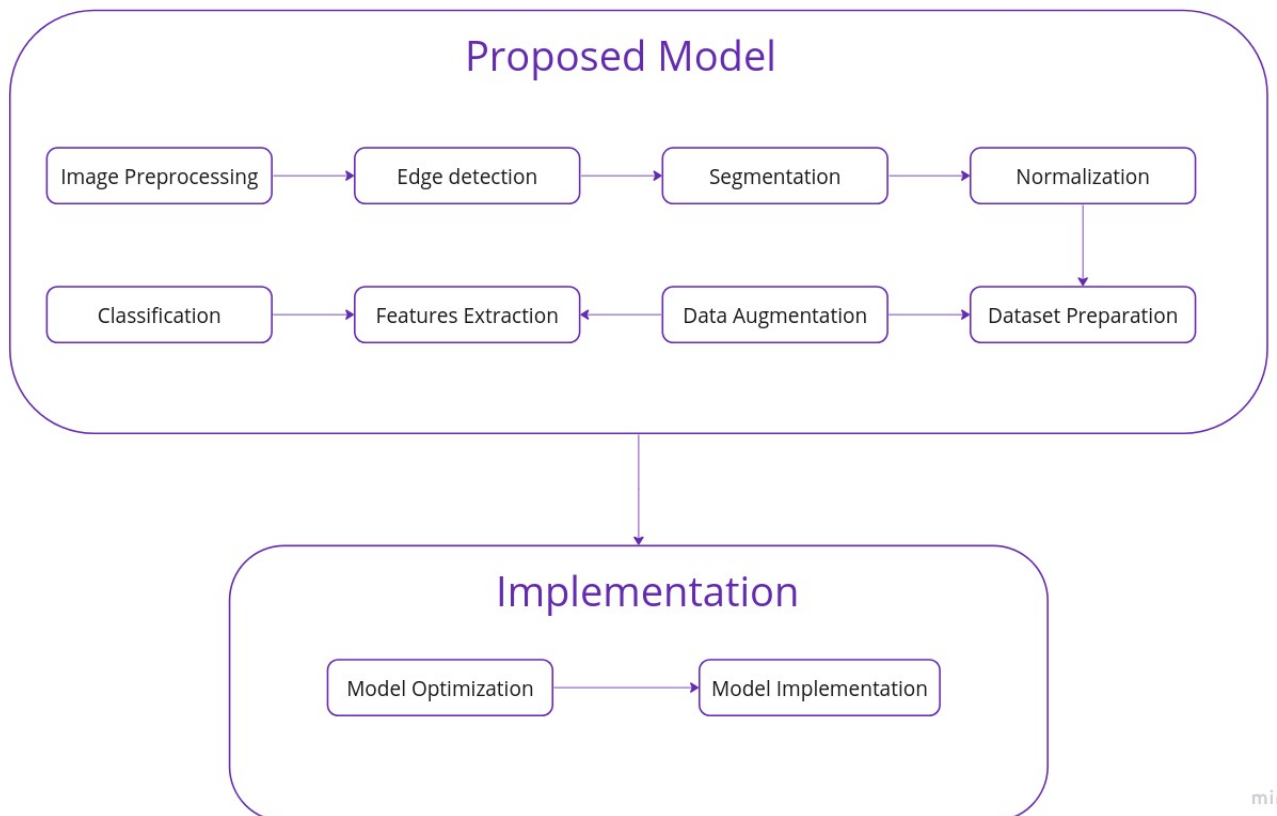


Figure 1.1 : Proposed Method

1.4.1 Proposed Model

A. Image Preprocessing

As mentioned earlier, the primary objective of this phase is to enhance the quality of the image. This can be accomplished through the following methods :

- Enhancement of the image : in order to improve the contrast, we considered a modified form of the method used in article [1] ;
- Denoising the image : in order to reduce reflections and the usual types of noise, namely the Gaussian and the impulsive. For reflections reduction, we proposed a new approach that will be discussed in the next chapter.

B. Edge Detection

So far, the Canny edge detector has proven to be one of the most effective edge detection algorithms, as it allows a precise edge detection without thickening the edges or excessively blurring the image, thus preserving important information. This quality has greatly influenced our decision to employ the Canny edge detector in this particular sub-stage. It is important to note that the Canny edge detector requires binary images as input. Therefore, as part of our process, we apply a thresholding technique prior to applying the Canny edge detector. We will provide a detailed explanation of this process in the subsequent chapter.

In addition to the Canny edge detector, we have also considered the Sobel filter as a reliable alternative. The Sobel filter serves as a backup option in cases where the threshold used for detecting the outer boundary is not appropriately adapted to the intensity level of the iris in a given image. By incorporating the Sobel filter, we can ensure robust edge detection even in challenging scenarios.

By incorporating these filters into our system, we aim to achieve precise edge detection while maintaining the integrity of the image information.

C. Segmentation

In this phase, once the crucial edges of the image representing the iris and pupil boundaries have been detected, we proceed to isolate the iris region. To accomplish this, we utilize the Hough Transform, which is the most commonly used method for iris segmentation. This choice is motivated by the Hough Transform's favorable characteristics, such as its low computational cost and its effectiveness in accurately segmenting the iris region.

By employing the Hough Transform in our segmentation process, we aim to precisely extract the iris region from the input image to facilitate subsequent analysis and the features extraction.

D. Normalization

Due to its efficiency and computational simplicity, we have opted to utilize the Rubber Sheet method for normalizing the iris zone. The Rubber Sheet method is a well-established technique that transforms the circular iris region into a rectangular shape, making it easier to process and analyze.

E. Dataset Used

To ensure comparability with results from other articles, we decided to utilize the widely used iris datasets for evaluating our findings. The datasets used are :

- CASIA-Iris V-4 that includes 8 sub datasets [40].
- IIT Delhi Iris Database [41].
- Multimedia University (MMU) Database [42].
- Phoenix Database [43].

We preprocessed all the images of each dataset and finally stored them with their respective label (identifier of the person) in order to feed them to the classifier.

F. Dataset Preparation and Data Augmentation

After preparing the segmented images, it is crucial to ensure that we only include the most representative ones, thus, we eliminate those that lack sufficient information.

After completing the cleansing process, it is important to ensure that our model is fed with a sufficient number of images. To achieve this, we employ Data Augmentation techniques which help us increase the number of images per class.

H. Features Extraction

As mentioned before, the features extraction is as important as the classification, and based on the most effective algorithms and methods that we tested on the normalized images: Convolutional Layers of different CNN architectures, Local Binary Patterns, Gabor filter, GLCM, and Biorthogonal Wavelet; we considered using Convolutional Layers because of their effectiveness with the textural images.

I. Classification

After testing different classifiers, we considered using a Random Forest Classifier. Our choice was motivated by the effectiveness of this Machine Learning method besides the reduced number of computations required when using it.

1.4.2 Implementation

- **Model Optimization**

After selecting the model, our focus shifted to the implementation stage. We considered employing in it the following optimization techniques:

- **Pruning**

- To eliminate less prominent connections in our model, we implemented a Low Magnitude Pruning technique. This involved removing connections with the lowest weights in each layer, which not only helped reduce their impact on the overall model, but also minimized the computational resources required for implementing our model.

- **Quantization**

- To minimize resource usage, our approach involves implementing quantization from 32-float to 16-float and 8-float. This technique was evaluated using both Python (32-f to 16-f) and Matlab (32-f to 16-f and 8-f).

- **Model Implementation**

We extensively studied various methods and approaches feasible for implementing Deep

Learning models during the last step of our solution. After careful consideration, we chose to focus on two specific approaches :

- Using High-Level Synthesis,
- Using Matlab Deep Learning toolbox and Deep Learning HDL Toolbox.

We made this decision based on the availability and effectiveness of these two approaches.

1.5 Conclusion

In conclusion, this chapter has provided a comprehensive overview of the current State Of The Art of iris recognition systems. We have presented the overall structure of the system and highlighted significant advancements made in each sub-stage. Furthermore, we have discussed the different approaches utilized for implementing the final system on FPGA.

Moreover, we have introduced the specific method adopted in our project, emphasizing our novel contributions. One noteworthy contribution is the development of a new approach for reflection reduction. Our approach builds upon the works cited as [1], [5], [10] in the Iris Detection stage, which includes Image Pre-processing, Edge Detection, Segmentation, and Normalization sub-stages. In the Iris Matching stage, we have drawn inspiration from [5] and [8]. These references have played a crucial role in shaping our methodology and have contributed significantly to the advancements achieved in our research.

Chapter 2

Terminology

2.1 Introduction

Before proceeding with the detailed presentation of our proposed approach, it is crucial to have a solid understanding of key concepts in Image Processing, Machine/Deep Learning and FPGA implementation.

Therefore, in this chapter, we will provide an overview of the essential terminology associated with Iris Recognition Systems. This will not only enhance comprehension of our approach but also facilitate a better understanding of the subsequent discussions.

2.2 Preprocessing and Segmentation

2.2.1 Filtering

Filtering is passing or rejecting a certain frequency component. Spatial filtering consists of applying a predefined operation to a neighborhood in order to create a new pixel having the coordinates of the center of that neighborhood and its value is the result of the filtering operation. When the filter (also called a mask or kernel) passes through all the pixels of the image, a new image (the filtered image) is generated [44].

In the frequency domain as well as the spatial domain, noise is considered as high frequencies, and rejecting this band is a key to reducing it. For this purpose, multiple filters are introduced, such as the Gaussian filter and Median filter that will be defined next.

A. Gaussian Filter

It is a 2D convolutional operator used to remove Gaussian noise and it's more effective at smoothing images. The degree of smoothing is determined by the standard deviation of the Gaussian distribution.

The Gaussian distribution in 2D has formula 2.1 [44] :

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp -\frac{x^2 + y^2}{2\sigma^2} \quad (2.1)$$

Where :

σ : the standard deviation of the distribution.

(x,y) : the coordinates of the pixel in question.

We have also assumed that the distribution has a mean of zero.

B. Median Filter

The Median filter is one of the order-statistic filters. It attributes to the new pixel the median of the intensity levels in the neighborhood, thus it has Equation 2.2 [44] :

$$f(x, y) = \text{median}I(s, t) : (s, t) \in S_{xy} \quad (2.2)$$

Where :

(x,y) : the coordinates of the pixel in question,

S_{xy} : the neighborhood of the center pixel,

(s,t) : the coordinates of the pixel in the neighborhood S_{xy} ,

I : the intensity level of the pixel having (s,t) coordinates.

Compared to the Gaussian filter and other known filters with similar kernel size, the Median filter effectively reduces both bipolar and uni-polar impulse noise with considerably less blurring the image.

C. Top and Bottom Hat Filters

Before defining this filters, we need to introduce two important morphological operations : Erosion and Dilation.

It is crucial to know that morphological operations are a set of mathematical tools used to extract the pertinent components of an image to represent and describe a region's shape. They are based on a structuring element (kernel) and two ideas, fit and hit [44] :

Fit : when all on pixels in the structuring element put at a determinate position, cover all on pixels in the image.

Hit : if any on pixel in the kernel covers an on pixel in the original image

Based on this two ideas, we can define :

- **Erosion** : Mathematically, the erosion of an image by a structuring element is given by Equation 2.3 :

$$G(x, y) = \begin{cases} 1 & \text{if } s \text{ fits } f \\ 0 & \text{Otherwise} \end{cases} \quad (2.3)$$

Where f is the image, s is the structuring element and (x,y) are the coordinates of the center pixel.

In other terms, the operation consists of eliminating the pixel that is not surrounded from all sides by white pixels.

- **Dilation** : It is the opposite of erosion ; it expands the boundaries of an object. When f is the original image and s is the structuring element, the dilation is defined by Equation 2.4 :

$$G(x, y) = \begin{cases} 1 & \text{if hits } f \\ 0 & \text{Otherwise} \end{cases} \quad (2.4)$$

Where f is the image, s is the structuring element and (x,y) are the coordinates of the center pixel.

In other words, the dilation adds pixels that have at least one white pixel in their neighborhood.

Now that we have an understanding of what dilation and erosion are, we can proceed to define the Top Hat and Bottom Hat filters.

- **Top-hat filtering** computes the morphological opening of the image, which consists of erosion that shrinks the foreground objects and enlarges foreground holes, followed by dilation that adds pixels to the boundaries of objects. Thus, opening generally smooths the contour of an object, breaks narrow isthmuses, and eliminates thin protrusions. The Opening of an image by a structuring element is defined by formula 2.2.1 [44] :

$$A \circ B = (A \ominus B) \oplus B \quad (2.5)$$

where :

- \oplus and \ominus : erosion and dilation, respectively,
- A : the image and B is the structuring element,
- \circ : the opening of an image with a structuring element.

The Figure 2.1 is a 2D and 3D representations of Top Hat filter :

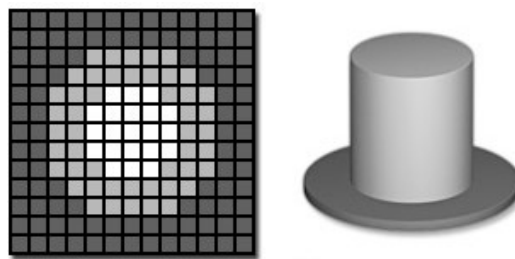


Figure 2.1 : Top Hat Filter

- **A bottom-hat filter** computes the morphological closing of an image in order to eliminate the holes present in that last. A morphological closing is a dilation followed by an erosion. It tends to smooth sections of contours but it generally fuses narrow breaks and long thin gulfs, eliminates small holes, and fills gaps in the contour [44]. It is given by the Equation 2.6 :

$$A \bullet B = (A \oplus B) \ominus B \quad (2.6)$$

where :

- \oplus and \ominus : erosion and dilation, respectively,
- A : the image and B is the structuring element,
- \bullet : the closing of an image with a structural element.

D. Sobel Filter

Sobel filter is a derivate mask used to extract horizontal and vertical edges in an image. It calculates the gradient of image intensity at each pixel within the image. Its kernels are the followings (Figure 2.2) [44] :

-1	0	+1
-2	0	+2
-1	0	+1

G_x

+1	+2	+1
0	0	0
-1	-2	-1

G_y

Figure 2.2 : Sobel Filter Kernels

E. Canny Filter

Besides achieving a low error rate, the Canny filter is an analytic edge detection operator that aims to have a well-localized edge point and a single edge point response. It is based on, first, a Gaussian filter, followed by applying a Gradient with Non-maxima Suppression, and finally, a Dual-threshold Segmentation and edge stitching where [44] :

- **Gradient with Non-Maxima Suppression :**

It is the stage where the smoothed image is filtered with Sobel kernel in both horizontal and vertical directions to get first derivatives G_x and G_y . From these two images, we can find edge gradient (Equation 2.7) and the direction equation (Equation 2.8) for each pixel as follows :

$$EG = \sqrt{G_x^2 + G_y^2} \tag{2.7}$$

$$\theta = \tan^{-1}(G_y/G_x) \tag{2.8}$$

where :

G_x and G_y : the first derivatives of the intensity function,

EG : the edge gradient,

θ : the angle of the pixel in question.

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, the center pixel is checked if it is a local maximum in its neighborhood in the direction of gradient. The Figure 2.3 [45] illustrates this step :

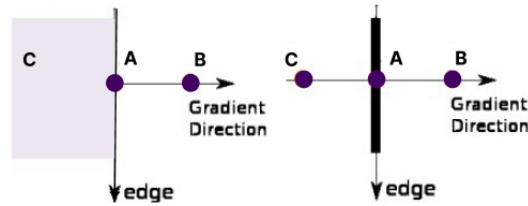


Figure 2.3 : Non-maximum Suppression

Point A is on the edge (in vertical direction). Gradient direction is normal to the edge. Point B and C are in gradient directions. So point A is checked with point B and C to see if it forms a local maximum. If so, it is considered for next stage, otherwise, it is suppressed.

Finally, the result we get is a binary image with thin edges.

• **Dual-threshold segmentation :**

It decides which are all edges are really edges and which are not. For this, we need two threshold values, *minVal* and *maxVal*. Any edges with intensity gradient more than *maxVal* are sure to be edges and those below *minVal* are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded. The Figure 2.4 illustrates the Dual-Threshold Segmentation [45] :

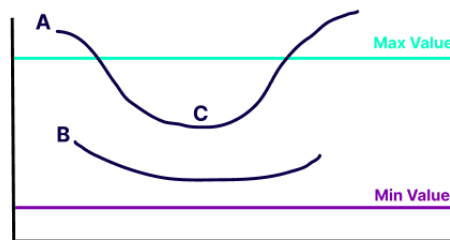


Figure 2.4 : Dual-Threshold Segmentation

The edge A is above the *maxVal*, so it is considered as "sure-edge". Although edge C is below *maxVal*, it is connected to edge A, so that also considered as valid edge and we get that full curve. But edge B, although it is above *minVal* and is in same region as that of edge C, it is not connected to any "sure-edge", so that is discarded. So it is very important that we have to select *minVal* and *maxVal* accordingly to get the correct result.

• **Edge Stitching with Hysteresis Thresholding :**

The algorithm known as "Edge Stitching with Hysteresis Thresholding" can be described as follows :

- Set two thresholds, denoted as τ_{low} and τ_{high} ,
- From the result of the Non-Maxima Suppression step, identify a point q_0 where a local maximum occurs,
- Initiate edge chain tracking at pixel location q_0 , selecting one of the two available directions,
- Continue tracing the edge chain until the gradient magnitude falls below τ_{low} .

In other words, we use the high threshold to start the edge curves and the low threshold to maintain their continuity. The Figure 2.5 illustrates this step :

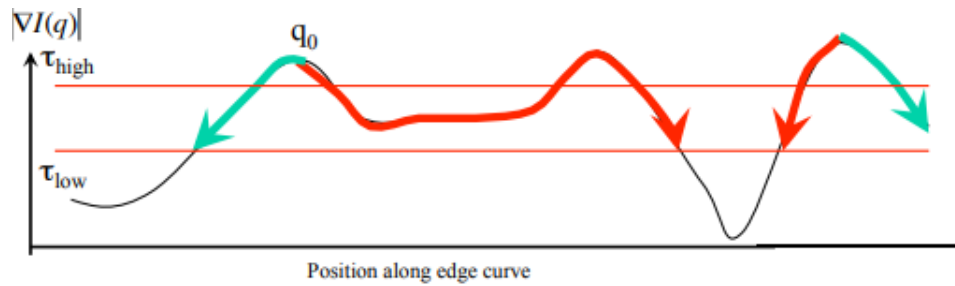


Figure 2.5 : Edge Stitching with Hysteresis Thresholding

By employing this approach, we establish a connection between the different points along the edges.

2.2.2 Histogram Equalization

As it is known, the histogram of an image is the distribution of the intensity levels present in the image. When the distribution is not quite good, i.e. intensity levels are not equally distributed but are compact in a certain region, resulting in a bad image brightness, we need to transform the histogram in order to improve the image quality.

One of the transformations that we can use is Histogram Equalization, also called Histogram Linearization Transformation. Histogram Equalization consists in attributing an output intensity to the new pixel for each input pixel based on the Equation 2.9 :

$$s_k = \frac{(L - 1)}{MN} \sum_{j=0}^k n_j \quad k = 0, 1, 2, \dots, L - 1 \quad (2.9)$$

where :

s_k : the histogram's values,

MN : the total number of pixels in the image,

n_j : the number of pixels that have intensity r_k of which we are calculating the new histogram value,

L : the number of possible intensity levels in the image.

The Figure 2.6 shows an example of applying an Histogram Equalization :

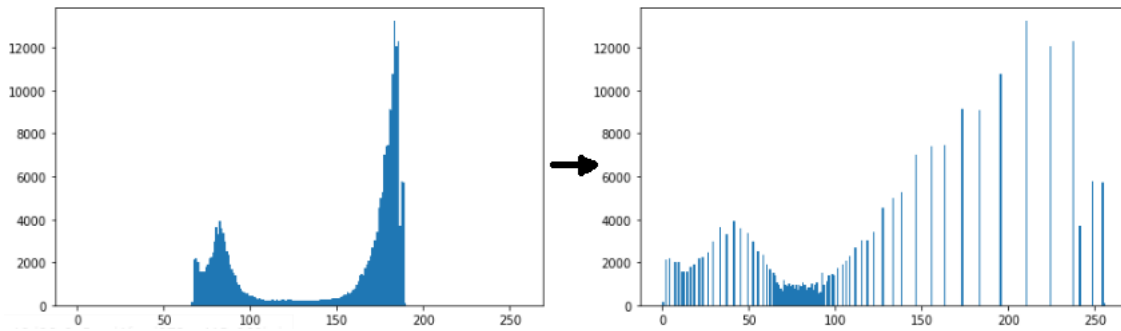


Figure 2.6 : Histogram Equalization example

2.2.3 Contrast

The amount of variation between the intensity levels of pixels in an image is called contrast. As a result, images with high contrast display a greater number of gray-scale levels than images with low contrast [44].

2.2.4 Gamma Correction (Power Law transformation)

One other transformation used to adjust the brightness of an image, in addition to Histogram Equalization, is the Gamma Transformation. It is generally used to manipulate contrast [44]. It is defined as Equation 2.10 :

$$s = cr^\gamma \quad (2.10)$$

where :

s : the intensity level of the new pixel.

r : the intensity level of the input pixel.

c : a constant.

γ : the transformation constant.

The curves representing the Gamma Correction function with different γ values are shown below (Figure 2.7) [44] :

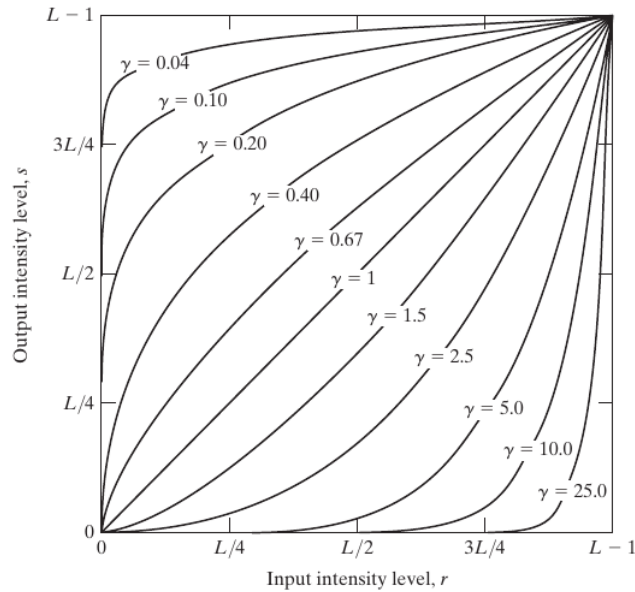


Figure 2.7 : Curves of Gamma Correction

2.2.5 Thresholding

Thresholding is a binarization technique used to separate objects present in an image from the background when the intensity distributions of these objects are sufficiently distinct.

The main idea of this technique is to choose a threshold according to which, the pixels of the image will be transformed to zero (black), if the input intensity is lower than the threshold, and one (white) if it is greater than that threshold.

Depending on the way this threshold is determined, we define two types of thresholding : static and adaptive. When the threshold is global (unique), applicable over the entire image and fixed from the beginning, we talk about the static thresholding ; otherwise, i.e. if it is dynamically fixed (threshold automatically estimated), the thresholding is adaptive [44].

2.2.6 Hough Transform

The Hough Transform is a technique that can be used to isolate features of a particular shape within an image. It requires that the desired features be specified in some parametric form.

The classical Hough Transform is most commonly used for the detection of regular curves such as lines, circles, ellipses, etc. Despite its domain restrictions, the classical Hough Transform retains many applications, as most manufactured parts contain feature boundaries that can be described by regular curves.

The main advantage of the Hough Transform technique is that it is tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise [44].

2.3 Features Extraction

2.3.1 Gabor Filter

A Gabor filter is a linear filter used in image processing for edge detection, texture classification and features extraction.

It is a band-pass filter, i.e. it passes frequencies in a certain band and attenuates the other frequencies outside such band. A Gabor filter is a Gaussian modulated by a plane wave [46]. Thus it is given by formula 2.11 in spatial domain :

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \exp\left[i\left(2\pi\frac{x'}{\lambda} + \psi\right)\right] \quad (2.11)$$

$$x' = x \cos \theta + y \sin \theta$$

$$y' = y \cos \theta - x \sin \theta$$

Where :

(x,y) : the coordinates of the pixel,

λ : the wavelength of the plane wave,

θ : the orientation of the normal to the parallel stripes of the Gabor function,

σ : the standard deviation of the Gaussian component,

γ : the aspect ratio that defines the ellipticity of the function support,

ψ : the phase of the plane wave.

Several filters can be defined by changing the parameters $\lambda, \theta, \sigma, \psi$ and γ . A Gabor filter bank is a set of Gabor filters with different parameters. Different low-level features can be extracted from the original image via the convolution operation by varying the Gabor parameters, Making the gabor filter a powerful tool for features extraction [46].

2.3.2 Wavelet Transform

Wavelet transforms are mathematical tools for analyzing data where features vary over different scales. For images, features include edges and textures [47].

Wavelet analysis is based on decomposing signals/images into shifted and scaled versions of wavelets. A Wavelet is a rapidly decaying, wave-like oscillation that is localized in time. The equations representing the Wavelet Transform are (Equation 2.3.2) :

$$x(t) \approx \sum_k s_{J,k} \phi_{J,k}(t) + \sum_{j=J}^1 \sum_k d_{j,k} \psi_{j,k}(t) \quad (2.12)$$

Where :

x : the input signal,

$\phi_{J,k}, \psi_{j,k}$: the Father and Mother Wavelets respectively,

$s_{J,k}, J_{J,k}$: the coefficients of the Father and Mother Wavelets respectively.

Wavelets have two basic properties : scale and location. Scale (or dilation) defines how stretched a wavelet is. This property is related to frequency as defined for waves. Location defines where the wavelet is positioned in time (or space) because wavelets are only non-zero in a short interval [48]. The Figure 2.8 shows some common wavelets :

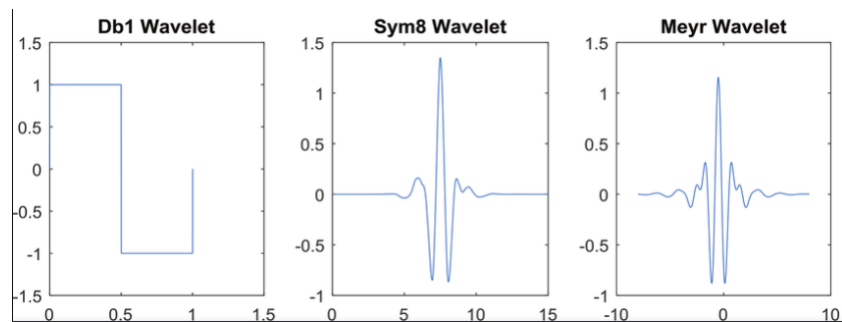


Figure 2.8 : Examples of different types of wavelets

Overall, wavelets can represent data across multiple scales and different wavelets can be used depending on the application [47].

2.3.3 Local Binary Patterns

Local Binary Pattern (LBP) is a prevalent method employed in the field of texture recognition. It involves the computation of binary patterns within digital images. These resulting features, derived from the input images, are subsequently represented through a binary image visualization.

To compute the Local Binary Pattern, the technique analyzes the intensity of a rectangular region. It compares each pixel in a 3x3 window (which consists of 9 pixels) with the center pixel [49].

When comparing, if a neighboring pixel has a value greater than or equal to the center pixel, it is assigned a value of 1, otherwise, it is assigned a value of 0 as the following Equation (2.3.3) :

$$s'(x) = \begin{cases} 1, & \text{if } s(x) \geq s_{pc} \\ 0, & \text{otherwise} \end{cases} \quad (2.13)$$

Where :

x : the pixel of the neighborhood,

s, s' : the old and new pixel value respectively,

s_{pc} : the center pixel value,

By treating the 3x3 matrix as a histogram, we can create a descriptive representation of the Local Binary Pattern. This technique enables us to capture and analyze the frequencies of different patterns observed within the 3x3 window.

By converting the pixel values into histogram bins, we can visualize the distribution of patterns and gain insights into the texture characteristics of the image. The Figure 2.9 illustrates the computation of the local binary pattern on the 3x3 matrix, showcasing the transformation from pixel values to a histogram representation :

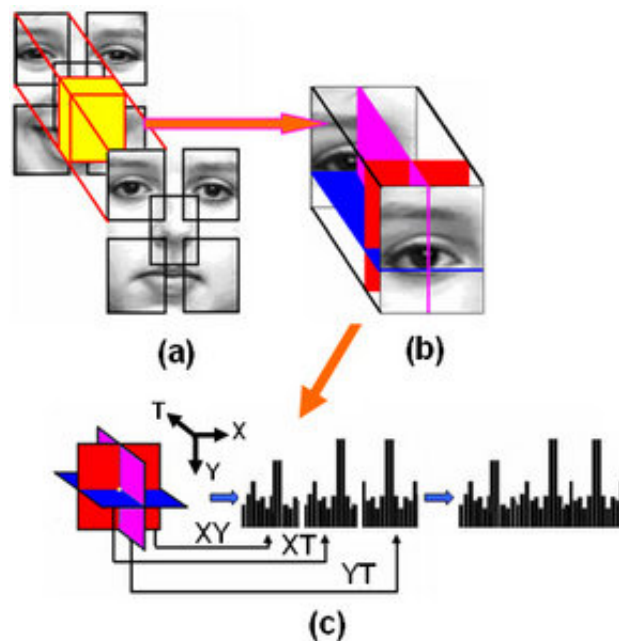


Figure 2.9 : LBP Computation

2.3.4 Gray Level Co-occurrence Matrix

The Grey Level Co-occurrence Matrices (GLCM) is a pioneering technique in Texture Features Extraction, which has found extensive applications in various texture analysis fields. Despite the emergence of newer methods, it continues to hold significance as a crucial feature extraction technique.

The GLCM is a matrix that represents the frequency of occurrence of different sets of pixel gray levels in an image. It is computed for a selected pair of distance and angle. For each pixel and its neighboring pixels, relative recurrences of the specified pair are calculated, as illustrated in Figure 2.10 [50] :

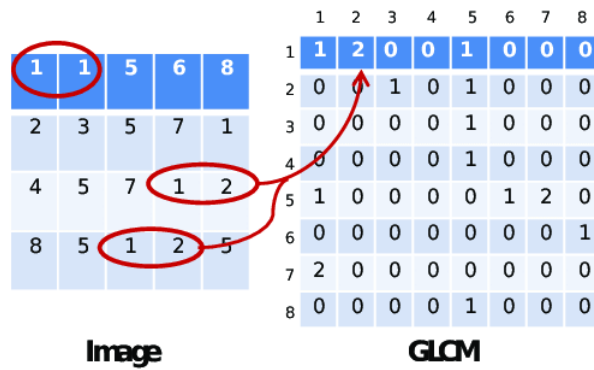


Figure 2.10 : GLCM Matrix Calculation

The resulting matrix is then normalized by dividing it by the sum of all its elements to obtain a normalized matrix.

Through the analysis of GLCM, we can extract crucial texture features that offer valuable insights into the image's characteristics. These features encompass contrast, energy, homogeneity, and entropy. Contrast measures the disparities in gray levels, energy denotes the local homogeneity and entropy, homogeneity gauges the frequency of non-zero values, while entropy quantifies the randomness or uncertainty within the GLCM [50].

The four features : Contrast, Entropy, Homogeneity, and Energy, extracted from GLCM are summarized as follow :

- *Contrast* in the GLCM matrix is manifested as the difference in gray levels. It calculates the intensity between a pixel and its neighbor. Its formula is Equation 2.14 :

$$\text{Contrast} = \sum_{i,j=0}^{N-1} P_{ij}(i - j)^2 \quad (2.14)$$

- *Entropy* Feature represents the quantity of energy. It is given by Equation 2.15 :

$$\text{Entropy} = \sum_{i,j=0}^{N-1} -\ln(P_{ij}) P_{ij} \quad (2.15)$$

- *The Energy* feature computes the local homogeneity, which is closely related to entropy. Its value ranges from 0 to 1, indicating the level of homogeneity present in the analyzed region. It follows Equation 2.16 :

$$\text{Energy} = \sum_{i,j=0}^{N-1} (P_{ij})^2 \quad (2.16)$$

- *The Homogeneity* feature calculates the presence of non-zero elements in the GLCM, representing the inverse of the contrast weight. Its value ranges from 0 to 1, indicating the degree of uniformity or homogeneity in the analyzed area. It is defined by Equation 2.17 :

$$\text{Homogeneity} = \sum_{i,j=0}^{N-1} \frac{P_{ij}}{1 + (i - j)^2} \quad (2.17)$$

where :

$P_{i,j}$: Element i,j of the normalized symmetrical GLCM.

N : Number of gray levels in the image as specified by Number of levels in under

In summary, GLCM is a powerful tool for texture feature extraction, allowing us to capture valuable information about the texture patterns present in the image. [51]

2.4 Convolutional Neural Network

2.4.1 Convolution Filters

The 2D convolution layer, commonly abbreviated as conv2D, is the most frequently used type of convolution in neural networks. In a conv2D layer, a filter consists of a set of kernels, with each kernel corresponding to an input channel in the layer. Each kernel is distinct and serves a specific purpose in the convolution process.

A conv2D layer in a neural network performs a 2-dimensional convolution operation on input data. Here's a brief explanation of how it works [49] :

- **Input** : The conv2D layer takes as input a 3-dimensional tensor representing an image or a feature map. The dimensions are typically (height, width, channels).
- **Filters** : The layer consists of multiple filters, each comprising a set of kernels. The number of filters determines the number of output channels.
- **Convolution operation** : The layer applies convolution by sliding each kernel across the input tensor. At each position, an element-wise multiplication is performed between the kernel and the corresponding input patch. The results are summed up to produce a single value.
- **Activation** : An activation function is applied to the summed value, introducing non-linearity to the architecture. One of the common activation function is ReLU (Rectified Linear Unit) which is defined by Equation 2.18 :

$$ReLU(x) = \max(0, x); \forall x \in \mathbb{R}. \quad (2.18)$$

- **Output** : The conv2D layer generates a new tensor called the feature map, which represents the output of the convolution operation. It has dimensions determined by the number of filters and the spatial dimensions of the input.

The Figure 2.11 [52] summarizes the concept of Convolutional filters :

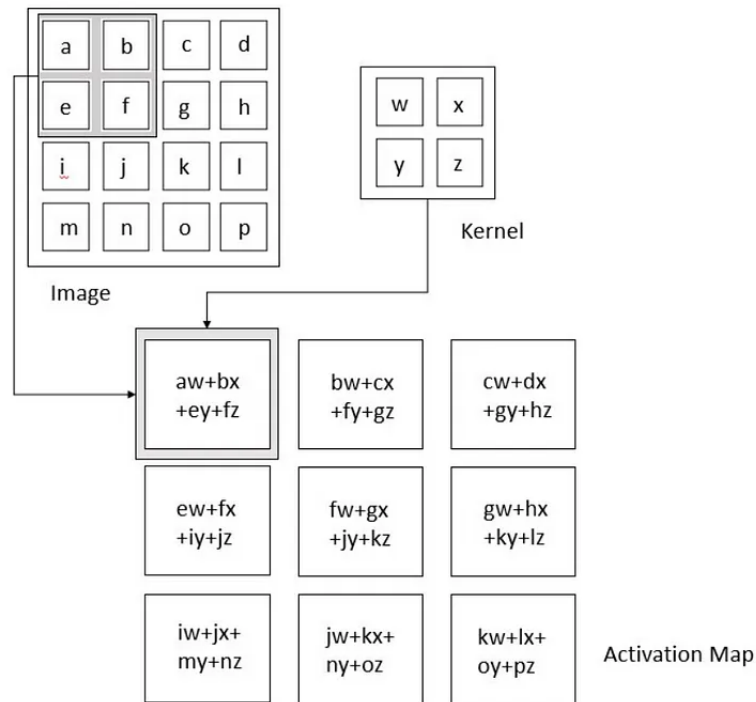


Figure 2.11 : Convolution Operation

2.4.2 Pooling

The pooling layer is generally applied after each convolutional layer. It performs a down-sampling operation based on the spatial dimensions, specifically width and height, and the stride.

This layer helps reduce the number of parameters and computational power required while preserving the most important features. In practice, the image is divided into small adjacent square cells, spaced apart by a stride, to avoid losing too much information. The number of output images remains the same as the input images, but each image has a reduced number of pixels.

There are two main types of pooling: max-pooling and average pooling. Max-pooling returns the maximum value within the part of the image covered by the pooling window, while average pooling returns the average of all the values in that region. In practice, max-pooling is more commonly used and tends to work better than average pooling [49].

The Figure 2.12 [52] illustrates the Max Pooling operation:

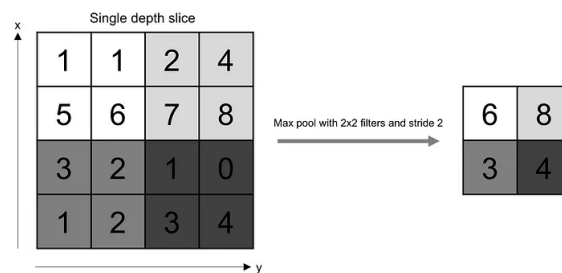


Figure 2.12 : Pooling Operation

2.4.3 Flatten Layer

The flatten layer is typically used as a transition layer between convolutional layers and the prediction block in deep learning models. It reshapes the input tensor into a one-dimensional vector. It takes the multidimensional input, such as an image or a feature map, and flattens it into a single continuous vector without changing the content [49].

The purpose of the flatten layer is to convert the spatial information present in the input tensor into a format that can be passed to the neural network for further processing. The Figure 2.13 [52] shows the Flattening operation :

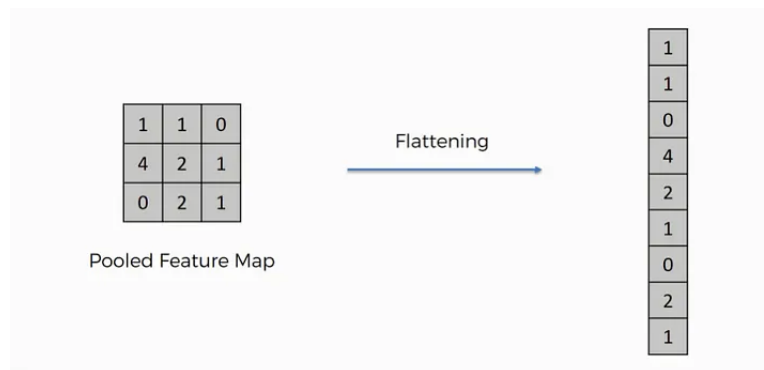


Figure 2.13 : Pooling Operation

2.4.4 Stride and Padding

The Convolution operation as well as the Pooling or any other operation based on a kernel, are influenced by parameters like *Stride* and *Padding*.

The **Stride** determines the step size of the kernel while sliding over the input, affecting the size of the output, in other words Stride is the number of pixels shifts over the input matrix [49].

The **Padding** which refers to the addition of extra elements (typically zeros) around the edges of an input tensor or image, can be added to the input to preserve spatial dimensions or avoid border effects. The Figure 2.14 [52] illustrates the effect of the Padding on the output image size after applying a filtering :

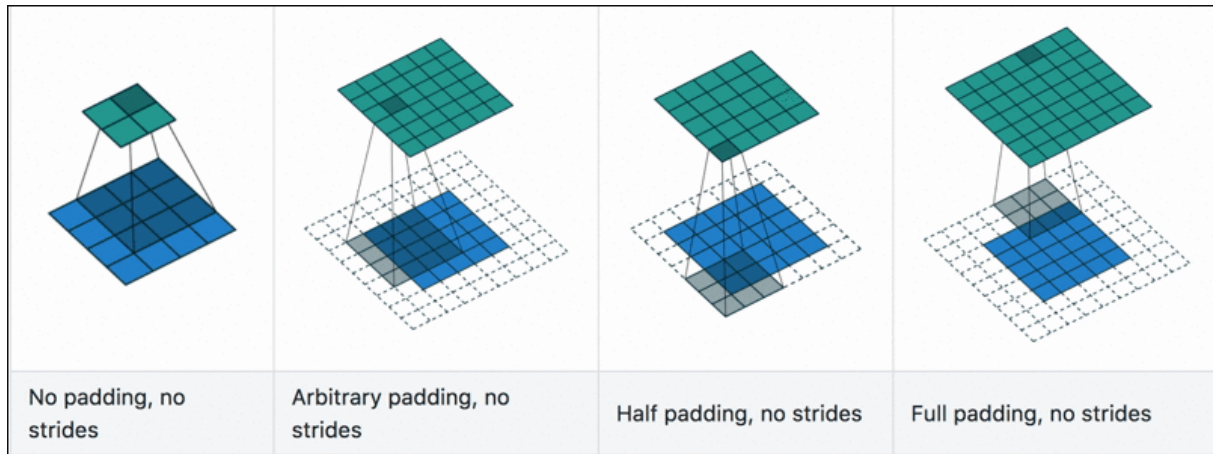


Figure 2.14 : Padding effects on Output Image Size

2.4.5 Dropout

Dropout is a regularization method employed during the training phase to address overfitting by randomly deactivating a specified number of neurons. This strategy guarantees that no individual neuron excessively influences the learning process [49].

2.4.6 Adam Optimizer

Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network [49].

2.4.7 Sparse Categorical Cross Entropy Loss

In the classification problems that have a multi-class single label, the Sparse Categorical Cross Entropy (SCCE) can be used. It produces a category index of the most likely matching category. It is defined by the next equation 2.19 :

$$SCCE = - \sum_{i=1}^n T_i \cdot \log(S_i) \quad (2.19)$$

where S_i are the Softmax probabilities and T_i the labels.

Note :

If we have an input image of size $W \times W \times D$, an operation, like pooling or convolution, then the size of output volume can be determined by the following formula 2.20 :

$$W_{out} = \left\lceil \frac{W + 2P - F}{S} \right\rceil + 1 \quad (2.20)$$

Where :

F : size of the kernel,

S : stride,

P : padding.

2.5 Machine Learning Methodes

2.5.1 Support Vector Machines

Support Vector Machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection. SVM finds the best possible line or boundary (called a hyperplane) that separates different classes of data points.

Here's a brief explanation of how SVM works :

Classification : In classification tasks, SVM aims to divide data points into different categories by finding an optimal hyperplane. The hyperplane is chosen to maximize the margin, which is the distance between the hyperplane and the nearest data points of each class.

Support Vectors : SVM identifies a subset of data points called support vectors that are closest to the hyperplane. These support vectors play a crucial role in defining the hyperplane and determining the decision boundary.

Non-linear Separation : SVM can handle non-linearly separable data by transforming the original input space into a higher-dimensional feature space. This allows SVM to find non-linear decision boundaries by implicitly projecting the data into a higher-dimensional space [49].

The figure 2.15 is an example that can explains the SVM :

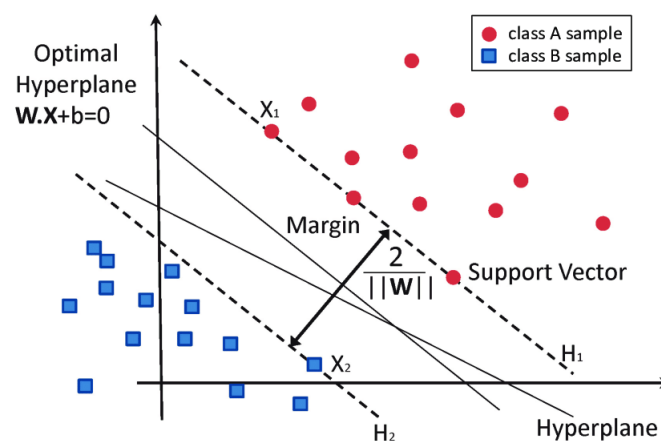


Figure 2.15 : SVM example

The main advantages of support vector machines are the effectiveness in high dimensional spaces and the memory efficiency.

2.5.2 Random Forest Classifier

Random Forest is a highly popular supervised learning algorithm based on decision trees, known for its flexibility and user-friendliness. It is capable of tackling classification problems.

To build a Random Forest, numerous decision trees are combined, with each tree trained on a different subset of observations. To make predictions, the Random Forest aggregates the individual predictions from each tree by averaging them [14].

The Figure 2.16 [14] provides a explanatory illustration of the method :

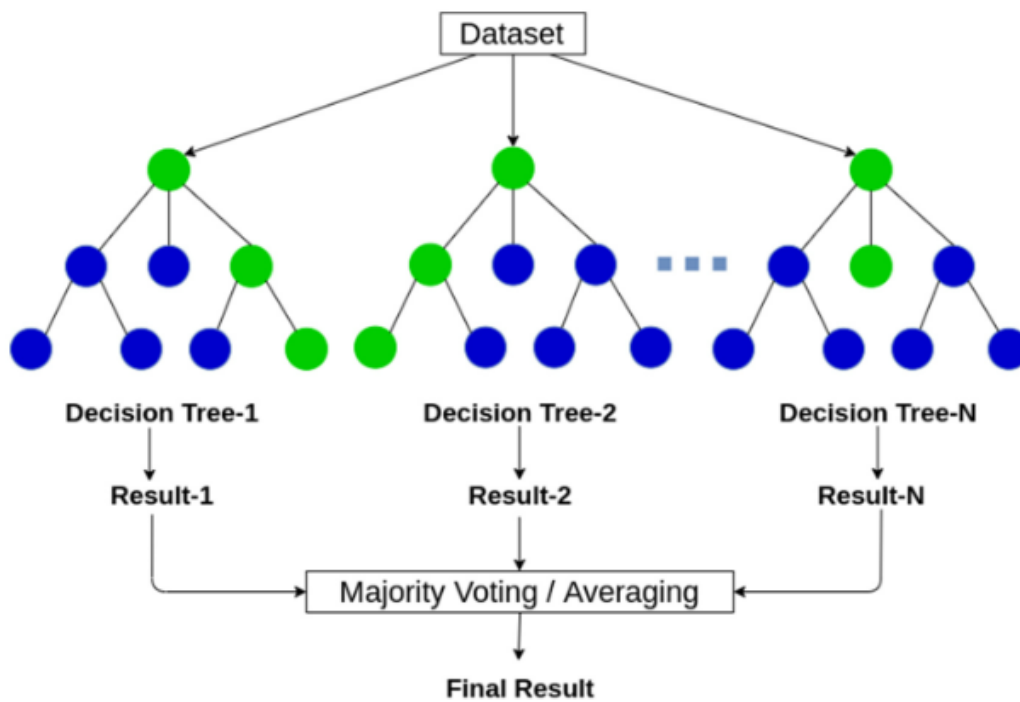


Figure 2.16 : Random Forest Classifier

Random Forest offers several advantages, we can mention some :

- It addresses the issue of overfitting by averaging predictions from multiple trees. As a result, Random Forest achieves higher predictive accuracy compared to a single decision tree.
- Moreover, the Random Forest algorithm aids in identifying important features within a dataset, providing valuable insights [14].

2.5.3 Accuracy

Accuracy is a common metric used to quantify the extent to which a model makes correct predictions. The formula for calculating accuracy is provided below 2.21 [49] :

$$Accuracy = \frac{TP + TN}{FP + FN + TP + TN} \quad (2.21)$$

Where :

TP, FP : True and False Positives predictions respectively,

TN, FN : True and False Negatives predictions respectively,

2.5.4 Epochs

An epoch represents a complete iteration through all the training data during the training process of a machine learning model [49].

2.6 Implementation

2.6.1 IP Block

An Intellectual Property (IP) block, also known as an IP core, is a pre-designed and reusable module or component that serves a specific function or implements a specific feature in an FPGA design.

An IP block can be a complete subsystem, such as a microprocessor or a memory controller, or it can be a smaller functional unit, such as an Arithmetic Logic Unit (ALU) or a Digital Signal processor (DSP), intended for integration into larger designs [53].

2.6.2 Function Call Graph

A Function Call Graph is a graphical representation of the relationships between different function calls within a program. It shows how the functions in a program interact with each other, allowing developers to understand the flow of the program and identify potential performance issues [54]. Here's an example of a Call Graph (Figure 2.17) :

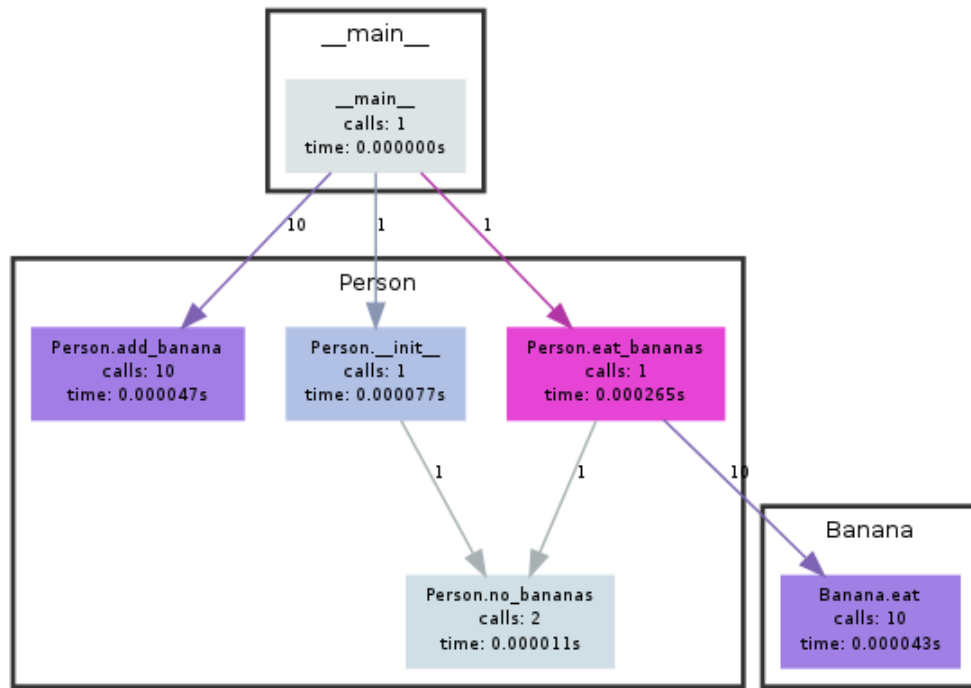


Figure 2.17 : Call Graph Example

2.6.3 AXI Protocol

AXI, which means *Advanced eXtensible Interface*, is an interface protocol defined by ARM as part of the AMBA (Advanced Microcontroller Bus Architecture) standard. It is especially prevalent in Xilinx’s Zynq devices, providing the interface between the processing system and programmable logic sections of the chip, in other terms, it ensures the communication between the different IP Cores of the FPGA [55].

There are 3 types of AXI4-Interfaces [55] :

- **AXI4 (Full AXI4)** : For high-performance memory-mapped requirements, it is used to transfer large amount of data such as arrays and pointers.
- **AXI4-Lite** : For simple, low-throughput memory-mapped communication, so it is mainly used for transfer of small data like scalars (for exemple, to and from control and status registers, flags etc).
- **AXI4-Stream** : For high-speed streaming data, thus used when we have a data flow (Data tranfers from an IP block to another without the involvment of the CPU of the Memory Block).

2.6.4 Netlist

A Netlist is a structural representation of a digital design of a specific system at the gate level, capturing the connectivity and relationships between the logic elements in the design [49].

2.7 Conclusion

In conclusion, this chapter served as an introduction to the detailed presentation of our proposed approach for Iris Recognition Systems. We highlighted the key concepts in Image Processing, Machine/Deep Learning, and FPGA, needed as a prerequisite to exploring the specific details of our approach.

By providing an overview of the necessary terminology, we aimed to enhance understanding and facilitate comprehension of the subsequent discussions. With this foundational knowledge in place, we can now proceed to the comprehensive presentation of our approach, covering each sub-stage of the system and addressing implementation aspects.

Chapter 3

Proposed Approach and Results

3.1 Introduction

In this chapter, we aim to offer a thorough and detailed presentation of our proposed approach. Our focus will be on presenting a comprehensive overview of each sub-stage of the system, starting from the initial phase and progressing towards the final stage. Furthermore, we will delve into the results obtained, providing in-depth explanations and valuable insights along the way.

3.2 Iris Detection

In our system, the pre-processing sub-stage marks the crucial initial phase of both Image Processing and Deep Learning solutions. It plays a vital role in improving the quality and suitability of the images for subsequent usage. This sub-stage involves filtering, enhancing, and resizing the images to meet the fixed image dimension requirement of the model to which they will be fed. In this section, we will outline the primary methods we employed to fulfill these objectives.

3.2.1 Preprocessing - Reflections Reduction

In iris detection, reflections pose a significant problem that can impact the iris recognition. Therefore, to mitigate information loss, we have proposed a contrast manipulation-based method. This method aims to extract the valuable information that is obscured by reflections, ensuring that it is not overlooked.

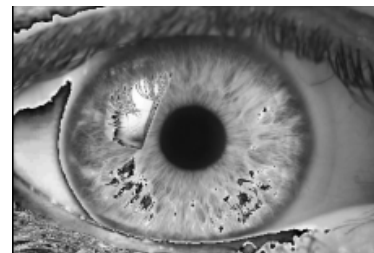
The approach adopted consists of :

- **Isolate the reflections from the iris** : since the reflections have high intensity levels, we apply a binary thresholding with a threshold of 230. Then we use the output as a mask that isolates reflections in the original image.
- **Adjust the brightness of the isolated reflections using Histogram Equalization** : this technique helps to balance the distribution of pixel intensities and enhance the hidden textures within the reflections.

Here's an example that shows the difference between an input image and the image with reduced reflections (Figure 3.1) :



(a) Original image



(b) Reduced Reflections

Figure 3.1 : Comparison between an original image and image got after reflections reduction

Our new proposed approach preserves the entire iris region, even the parts affected by reflections.

3.2.2 Edge Detection

In order to recognize the iris and match it with a person, we need to detect its placement, but first, we detect its outer and inner edges in a full eye image as follows (Figure 3.2) :



Figure 3.2 : Iris boundaries detection

Once the images are preprocessed, we propose to use the Canny edge detection algorithm or Sobel filter.

The Canny edge detection algorithm is Gaussian-based operator that detects a wide range of edges in images, it is composed of 5 steps : Noise reduction ; Gradient calculation ; Non-maximum suppression ; Double threshold and finally Edge Tracking by Hysteresis.

This algorithm is known for its ability to extract image features without affecting or altering the information and its non-susceptibility to noise. However, it has a complex computation and it is time-consuming.

However, edge detection using Canny filter may not provide perfect results, some edges may not be accurate edges due to noise in the image. To address this issue, we added a crucial step to improve the image for further use, which is applying double threshold to convert the image from gray-scale to binary. This step allows the classification of each pixel as either background (lashes, eyelids, skin and white of the eye) or useful pixels (the iris). The threshold values of 35 and 110 were set for the inner and outer iris edges, respectively, enabling us to successfully detect edges for the whole dataset with approximately the same ratios.

To mitigate potential information loss caused by thresholding, it is crucial to choose an appropriate threshold value tailored to the specific image. This ensures that valuable details are not inadvertently discarded. If the selected threshold value is unsuitable for a particular image, an alternative approach is to utilize a Sobel filter for edge detection. By employing the Sobel filter in such cases, we can detect edges without relying solely on thresholding, thereby preserving more information from the iris.

As well as Canny filter, Sobel filter is a well known method to detect edges. It calculates the partial derivatives of the function $I(x,y)$ that evaluates the pixel intensity I at pixel location (x,y) . Due to its ability to detect all edges without the need for thresholding, the Sobel filter increases the likelihood of detecting iris boundaries in an image. However, this advantage comes at the expense of computational resources during the subsequent stage of applying the Hough

Transform. Therefore, in comparison to the Canny filter, we prioritize using the Sobel filter in the second position.

We can see the output of this step in the Figure 3.3 :

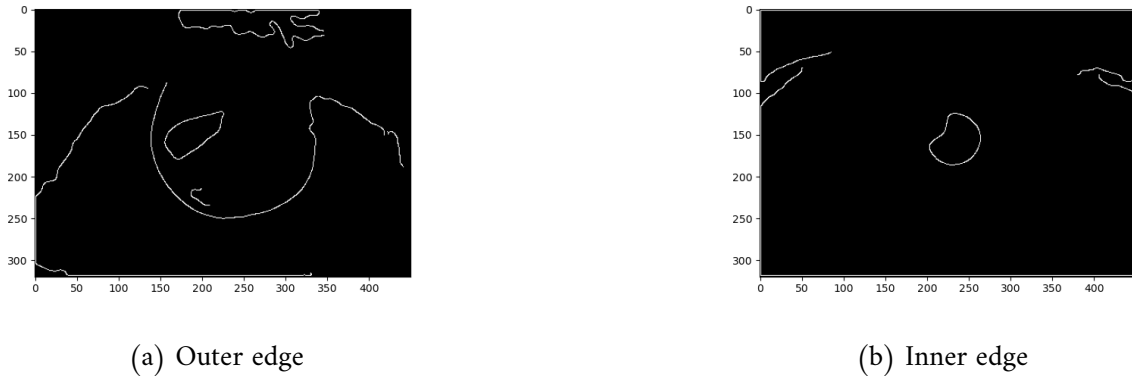


Figure 3.3 : Outer and Inner Boundaries Edge detection

3.2.3 Iris Isolation

Now that we have determined what strong edges and weak edges are, we need to determine and locate the inner and outer boundaries of the iris. Considering the iris to be circular in shape, localizing the iris zone involves detecting the circles present in the image and selecting the two most prominent circles.

Unlike detecting squares or rectangles in images, detecting circles is significantly harder because we cannot rely on approximating the number of points in a contour. That is why we propose using the Circular Hough Transform (CHT) to predict the existence of circular shapes in the provided image.

The CHT consists of :

- First, identify all the probable circles in the image,
- Then, a filtering process is applied to remove any incorrectly detected circular-shaped edges that fall outside the range of the maximum and minimum iris radius,
- After that, the highly linked edges are selected as the edges of the iris, and the rough iris boundary is recognized.

The Figure 3.4 shows the circles found using the CHT on an iris image :

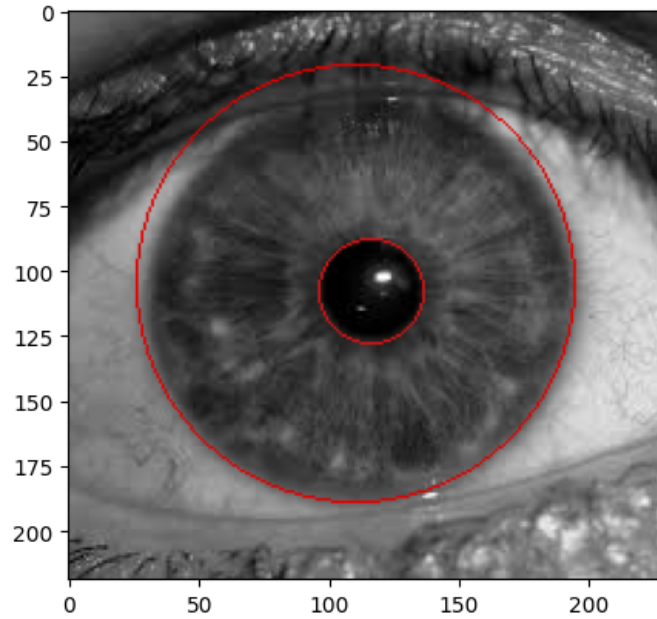


Figure 3.4 : CHT Output Example

The next process after detecting the inner and outer circles of the iris is segmentation. As mentioned earlier, image segmentation can be an extensive and a time consuming process. Therefore, to reduce the complexity of this operation, we utilize the results of the Circular Hough Transform to create a custom mask that can effectively segment the iris. This approach avoids the need for other resource-intensive methods such as utilizing a pre-trained model, which may take several minutes to complete the segmentation of a single image.

The approach followed consists of :

1. Creating a mask A of a circle with the center and radius of the outer circle previously calculated out of the CHT, i.e. creating a binary image that respects $(z < ExtRadius) = 1$ with $z = x^2 + y^2$.
2. Creating a mask B of a circle with the center and radius of the inner circle previously calculated out of the CHT, i.e. creating a binary image that respects $(IntRadius < z) = 1$ with $z = x^2 + y^2$.
3. Multiplying A and B and the input image to get the iris segmented.

The Figure 3.5 represent the created mask used for segmentation :

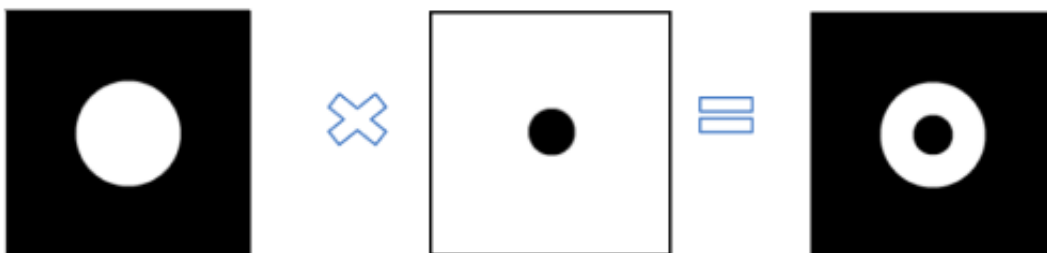


Figure 3.5 : Final Mask

The outcomes of this step are now visible in Figure 3.6 :



Figure 3.6 : Segmented iris of person 224 of IITD dataset

3.2.4 Normalization

The Normalization phase, as mentioned, consists on transforming the circular zone of the iris into a rectangular one. For this, we use the Rubber Sheet Transformation, which is an efficient Cartesian to Polar coordinates system transformation. The transformation equation is given as follow formula 3.1 :

$$I_{new}(r, \theta) = I_{old}(x_0 + (ri + r) \cdot \cos(\theta), y_0 + (ri + r) \cdot \sin(\theta)) \quad (3.1)$$

Where :

(r, θ) : coordinates of the new pixel

(x_0, y_0) : the coordinates of the center of the inner boundary.

ri : radius of the inner boundary.

I : intensity level of new or old pixel.

We note that the Normalization process ensures that the iris region has consistent dimensions. Normalization encompasses the adjustment of iris size caused by alterations in pupil size resulting from changes in external lighting conditions. Additionally, it facilitates the alignment of iris images from different individuals to a standardized size and format, allowing two photographs of the same iris captured in different environments to exhibit identical characteristic features.

The process of normalization enhances the quality of the images by reducing degradation effects, thereby improving clarity and enabling the extraction of precise features for accurate recognition. By transforming the iris region into a rectangular block of fixed dimensions measuring 128x128 pixels, we have successfully unwrapped and standardized its representation using Rubber Sheet method.

For the Rubber Sheet method, it basically extracts the localized circular iris and yields the polar form image of the iris circular ring region as shown in Figure 3.7 :

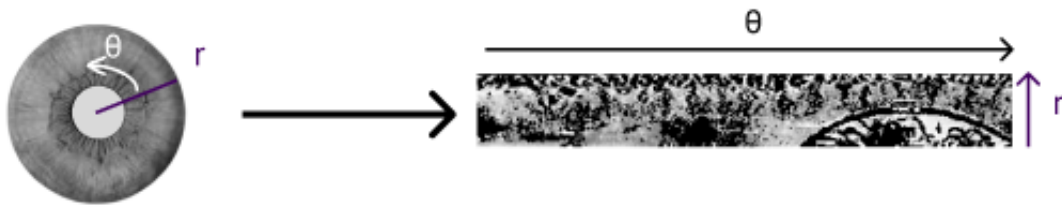


Figure 3.7 : Normalized iris of person 224 of the IITD dataset

3.2.5 Preprocessing - Enhancement

Having a good image quality is one of the necessary requirements we need to ensure. . Thus, in order to improve it, we worked on adjusting its brightness. We applied the method proposed in [1] which consists of four steps :

- First, applying Histogram Equalization.
- Second, reducing eventual unipolar noise due to the previous operation, using a Median filter.
- Furthermore, we apply Gamma Correction to further improve the brightness adjustment. The selection of the Gamma value is based on an analysis of the image's histogram. Depending on the distribution of intensity levels, different Gamma values are applied to achieve the desired correction :
 - 1.1 if the max of brightness is shifted to dark side (index less than 60).
 - 0.8 if the max of brightness is shifted to bright side (index less than 160).
 - 1 if the max of brightness is in between, so that no correction will be applied.
- Finally, we apply Top Hat and Bottom Hat filters to reduce any imperfections that may be present in the image, such as holes caused by imperfect boundary detection or noise. These filters help to refine the image by effectively removing unwanted artifacts and enhancing the overall quality of the iris segmentation.

The Figure 3.8 illustrates the enhancement of one of the normalized images that we have :



(a) Before enhancement



(b) After enhancement

Figure 3.8 : Comparison of before and after enhancement

It is crucial to note that performing enhancement after normalization ensures that, when enhancing the contrast of the iris, the intensity levels of background pixels, which are considered noise, do not impact the newly generated pixels in our enhanced region of interest.

3.2.6 Iris Detection Evaluation

In order to thoroughly evaluate the effectiveness of the preceding steps shown in Figure 3.9, it is imperative to conduct a comparison between the number of images available in each dataset both before and after the iris detection process.

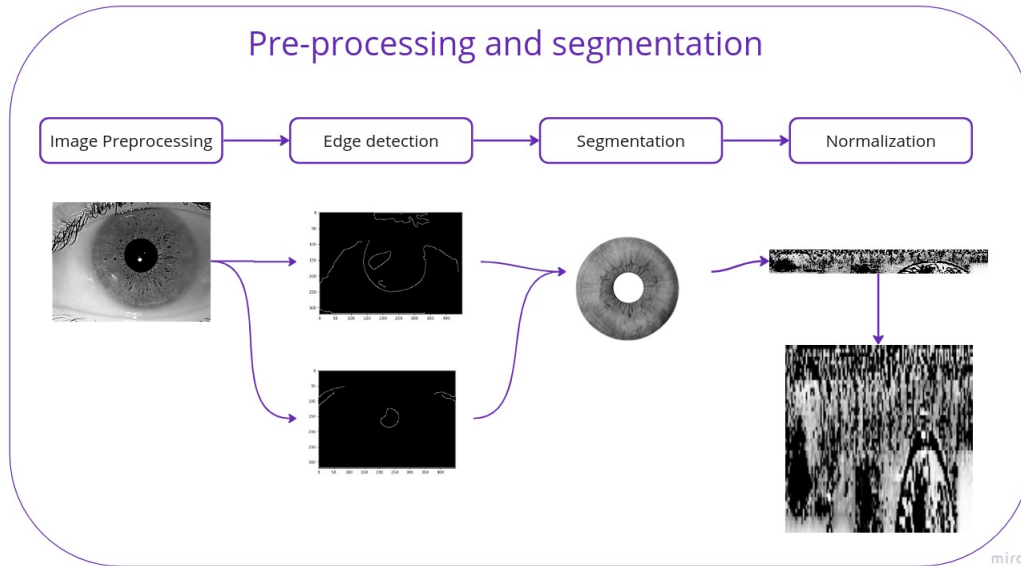


Figure 3.9 : Iris Detection Process

The detailed information regarding each dataset, prior to any preprocessing or segmentation procedures, can be found in Table 3.1.

Datasets	Classes	Images/Class	Images/Dataset	After Segmentation	Effectiveness %
IITD	224	R : 5 L : 5	2240	2240	100,00
MMU	45	R : 5 L:5	450	420	93,33
Phoenix	64	R : 3 L : 3	384	384	100,00
Casia V-4	1000	R : 10 L : 10	20000	11784	58,92

Table 3.1 : Our datasets before and after proceeding with the substage described earlier.

3.2.7 Dataset Preparation

After the process of iris detection, certain images may still lack informative content and have the potential to mislead our model during training. Consequently, we opt to exclude images that consist of more than 70% black pixels, as these pixels do not contribute valuable information.

After proceeding this step, the description of our processed datasets will be as shown in table 4.43 :

Datasets	Images/Dataset	After Data preparation	Effectiveness %
IITD	2240	2240	100,00
MMU	450	399	88,66
Phoenix	384	332	86,45
Casia-Iris V-4	20000	10084	50,42
Total	23084	13055	

Table 3.2 : Our datasets before and after proceeding data preparation

The effectiveness of different datasets can vary significantly. For instance, when considering the Casia-Iris V-4 dataset, the results are not optimal due to the challenges posed by Asian eyes and the difficulty in accurately localizing the iris.

Based on these previous results, we have made the decision to proceed with our project using the Phoenix and IITD datasets, which offer more favorable conditions.

3.2.8 Data Augmentation

Initially, all classes within the various datasets had an equal number of images. However, after preprocessing them (particularly segmentation) and Dataset Preparation, certain images were eliminated due to the absence of iris detection, resulting in an imbalanced dataset with only a few images per specific classes. To tackle this problem and ensure a more balanced dataset and an important amount of data to be fed to the model, we employed Data Augmentation techniques. These techniques allowed us to increase the number of images and achieve a more equitable distribution among the classes.

Regarding the shifting scenario, the human head could rotate a maximum of 45 degrees, which accounts for approximately 12.5% of the total head rotation and is equivalent to a maximum of 15 pixels in terms of image displacement. Additionally, the cropping method can simulate the tolerable lack of data in the images. Through the application of these augmentation methods, we successfully generated additional images that provided a more comprehensive representation of the various classes.

Ultimately, this process resulted in a dataset containing 20 images per class, addressing the initial issue of data imbalance.

An example of person 224 of the IITD dataset is shown in Figure 3.10 :

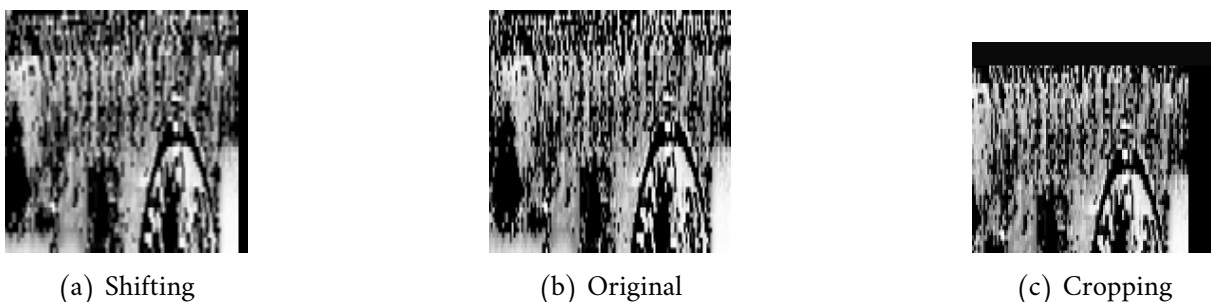


Figure 3.10 : Different Data Augmentation Techniques

The results of the data augmentation, including the details of the number of images per dataset and the total are in table 3.3 :

Datasets	Number of Classes	Images/Classe	Images/Dataset
IITD	224	20	4480
MMU	45	20	900
Phoenix	64	20	1280
Casia v-4	1000	20	20000
Total			26660

Table 3.3 : Our datasets after proceeding data augmentation

3.3 Features Extraction

Once the dataset has been preprocessed and the final versions of the images are obtained, the next step is to extract relevant information from them. However, it's crucial to acknowledge that in Machine Learning, particularly in Deep Learning classification problems, there are often numerous features that contribute to the final classification. As the number of features increases, visualizing and effectively working with the training set becomes progressively challenging. Additionally, many of these features are frequently correlated. This is where Dimensionality Reduction techniques come into play to address this issue.

Dimensionality Reduction involves reducing the number of features in a dataset while preserving as much information as possible. It can be achieved through features selection or features extraction.

Features extraction aims to identify and separate independent components from a mixture of inputs, effectively removing unnecessary features. It reduces the initial set of raw data into more manageable groups for further processing. On the other hand, features selection focuses on determining the importance of existing features in the dataset and discarding less significant ones, without creating new features. As a result, the first step is Features Extraction.

3.3.1 Deep Learning based Methods

DL-based techniques are widely utilized for features extraction. We encounter various architectures that incorporate the features extraction component. In our experiments, we tested the following architectures, each paired with its respective classifier :

A. IRISNet

IRISNet is a Convolutional Neural Network first proposed by Maryim Omran and Ebtesam N. AlShemmary for the iris features extraction and classification in 2020. It is composed of four Convolution layers, six activation Relu layers, three Pooling layers and two fully connected layers to classify and extract features from image automatically without any domain knowledge.

The architecture of the model is shown in Figure 3.11 :

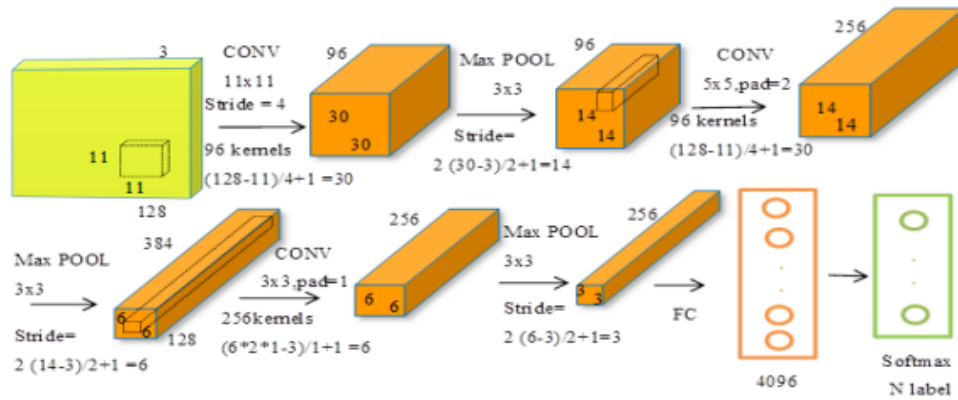


Figure 3.11 : IRISNet Architecture

The most remarkable aspect of this architecture is its ability to achieve high accuracy in iris recognition despite its simplicity and shallow depth. This leads to significant savings in both training time and computational resources during both the training and inference phases.

B. ResNet

Launched in 2015 by Microsoft Research Asia, the ResNet architecture, with its three achievements ResNet-50, ResNet-101 and ResNet-152, obtained very good results in the ImageNet and MS-COCO.

It turns out that the central idea exploited in these models, namely, the residual connections, greatly improves the gradient flow, thus allowing training creation of much deeper models, with tens or even hundreds of layers. However, increasing the depth of the network is not done simply by piling it up.

Deep Networks are difficult to train due to the famous problem of the evanescent gradient : when the gradient is back-propagated towards the preceding layers, the repeated multiplication can make the gradient infinitely small. Therefore, the deeper the network is, the more its performances are saturated, even degrading quickly. The central idea ResNet's task is to introduce what is known as an "identity shortcut connection" or "residual connection", which skips one or more layers, as shown in Figures 3.12[56] and 3.13[56] :

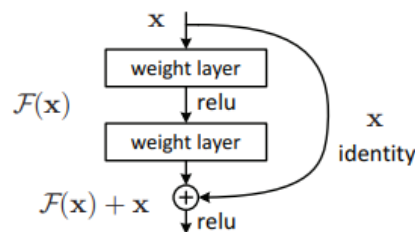


Figure 3.12 : Residual learning : a building block

The authors of the article [56] claim that layer stacking should not degrade network performance, because we could just stack the identity mappings on the current network, and

the architecture resulting would have the same performance. This indicates that the deepest model should only not produce higher learning error than its shallower counterparts.

They assume it is easier to let stacked layers fit a map residual than letting them adapt directly to the desired underlying mapping. This is precisely what the residual block above does.

We then have the various ResNet Layers used from 18 to 152 layers of which we can describe the constitution in the following figure 3.13 :

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 3.13 : Details About ResNet Layers

C. AlexNet

AlexNet is a Convolutional Neural Network that is eight layers deep trained on the ImageNet database. The pretrained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224 [47]. The architecture of the network is shown in the Figure 3.14 [49] :

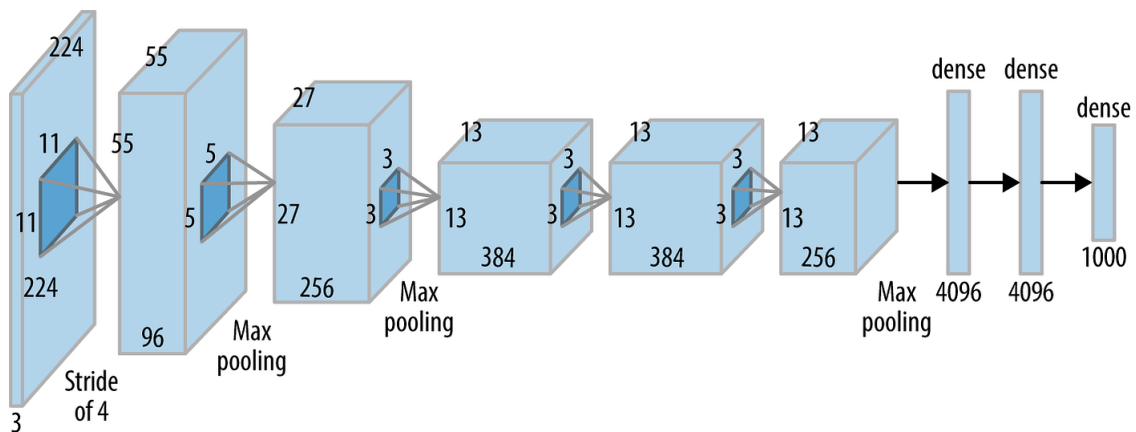


Figure 3.14 : AlexNet Architecture

Multiple previous works used this architecture and had excellent results, that’s what motivated us to test it.

D. Siamese Architecture

Siamese architecture refers to a type of Neural Network architecture that is commonly used for tasks such as image recognition, object tracking, and similarity matching. The term "Siamese" is derived from the famous thought experiment known as "Siamese Twins," where two individuals are physically connected. In the context of Neural Networks, the Siamese architecture typically consists of two or more identical sub-networks that share the same weights and architecture.

In a Siamese network, the input data, such as images or feature vectors, are fed into each sub-network simultaneously. The sub-networks process the inputs independently and produce embeddings or feature representations of the inputs. These embeddings are then compared or combined to determine similarity or dissimilarity between the inputs.

The basic structure of a Siamese network consists of parallel sub-networks that process separate input data. These sub-networks are typically identical in terms of architecture and have shared weights, meaning they learn and update their parameters together.

The Figure 3.15 is a basic example of a Siamese Network :

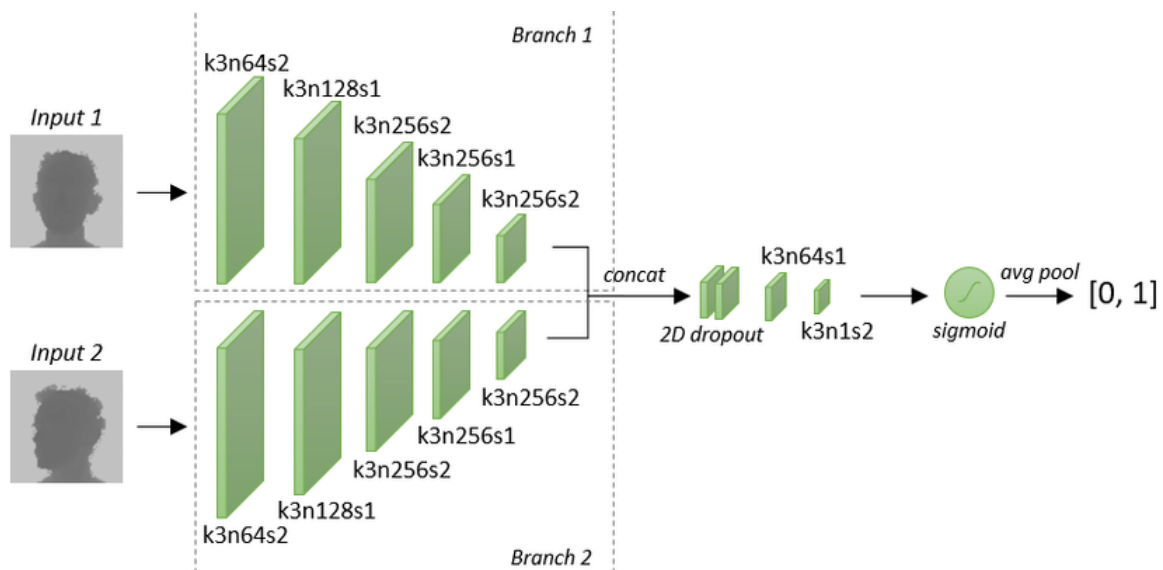


Figure 3.15 : Example of Siamese Architecture

During training, a Siamese network is presented with pairs of inputs, usually representing similar or dissimilar instances. For example, in facial recognition, a pair could consist of two images of the same person (similar) or two images of different people (dissimilar). The sub-networks process each input independently, transforming them into high-dimensional feature representations.

The representations generated by the sub-networks are then compared using a similarity metric, such as Euclidean distance or cosine similarity. The network learns to minimize the distance between similar pairs and maximize the distance between dissimilar pairs through a loss function.

The advantage of Siamese architecture is that it can learn powerful representations by leveraging shared weights and jointly optimizing the sub-networks. This makes it particularly useful when dealing with tasks that involve measuring similarity or finding patterns in pairs of inputs.

E. Capsule Network

One of the recent architectures suggested for iris recognition is Capsule Networks.

Capsule Networks, also known as CapsNets, are a novel and promising approach in the field of Deep Learning. They were introduced by Sara Saboor and her colleagues in 2017 as an alternative to traditional CNNs in order to overcome some of their limitations, such as their sensitivity to variations and differences in positions and rotations in the input images, inability to capture long-range dependencies and the lack of Spatial Hierarchies.

As its name indicates, CapsNets uses capsules instead of using only scalar outputs, i.e, instead of only considering neurons and the relationship between each two to represent specific characteristics of an object, we consider each group of neurons, that we call *a capsule*, and the connections between each group and another for identifying different features of the object we search in the image.

More explicitly, each layer of the architecture is composed of a set of capsules that each one represents a specific property of the object. These capsules capture not only the presence of features but also their spatial relationships and pose, providing richer information than individual neurons in CNNs. The overall architecture of the model is shown in the Figure 3.16 :

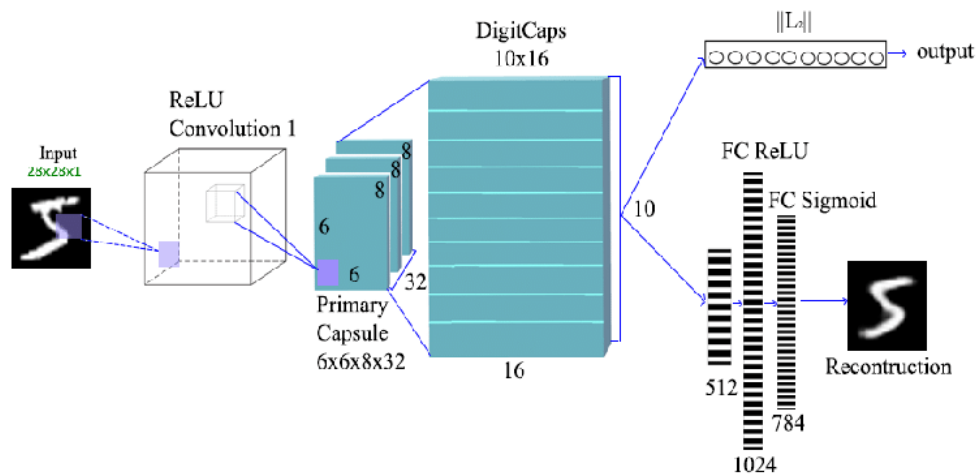


Figure 3.16 : CapsNet Architecture

Unlike CNNs, in CapsNets we find two types of weights :

1. the classical weights W relating each neuron of a layer i with the neurons of the layer $i-1$. As known, these weights are updated by the Backpropagation algorithm ;
2. the Routing Coefficients C relating each capsule of a layer i with the capsules of layer $i-1$. For the update of Routing weights, a new algorithm is introduced called the Dynamic Routing.

Dynamic routing refers to the iterative process of determining the routing coefficients between capsules during the forward propagation of data through the network. The routing weights indicate the strength or agreement between capsules, determining how much information is passed from one capsule to another.

in a simpler way, Dynamic routing ensures that each capsule contributes the best in the prediction of the output and helps the capsules of the next layer make accurate predictions as well. For example, if the capsule A in layer 2 contributes in the determination of the shape property which will be used next by capsule B in layer 3, then the coefficient between capsule A and B will be strengthened during training since caps A and B are in *agreement*.

This ensures that capsules work together effectively to improve the accuracy of predictions.

This new algorithm helps to establish meaningful connections between capsules and enables the network to learn hierarchical structures and pose relationships. By iteratively updating the routing coefficients, capsule networks can better capture complex variations in objects and improve their ability to recognize objects (the Iris in our case) from input data.

3.3.2 Hybrid Method (Features Fusion)

As part of our project's experimentation, we conducted tests on the Hybrid method. This approach involved providing the classifier with a combination of features extracted through a classical method and features extracted through a Deep Learning based method. Once we have built our main feature extractor based on DL which will be explained in details later on, we combine the features obtained from this architecture with those extracted using other classical techniques : : Wavelet Transform, Gabor filter, LPB, and GLCM. We will provide detailed explanations of these statistical methods.

It is necessary to mention that for our application, it is optimal to use texture descriptors since the normalized iris image represents the texture of the iris itself. Texture features in image analysis are derived from the observed groups of intensity in specific locations and their statistical distribution relative to each other [2], [7], [31].

These are the most used methods in the State of The Art for Texture Analysis :

A. Local Binary Patterns

Local Binary Patterns, also known as LBP, is a gray-scale texture descriptor measure for classification. The idea behind it is to generate, at each pixel, a binary pattern by thresholding its neighboring pixels to either 0 or 1 based on the value of the center pixel.

The steps to find the Local Binary Patterns of an image are the following :

- .1 Set a pixel value as a center pixel.
- .2 Collect its neighborhood pixels.
- .3 Threshold its neighborhood pixel value to 1 if its value is greater than or equal to center pixel value otherwise threshold it to 0. The Figure 3.17 illustrates this step :

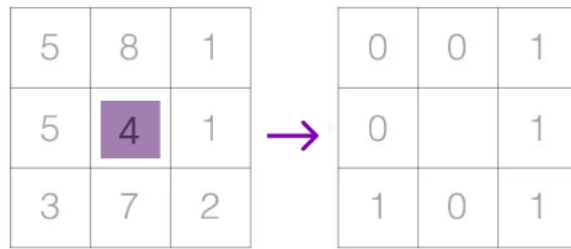


Figure 3.17 : Thresholding the center pixel neighbors

- .4 After thresholding, collect all threshold values from the neighborhood. The collection operation is shown in Figure 3.18 :

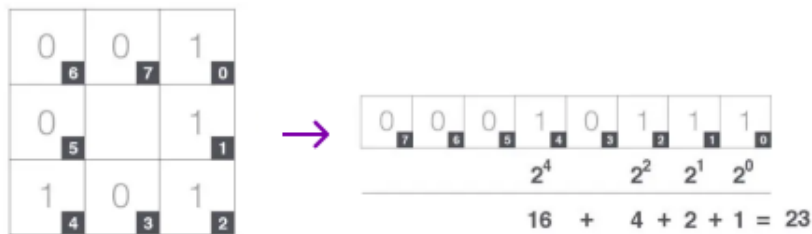


Figure 3.18 : Collecting the threshold values

- .5 Replace the center pixel value with resulted decimal (the previous collection will give you 8 digit binary code and convert it to the binary code into decimal). The Figure 3.19 shows this step :

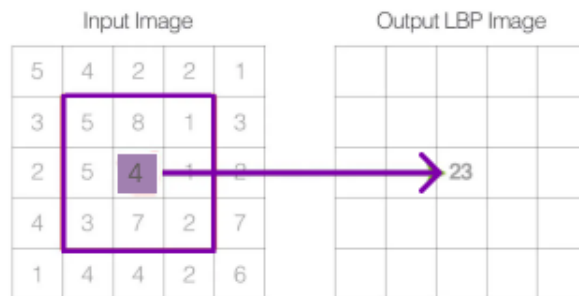


Figure 3.19 : Replace the center pixel's value by the threshold values from the neighborhood

- .6 Do the same process for all pixel values present in image.

Figure 3.20 is an example of the features extracted from an image using LBP :

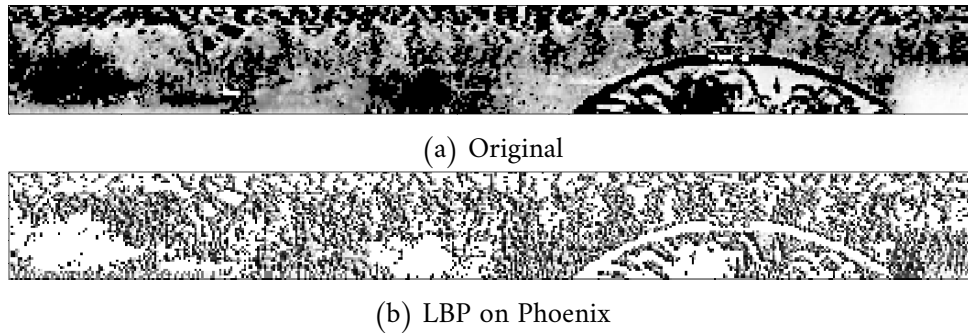


Figure 3.20 : Example of LBP on Pheonix's image

B. Grey Level Co-occurrence Matrices

The Grey Level Co-occurrence Matrices (GLCM) is among the initial techniques employed for Texture Feature Extraction. Over the years, it has gained significant popularity and widespread utilization in various texture analysis applications. It continues to be a vital method for extracting features in the field of texture analysis.

Given the significance of texture as a descriptor in our application, we recognized the need to test and implement GLCM in our project. By incorporating GLCM as a texture feature descriptor, we aim to extract valuable texture information that could further enhance our classification task. First phase of texture feature extraction is summarized as shown in the following :

- Find GLCM values for Four Angles (0° , 45° , 90° , and 135°). The process is described in Figure 3.21 :

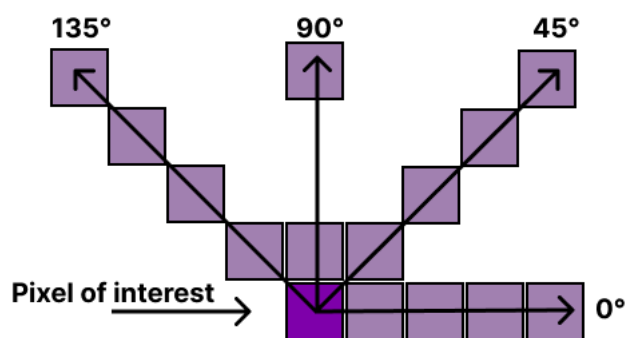


Figure 3.21 : GLCM's four angles

- Compute The four Features values, by Compute each feature (Eq. 1, 2, 3 and 4) with result of GLCM from step two, that we have four values (features) for each input medical image.
- Compute the feature values for each image.

C. Wavelet Transform

Wavelet analysis involves decomposing a signal or image into a set of wavelet coefficients through a process called Wavelet Transform. This transformation captures both local and global information of the signal or image, allowing for a detailed and flexible analysis.

Wavelet transforms can be classified into two broad classes 3.22 :

- The Continuous Wavelet Transform (CWT) : it can be used to analyze transient behavior, rapidly changing frequencies, and slowly varying behavior.
- The Discrete Wavelet Transform (DWT) : it is useful for compressing and denoising signals and images while preserving important features that's what motivated us to use it [47].

In the wavelet transform, a serie of Low and High Pass filters are applied on the image so that low and high frequencies will be separated as shown below in Figure :

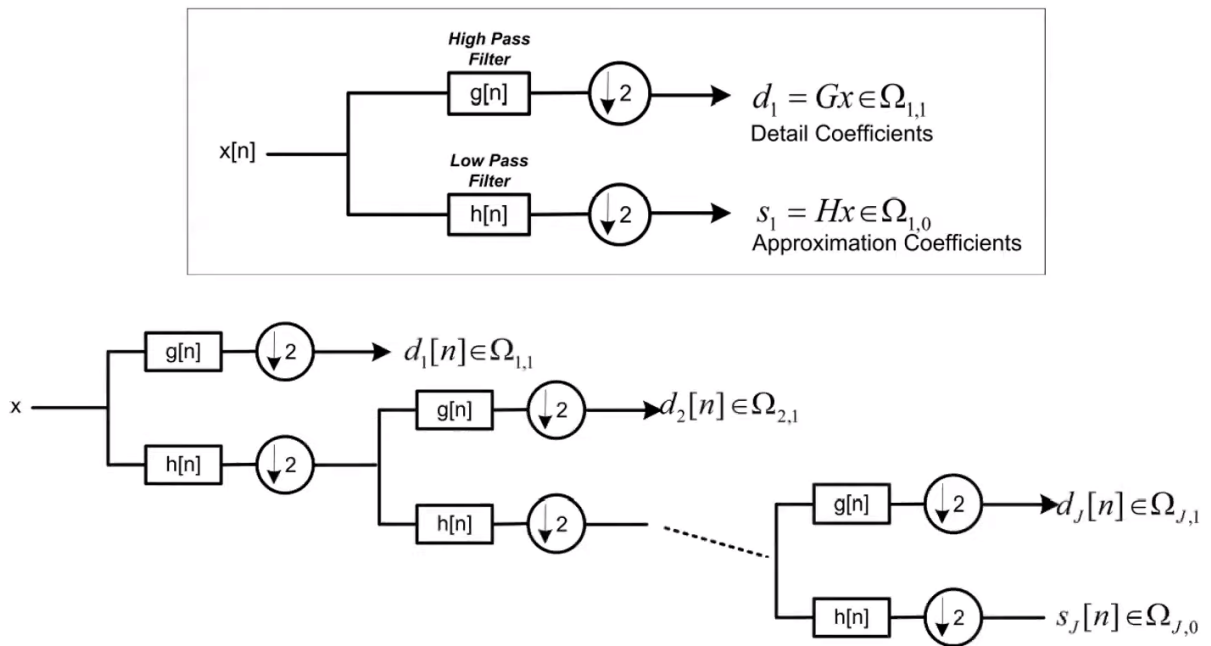


Figure 3.22 : Wavelets filters

Based on the spectrum of the images, we consider using the LL results, i.e, the result images after passing by two Low Pass filters. The LL results preserve the low frequencies that contain the useful information, and get rid off the high frequencies which are considered as noise.

Among Haar, Biorthogonal, Dubechee and Discrete Mayor Wavelet forms, Biothogonal Wavelet was, so far, the best one, the result of each one is shown in the Figure 3.23 :

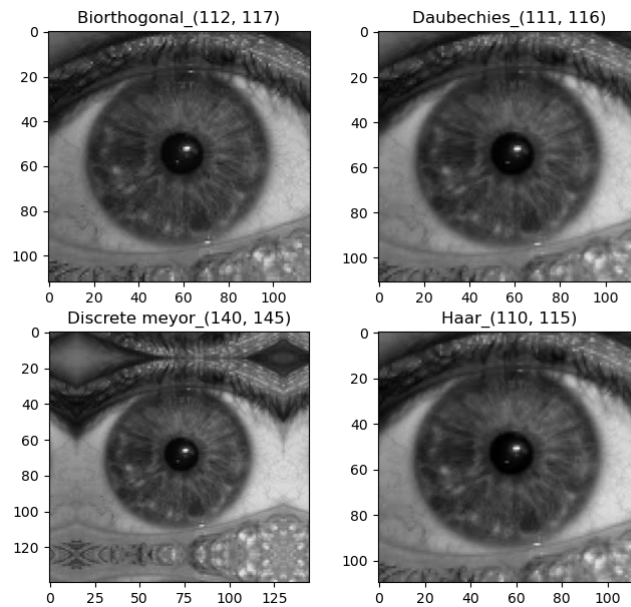


Figure 3.23 : Haar, Biorthogonal, Dubechee and Discrete Mayor wavelet app on iris image

In addition to features extraction, the Wavelet Transform offers the advantage of dimensionality reduction by generating characteristic images with half the dimensions of the input images. Given these advantages, we decided to experiment with the Discrete Wavelet Transform (DWT) with Biorthogonal form to extract the most significant features from the normalized images. We can see in the Figure 3.24 the output of the WT applied on Phoenix's image :

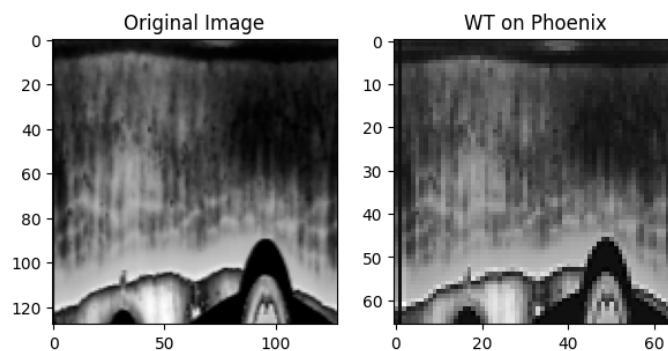


Figure 3.24 : Example of WT on Pheonix's image

By applying the DWT, we aim to identify and capture prominent features in the image. This transformation allowed us to analyze the image at multiple levels and extract relevant information in a more compact representation. The resulting wavelet coefficients provided valuable insights into the image's frequency content and variations, enabling us to effectively reduce the dimensionality of the feature space while preserving important image characteristics.

D. Gabor Filter

The Gabor filter is a classical method widely utilized for feature extraction and texture analysis. Given that our focus was on capturing the texture of the iris, the Gabor filter emerged as an intriguing choice.

The Gabor filter combines the concepts of frequency and orientation selectivity, making it particularly effective in capturing textural details in an image. It uses a set of sinusoidal functions modulated by a Gaussian window to analyze an image at different frequencies and orientations. By convolving the Gabor filter with the iris image, we can extract texture features that are sensitive to local variations and patterns. The Figure 3.25 illustrates the output of GB applied on a IITD's image :

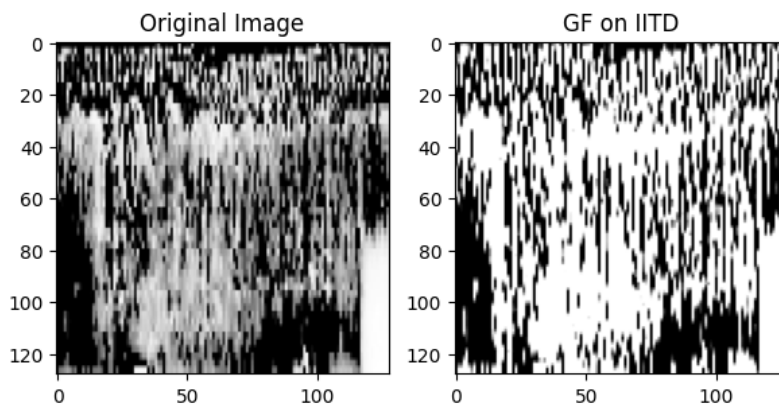


Figure 3.25 : Features of GF applied on IITD's image

In the context of iris texture analysis, the Gabor filter has demonstrated its efficacy in capturing intricate and unique texture information that can be used for iris recognition and classification tasks. Its ability to extract relevant features from iris images makes it a valuable tool in our pursuit of accurately representing and distinguishing different iris textures.

We experimented with several methods to accomplish this task of feature extraction in the chart in Figure 3.26 :

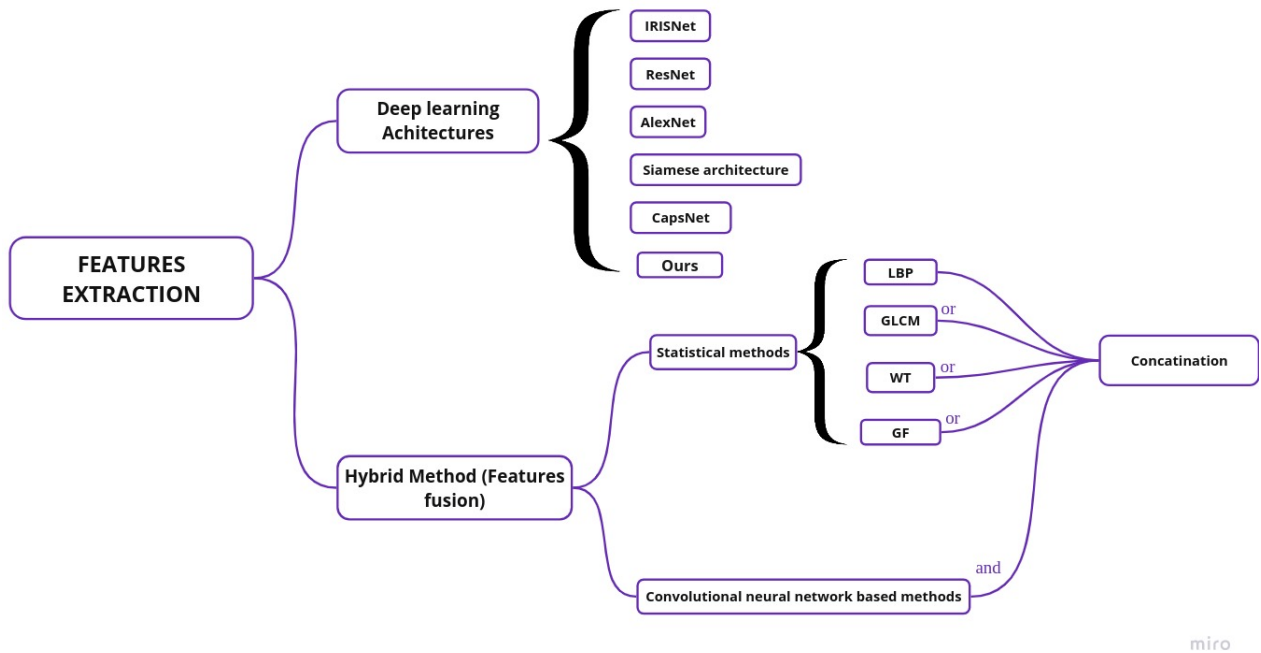


Figure 3.26 : Experimented Features Extraction Methods

3.3.3 Results of Feature Extraction methods

We showcase the result of each technique that is the number of features in Table 3.4 :

	Method	Features Number
DL based methods	IRISNet	2304
	AlexNet	43264
	ResNet	51210
	Siamese	512
	CapsNet	160
	Our architecture	384
Hybrid Method	Ours + LBP	25067
	Ours + GLCM	22510
	Ours + WT	160+90h
	Ours + GF	160 + 360.h

Table 3.4 : Results of Features Extraction Methods

Where h is the high of the input image which depends on the radius of the iris detected.

Up until now, both CapsNet and our suggested architecture stand out as the most efficient options in terms of resource utilization and computational requirements due to their minimal number of features comparing to the other architectures and methods.

However, solely considering the number of features is insufficient for determining which method to use. The optimal choice of a Features Extraction method can be made based on the

results obtained from the classification sub-stage. By evaluating these results, we can make an informed decision regarding the most suitable Features Extraction method.

3.4 Classification

After completing the Features Extraction step, the next sub-stage involves performing the matching process, which entails evaluating various types of classifiers. We are contemplating testing the following classifiers (Figure 3.27), taking into account their effectiveness in classification tasks.

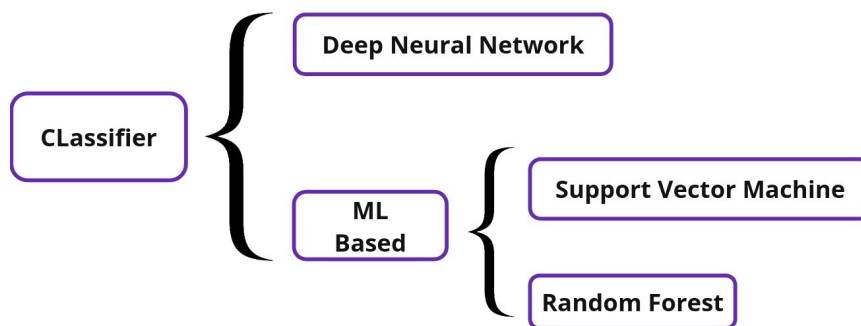


Figure 3.27 : Classifier - Explanatory Schema

3.4.1 Deep Neural Network Classifier

A Deep Learning classifier is a type of Machine Learning model that uses Deep Neural Networks to perform classification tasks. It is specifically designed to learn patterns and relationships in complex data by leveraging multiple layers of interconnected neurons.

The DNN used in this section as classifiers are mainly the ones defined with the architectures previously used in Features Extraction sub-stage :

- **IRISNet** : Two dense layers of 4096, 2048 neurons respectively, and finally softmax layer with a number of neurons depending on the number of classes that we have.
- **ResNet** : Unconstruct of other architectures , ResNet have only one dense layer of 1000 neurons.
- **AlexNet** : Two fully connected layers of each 4096 neurons and finally a Sotfmax layer with 1000 neurons.
- **Siamese** : One fully connected layer of 4096 neurons additionally to segmoid, L1 siamese distance.
- **CapsNet** : Two dense layers of 512, 1024 and Sigmoid layer of 784 neurons then determining the similarity between the input image of a certain class and the reconstructed image (output image) using the Margin Loss function.

Results of the DL-based classification

The table 3.5 shows the different results obtained when using the different DL-based architectures for Features Extraction and Classification, including the architecture that we proposed :

Architecture	Dataset	Accuracy (%)	Loss	Epochs
ResNet	IITD	62.21	1.0900	32
IRISNet	Phoenix	36.67	3.4500	32
AlexNet	Phoenix	88.81	1.0400	32
CapsNet	Phoenix	15.43	0.0080	20
Siamese Net	IITD	32.65	0.0021	32
Our Architecture	Phoenix	95.05	0.3100	32

Table 3.5 : Comparaision of Deep Learning based Methods

Among the various architectures we experimented with, our proposed architecture has demonstrated the highest accuracy thus far with a low loss.

For CapsNet, even if the loss is low but the accuracy is low as well, but this is not due to overfitting because the training accuracy is equal to 11.83.

3.4.2 Machine Learning Based Classifier

Due to the time, energy, and resource-intensive nature of DNN classifiers, it was deemed necessary to utilize alternative classifiers, such as SVM and Random Forest, which are comparatively less resource-consuming.

3.4.3 Support Vector Machine

The objective of this algorithm is to find a hyperplane in an n-dimensional space that separates the data points to their potential classes. The hyperplane should be positioned with the maximum distance to the data points. The data points with the minimum distance to the hyperplane are called Support Vectors.

The following formula 3.2 poses the optimization problem that is tackled by SVMs :

$$\min_{\omega, b, \zeta} \frac{1}{2} w^T w + C \sum_{i=1}^n \zeta_i y_i (w^T \phi(x_i) + b) \geq 1 - \zeta_i \tag{3.2}$$

where :

ζ_i denotes the distance to the correct margin with $\zeta_i \geq 0, i = 1, \dots, n$

C denotes a regularization parameter were $w^T w = \|w\|^2$ denotes the normal vector

$\phi(x_i)$ denotes the transformed input space vector

b denotes a bias parameter where y_i denotes the i -th target value

The objective is to correctly classify as many data point as possible by maximizing the margin from the Support Vectors to the hyperplane while minimizing the term $w^T w$ means finding the optimal w and b that most samples are predicted correctly.

When it comes to our case, we will use SVM for multi-class classification. It utilizes the same principle as binary classification. The multi-classes problem is broken down to multiple binary classification cases, which is also called one-vs-one.

The number of classifiers necessary for one-vs-one multi-class classification can be retrieved with the following formula 3.3 :

$$\frac{n * (n - 1)}{2} \quad (3.3)$$

With n being the number of classes.

In the one-vs-one approach, each classifier separates points of two different classes and comprising all one-vs-one classifiers leads to a multi-class classifier.

3.4.4 Random Forest Classifier

Random Forest is a highly popular algorithm in the field of Machine Learning due to its versatility in handling various types of problems, including classification tasks and other problem domains.

As its name suggests, it is comprised of numerous individual decision trees that work together as an ensemble. Each tree in the Random Forest produces a prediction for a specific class, and the prediction with the highest number of votes among the trees becomes the final prediction of the model [49].

Random Forest is constructed based on the fundamental concept of the "wisdom of crowds," which is both simple and powerful. This concept suggests that combining the predictions of multiple decision trees can yield more accurate and robust results compared to relying on a single tree. In a Random Forest, each decision tree contributes its unique knowledge and perspective to the overall prediction.

An essential aspect to consider is that each tree in the Random Forest is trained on a subset of the entire dataset, where it excels in making accurate predictions. This technique, known as Bootstrap, ensures that each tree focuses on specific subsets of the data. By aggregating the predictions through voting or averaging, the Random Forest leverages the collective wisdom of these individual trees. This process enhances the model's accuracy and its ability to generalize.

The diversity of opinions and perspectives among the trees plays a crucial role in minimizing biases and errors. Consequently, the Random Forest becomes a more reliable and effective model, benefiting from the combined knowledge and insights of its constituent trees [49].

Here is a explanatory example of the method (Figure 3.28) :

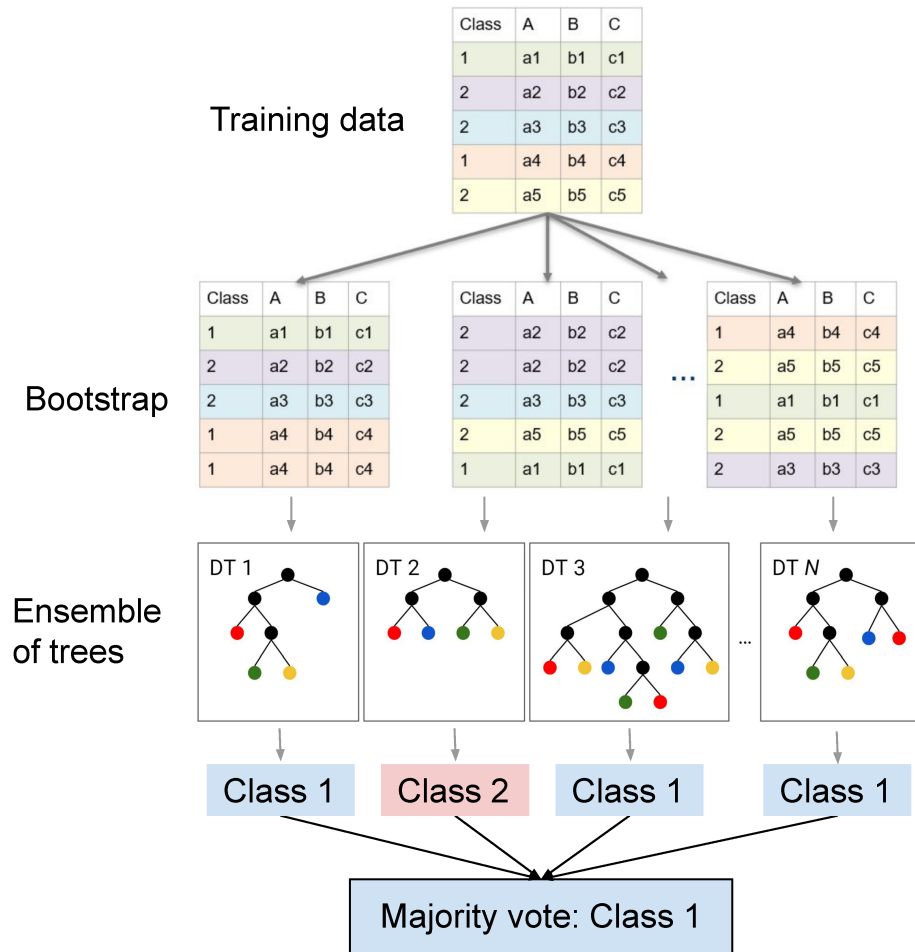


Figure 3.28 : Random Forest Algorithm

• **Note : Results of the ML-based classification**

Given that our proposed architecture achieved the best result when compared to other DNN-based architectures, and considering its lightweight nature, we made the decision to evaluate Machine Learning-based classifiers using its Features Extraction part. The outcomes of this combination will be presented in the following section.

3.5 Proposed Model

Within this section, we will present the conclusive model for Iris Features Extraction and Classification, alongside a detailed account of its training process.

3.5.1 Proposed Architecture

The structure of the proposed model is composed of :

- Three Convolution layers having : kernels of dimensions 11x11, 3x3 and 5x5, number

of kernels 96, 16 and 8, a padding of 0x0, 2x2 and 1x1 and a stride of 4x4, 1x1 and 1x1 respectively with ReLU as an activation function.

- Three Max Pooling layers of kernel 2x2 and a stride of 2x2, one after each convolution layer.
- One Flatten layer.
- Random Forest Classifier with a tree's depth of maximum of 30.

The following figure 3.29 gives a comprehensive and more representative idea about the architecture :

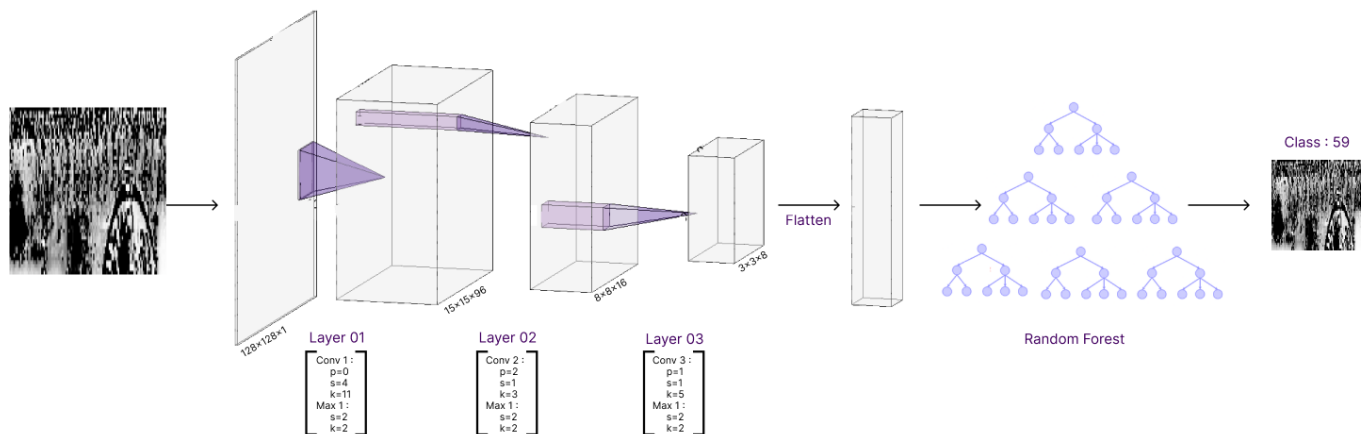


Figure 3.29 : Proposed Architecture

3.5.2 Training of the Architecture

It is important to know that the Features Extraction part that includes the Convolutions layers was trained using the following architecture :

- three Convolution layers having kernels of dimensions 11x11, 3x3 and 5x5, number of kernels 96, 16 and 8, a padding of 0, 2x2 and 1x1m and a stride of 4, 1 and 1 respectively with ReLU as activation function.
- Three Max Pooling layers of kernel 2x2 and a stride of 2, each one after each Convolution layer.
- One Flatten layer.
- Two Dense layers having 4096, 2048 neurons respectively with a Dropout of 50% applied during the training.
- Finally, a Softmax layer for classification with N neurons (N depends on the number of classes in the dataset).

The training process utilized the Back-propagation algorithm along with the Adam optimization technique. The Sparse Categorical Cross Entropy Loss function was employed as the training loss, and the training metric used was Accuracy.

To assess the performance of the Features Extraction Layers, the database is divided into two separate parts : the design set (D set) and the test set (T set). The design set is further

divided into training and validation subsets, allowing for model training and evaluation during the development phase. This separation ensures that the model is tested on unseen data, providing a reliable measure of its generalization performance.

Subsequently, for the training of the classifier, the generated data from the previously trained Features Extraction Layers were utilized as input for training the Random Forest Classifier.

Ultimately, the entire architecture is interconnected and dependent on each other.

3.6 Results of the proposed architecture

The different results we got when testing the different combination of the modified architecture of IRISNet, the four classical features extraction methods and the classifier are shown in the table bellow (Tab 3.6).

Discussion of the Results

- Initially, we observed improved performance across all four datasets by transitioning from a DNN classifier to a ML-based classifier. This improvement can be attributed to the well-documented effectiveness of SVM and Random Forest classifiers when provided with high-quality features.
- Furthermore, during the fusion of features, we observed lower performance when incorporating Local Binary Patterns (LBP), Gray-Level Co-occurrence Matrix (GLCM) and Gabor filter compared to other methods.
This can be attributed to the fact that although LBP and GLCM are texture descriptors, they do not perform well in capturing the intricate texture patterns present in iris images, which are more complex in nature.
- Concerning the Gabor filter, the selection of parameters for generating the Gabor filter kernel is not universally applicable and suitable for all images, even within the same dataset. This is primarily due to the variations in the spectral characteristics of different images. Consequently, using the same parameters across all images leads to suboptimal results, as the Gabor filter fails to effectively capture the desired features and textures present in each image.
- Moreover, when considering the performance of Wavelet Transform (WT), we observed that it generally yields satisfactory results, although its effectiveness varies depending on the dataset. Specifically, we noticed that WT does not produce favorable outcomes when applied to the IITD dataset. This can be attributed to the nature of the images in this dataset, which exhibit significant variations and high frequencies. As WT typically focuses on the low-frequency (LL) part of the image, it filters out the high-frequency components containing valuable information. Consequently, the loss of such crucial details leads to a decrease in model performance on the IITD dataset.
- We also observed that feeding a machine learning (ML) based classifier with a features vector of 4096 elements derived from the first Dense layer yielded better results compared to using two Dense layers or none. It appears that a single Dense layer was sufficient to refine the features extracted from the preceding layers. However, it is worth noting that the Dense layer contains a significantly larger number of parameters (approximately 8

million parameters). While we experienced only a marginal 1% loss in accuracy when using the three Convolutional layers (without the Dense layer), we deemed it more advantageous to omit the Dense layer due to the considerable computational resources required. This compromise allowed us to achieve a satisfactory level of accuracy while optimizing resource utilization.

- We note that we didn't explicitly mention the training accuracy because the test accuracy was sufficiently high, and we did not encounter any overfitting issues.

Finally, when taking into consideration all the previous discussed points, we can confidently state that the optimal architecture for our needs involves utilizing three Convolutional Layers for the Features Extraction sub-stage, combined with the Random Forest Classifier. This particular configuration yields the highest accuracy while minimizing resource consumption.

Note :

It is important to mention that all sub-stages of the system were programmed using Python Language, except the Data Augmentation where we used Matlab to generate augmented data.

3.7 Conclusion

In conclusion, we have provided a comprehensive overview of our proposed system, covering each sub-stage from the initial one, the Preprocessing sub-stage, to its final one, the Classification sub-stage. Additionally, we conducted thorough tests to refine this architecture and obtained notable results.

While our performance did not reach the level of the State of the Art, our results are quite satisfying. We achieved a commendable accuracy of 97.20% on the Phoenix dataset while optimizing resource utilization, particularly in terms of memory where we decreased by half the size of the model and in terms of computational requirements which are critical when moving to the implementation phase.

Proposed approach \Dataset			IITD %	MMU %	Phoenix %	Casia v-4 %	
IRISNet modified			90.11	74.05	86.88	61.54	
CNN	SVM	Two Dense Layers	None	78.79	84.28	93.76	69.80
			LBP	60.90	72.13	61.25	29.01
			WT	32.14	75.23	79.35	47.21
			GF	67.52	69.79	85.80	62.06
			GLCM	42.61	39.86	45.98	22.16
		One Dense Layer	None	84.93	67.85	79.57	64.07
			LBP	55.69	48.26	60.65	26.54
			WT	28.02	70.71	78.27	35.10
			GF	66.74	65.47	84.51	59.18
			GLCM	45.52	40.95	50.32	30.52
		None	None	82.03	60.95	67.74	49.88
			LBP	67.90	70.35	69.09	26.68
			WT	36.38	83.34	79.56	32.18
			GF	67.74	74.28	87.52	59.99
			GLCM	51.37	45.64	60.12	50.65
	Random Forest	Two Dense Layers	None	91.40	92.14	97.42	60.21
			LBP	90.06	89.99	94.89	30.82
			WT	89.50	91.58	97.42	40.63
			GF	89.73	93.50	97.20	65.18
			GLCM	52.60	50.99	54.80	47.46
One Dense Layer		None	91.07	95.15	96.56	66.06	
		LBP	89.77	89.02	92.34	30.20	
		WT	85.71	93.55	97.63	36.87	
		GF	86.72	90.88	96.34	63.05	
		GLCM	52.87	54.64	60.88	40.33	
None		None	93.42	91.58	96.34	51.03	
		LBP	62.50	81.32	76.99	65.85	
		WT	57.03	80.16	90.32	60.00	
		GF	59.26	55.64	92.68	70.07	
		GLCM	56.23	50.80	59.04	34.69	

Table 3.6 : Results of the different tests using the proposed architecture

Chapter 4

Implementation

4.1 Introduction

After successfully evaluating the model with high accuracy in Chapter 3, the next step is to implement the model on an FPGA. In this chapter, we will focus on optimizing the model to reduce resource usage, followed by implementing it with two different approaches and comparing their performance.

The first task is to optimize the model to minimize the utilization of FPGA resources. This involves analyzing the model architecture, identifying redundant operations or parameters, and applying techniques such as pruning and quantization. By reducing the model's size and complexity, we can effectively decrease the resource requirements while maintaining satisfactory performance.

Next, we will proceed with the implementation of the optimized model using two different approaches. The first approach involves utilizing HLS, a high-level design methodology that allows us to describe the model using a higher-level programming language. With HLS, we will be able to specify the model's functionality and optimizations at a more abstract level, enabling automated synthesis of the hardware implementation. By leveraging HLS, we will be able to take advantage of its optimizations to reduce resource utilization and enhance performance.

The second approach involves utilizing the MATLAB Deep Learning Toolbox, which provides a comprehensive set of functions and tools for developing and deploying deep learning models. Once both implementations are ready, we will thoroughly evaluate their performance and compare the results. Metrics such as resource utilization, execution time, power consumption, and accuracy will be considered. The goal is to identify the strengths and weaknesses of each approach and determine the most suitable implementation for the specific requirements and constraints of the project.

By implementing the model on an FPGA using HLS and the MATLAB Deep Learning Toolbox, this chapter aims to provide insights into the trade-offs between resource utilization, performance, and accuracy. The findings will guide the selection of the optimal implementation strategy, ensuring efficient utilization of FPGA resources while maintaining reliable and high-performance operation.

4.2 Optimization of the model : Pruning and Quantization

Algorithmic optimization strategies used in hardware architectures for Convolutional Neural Networks (CNNs) focus on improving the efficiency and performance of CNN computations on specialized hardware platforms. These strategies aim to exploit the inherent parallelism and spatial locality present in CNNs to accelerate the computation and reduce power consumption. Network Pruning and Compression, Quantization, Data Reuse, Memory Optimization, Parallelism and Pipelining are some commonly employed algorithmic optimization strategies in CNN hardware architectures [21].

These algorithmic optimization strategies are often combined with architectural optimizations, such as custom memory hierarchies, specialized computation units, and efficient interconnect architectures, to further enhance the performance and efficiency of CNN

hardware implementations. By leveraging these techniques, CNN hardware architectures can achieve high throughput, low power consumption, and real-time processing capabilities, making them well-suited for various applications like image recognition [21].

In our project, we decided to employ two optimization techniques, namely pruning and quantization. We chose these techniques based on our assessment that they would be adequate for implementing the proposed architecture on the FPGA. Since our architecture is not excessively deep or complex, we concluded that pruning and quantization would provide sufficient optimization to achieve our desired implementation goals. By leveraging these techniques, we aim to reduce the computational complexity, memory requirements, and overall resource utilization of our architecture, while still maintaining acceptable performance and accuracy levels.

4.2.1 Pruning

Since the Neural Network consumes considerable resources, especially memory, multiple pruning techniques can be applied in order to compress the network by reducing its complexity. There are two main pruning categories : Weights Pruning and Neurons Pruning.

- **Weights pruning** consists in eliminating the less relevant weights i.e. the weights that do not contribute enormously in the network. For this, multiple techniques are introduced, such as Low Magnitude Based Pruning and Low Gradient Based Pruning through which we remove weights having magnitude and gradient respectively, smaller than a determined threshold [52].
- **Neuron pruning** removes entire neurons or layers in the network, it can be achieved by Low-Density Pruning which enable the removal of neurons with low activity, and Filter Pruning which involves eliminating filters with low importance [52].

The next Figure (4.1) shows the difference between Weights Pruning and Neurons Pruning :

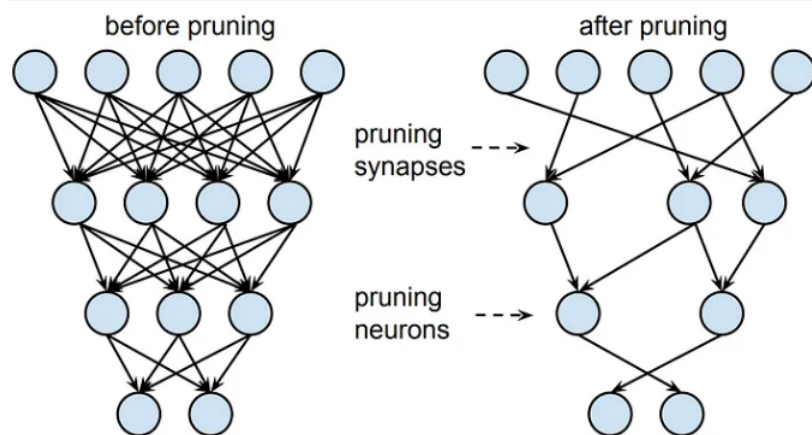


Figure 4.1 : Visualization of pruning weights/synapses vs nodes/neurons

4.2.2 Quantization

Model quantization is a technique used to reduce the memory footprint and computational requirements of deep learning models. By quantizing a model, the precision of its parameters is reduced. This compression technique allows for efficient storage and faster inference on hardware with limited resources, such as FPGAs. Quantization methods include *post-training quantization*, where a pre-trained model is quantized, and *quantization-aware training*, which incorporates quantization considerations during the training process.

In our scenario, we employed a post-training quantization technique to decrease the size of the model's Weights and Biases. Specifically, we reduced their precision from float-32 to float-16. Given that the model was constructed and trained using TensorFlow, we leveraged TensorFlow Lite quantization. This framework provides an efficient solution for deploying Machine and Deep Learning models on devices with limited resources like FPGAs. It encompasses a range of techniques aimed at reducing model size and enhancing inference speed while ensuring acceptable levels of accuracy. TensorFlow Lite supports multiple quantization approaches, including post-training quantization.

4.2.3 Model Optimization Results

The results of the optimization techniques, namely Pruning (Prun.) and Quantization (Quant.), including : Accuracy, Loss, Model Size and Compression Ratio (Comp. Ratio), are shown in the Table 4.1 :

	Prun. Ratio	Accuracy (%)		Loss		Size(MB)		Compr. Ratio
		Before	After	Before	After	Before	After	
Prun.	0.2	95.05	97.20	0.31	0.15	33.79	33.79	1
	0.5	95.05	94.62	0.31	0.22			
Quant.	0.2	97.204304	97.204301	-	-	33.79	16.88	2
	0.5	94.6236551	94.6236559					

Table 4.1 : Results of Pruning and Quantization of the model

- While practically pruning reduces computations, it may affect the performances of the model (accuracy in our case) either positively or negatively. In our case, when applying a Low Magnitude Pruning on the network with 20% of pruned weights, the performance rises from 95.05% to 97.20%, but when applying Pruning with 50% ratio for pruned weights, the performance decreases to 94.62 but the loss gets better. We can notice that the size of the model doesn't change when applying pruning, this is due to the nature of pruning used in Tensorflow, where the weights are not deleted or eliminated, but just put to 0 value.

- When we apply Quantization to the model, which typically leads to a decrease in accuracy, we observed that the accuracy decreases by approximately $2.10^{-6}\%$ when the pruning ratio is 20%, and it increases by approximately $9.10^{-7}\%$ when the pruning ratio is 50%. These results are excellent for us because our application does not require high precision. Since the quantization did not degrade the accuracy beyond a 5% threshold, it is considered suitable for our needs.

4.3 High Level Synthesis Approach

The first approach we consider to use is the High Level Synthesis approach which is used when the description of the Hardware architecture is quiet challenging to realize with VHDL directly.

HLS is a design methodology that enables the transformation of high-level programming languages (such as C, C++, or SystemC) into hardware descriptions. HLS tools analyze the behavior and algorithms described in high-level languages and automatically generate optimized hardware designs. This approach allows us to focus on algorithmic development and productivity, while still achieving efficient and high-performance hardware implementations.

The approach consists of seven main steps :

- Programming the function or the subsystem we want to implement on the FPGA in a High Level Language (C in our case),
- Running the simulation to ensure that the program written before works well,
- Synthesize, exporting in RTL and generating the Bitstream of the previous function. This step allows us to generate an IP Block for the function needed,
- Creating a Block Design for the main system that will be deployed on the FPGA and includes the IP Block generated previously,
- Validating the Block Design, creating then an HDL Wrapper which converts the schema of the architecture into an HDL program,
- Running the simulation in order to ensure that the architecture is working well,
- Synthesizing the project, exporting in RTL and then running the implementation and generating the Bitstream file,
- Finally, programming the device.

We note that some additional steps may be needed according to the FPGA and the peripherals used in the system. The Figure 4.2 represents a simplified synoptic schema of the approach :

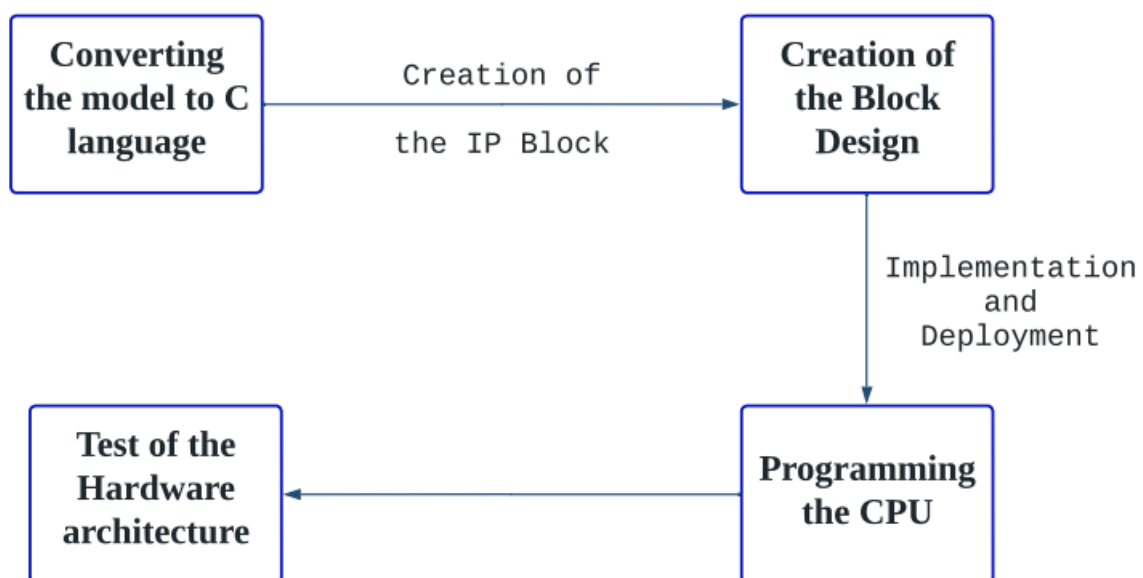


Figure 4.2 : Simplified Synoptic Schema of the HLS Approach

4.3.1 Subsystem in C language

Considering the computationally intensive nature of Convolutional Neural Networks, we have identified the Features Extraction substage, encompassing the Convolution Layers, as a critical component in our system. To address this, we have opted to implement this substage on the FPGA and the rest of the sub-stages of the system on a CPU.

Since the Convolution Layers have already been trained off-chip and we possessed the respective layer weights and biases, we have chosen to implement this sub-stage as a *set of functions* that work with each other to *emulate* a pre-trained model. By implementing it as a set of functions rather than a complete model, we can optimize resource utilization and achieve significant savings in time and energy consumption.

Totally, there are three main functions, each one represents one layer of the Features Extraction sub-stage. All the functions are written in C language. Here are the algorithms used for each functions :

- **Padding Layer function :**

1. Create a sub-function that computes the padding of one image. This sub-function receives as input the input image and the number of padding and gives as output the padded image.
2. Create the top function that computes the padding of multiple images, using the sub-function previously created.

- **Convolutional Layer function :**

1. Create a sub-function that computes the convolution of one image. This sub-function receives as input the input image, the weights of the kernel and its bias as well as some additional variables, like stride and dimension of the kernel.
2. Create a function that computes the convolution of one image with multiple kernels, using the sub-function previously created. It takes the weights and bias of each kernel, and the stride and kernels dimensions as attributes.
3. Create the top function that computes the convolution of multiple images with multiple kernels, using the sub-function previously created as well.

- **Maxpooling Layer function :**

1. Create a sub-function that computes the Maxpooling of one image. This sub-function receives as input the input image and some additional variables, like stride and dimension of the kernel.
2. Create a function that computes the Maxpooling of one image with multiple kernels, using the sub-function previously created. It takes the dimension of the kernels and the stride and kernels dimension as attributes.
3. Create the top function that computes the convolution of multiple images with multiple kernels, using the sub-function previously created as well.

Once the functions of the generalist layers are written, we use them to generate the layers we need in our architecture. Here's is an example of the function generated for the third Convolutional Layer (Figure 4.3) :

```
1 #include <string.h>
2 #define IMG_DIM (8+2)*(8+2)*16
3 #define W3 200
4 #define OUTPUT_DIM 6*6*8
5 #define B3 8
6
7 void Relu(int x)
8 {if(x<0){x=0;}}
9
10 void Conv2D (int *image, int *new_image, char stride, char img_dim, char kernel, short *weights, short bias, int final_dim,
11 int start)
12 {
13 }
14
15 void ConvOneChannel(int *input, int *output, char stride, char img_dim, char nbr_kernels, short *total_weights, short *biases,
16 char kernel, int final_dim)
17 {
18 }
19
20 void ConvManyChannel(int *input, int *output, int output_dim, int stride, int img_dim, int nbr_kernels, int *total_weights,
21 short *biases, int kernel, int nbr_input_chan)
22 {
23 }
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67 int CONV3 (int input[IMG_DIM],
68           short weights_l3[W3],
69           short biases3[B3],
70           int output[OUTPUT_DIM])
71 {
72     // Intermediate Outputs
73     int output_l7[6*6*8]=0;
74     ConvManyChannel(input, output_l7, 6, 1, 8*2, 8, weights_l3, biases3, 5, 16);
75     return 0;
76 }
```

Figure 4.3 : C code of CONV3

Note :

Initially, we developed a single function encompassing the entire architecture, incorporating the previously introduced functions as sub-functions within the main function, representing the entire sub-stage. Our objective was to create a single IP Block for the Features Extraction sub-stage. However, due to limitations in both the software and hardware resources, we decided to construct the system using separate blocks, with each block representing a specific layer.

4.3.2 Vitis HLS

The High-Level Synthesis consists of converting the program from a High Level Language into an HDL program that can be synthesized and implemented in an FPGA.

The Vitis HLS Software is utilized to perform the High-Level Synthesis for all the functions within the Features Extraction sub-stage. We present here the interface of the software with same example of the previous function (Figure 4.4) :

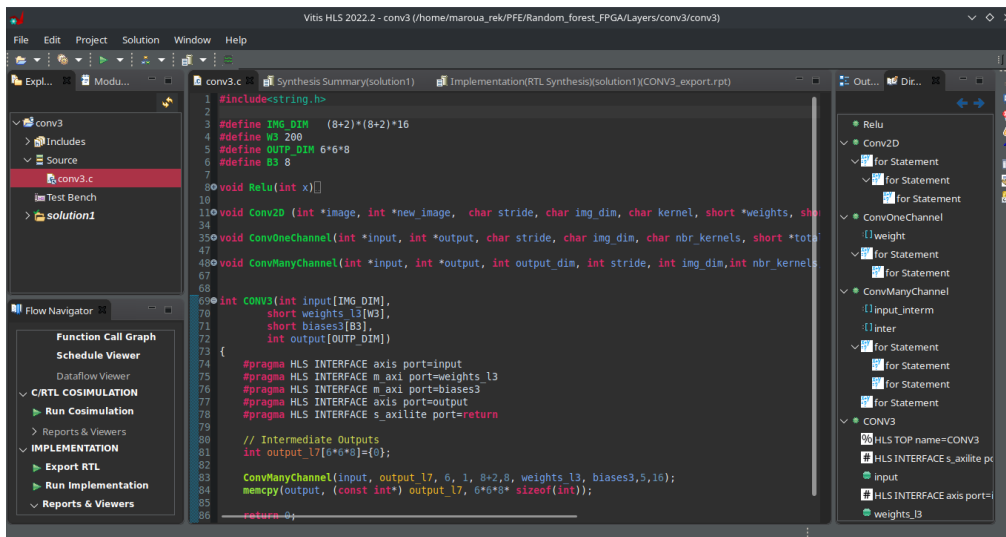


Figure 4.4 : Vitis HLS Interface

The simulation with Vitis HLS of all the layers is necessary, here's a result example when conducting a TestBench on the zero padding layer (Figure 4.5) :

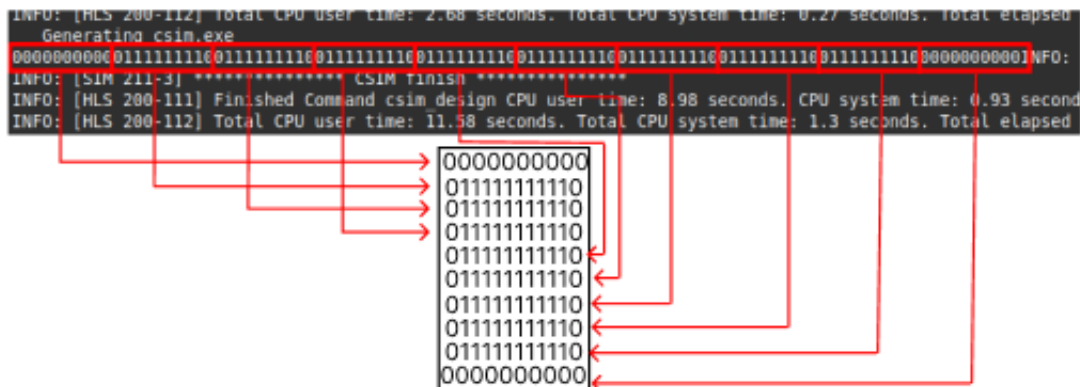
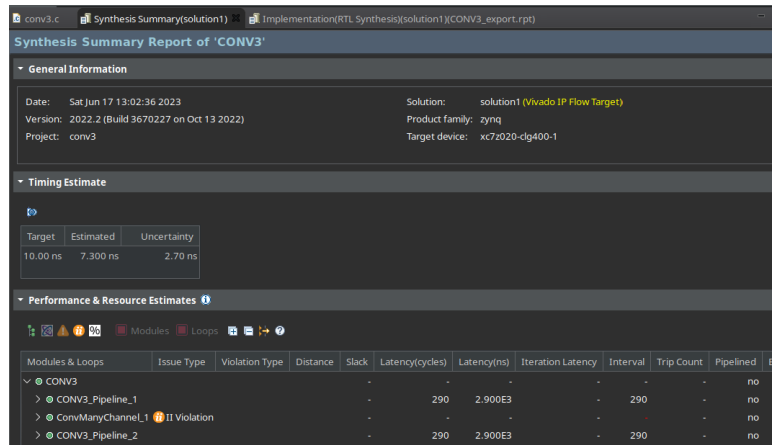
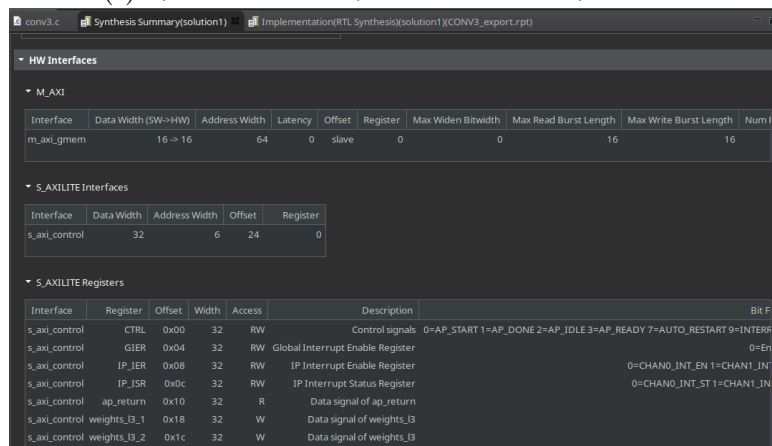


Figure 4.5 : Padding of an image having 5-by-5 dimension (pad=1)

When running the C Synthesis, we obtain a Synthesis Summary in which numerous information are given such as Timing, Performances and Resources estimation. The Figure 4.6 shows an example of Synthesis summary of the CONV3 layer :



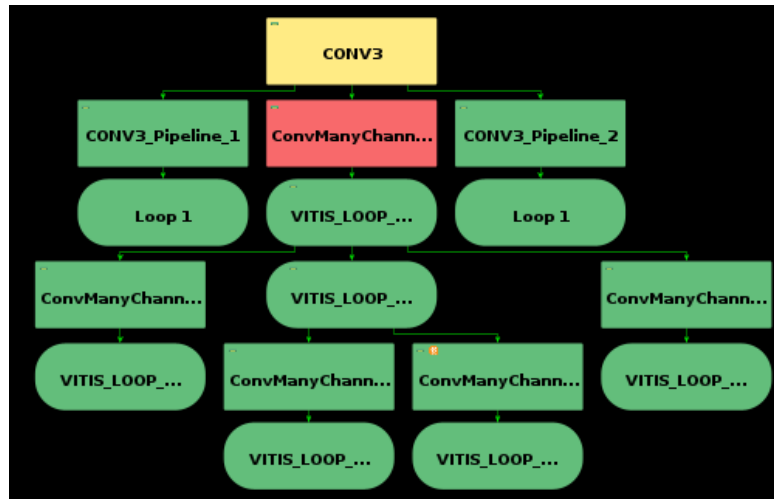
(a) Synthesis summary of the CONV3 layer Part 1



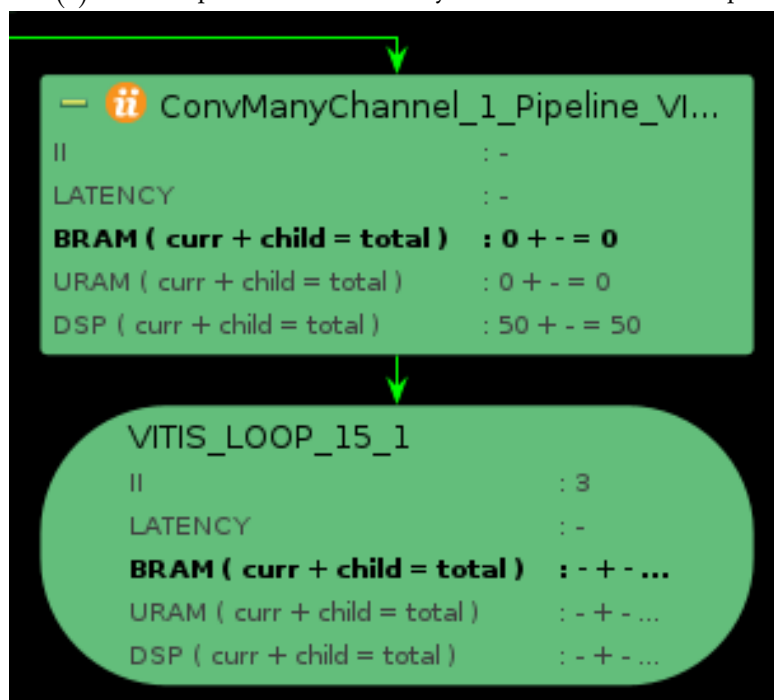
(b) Synthesis summary of the CONV3 layer Part 2

Figure 4.6 : Synthesis summary of the CONV3 layer

We can also see the Function Call Graph with the option of a Heat Map of a Specific Metric such as Block RAM (BRAM) and Latency. It is useful in visualizing the different subroutines of the program and their pipeline. Here is an example of the Function Call Graph of the CONV3 layer with BRAM Heat Map (Figure 4.7) :



(a) Call Graph of the CONV3 layer with BRAM Heat Map



(b) Call Graph of the CONV3 layer with BRAM Heat Map with Zoom on a Block

Figure 4.7 : Call graph of the CONV3

Once the synthesis is complete, we run the Register Transfer Level (RTL) export and then the Implementation. The RTL code specifies the logic gates, registers, and interconnections that comprise the digital circuit. During the implementation stage, the RTL code generated is further refined and optimized for the target hardware platform. This includes various transformations and optimizations to improve the design's performance, area utilization, power consumption, and timing. The Figure 4.8 shows the Implementation Summary of the CONV1 layer where we have the actual information like final resources that will be used for this block :

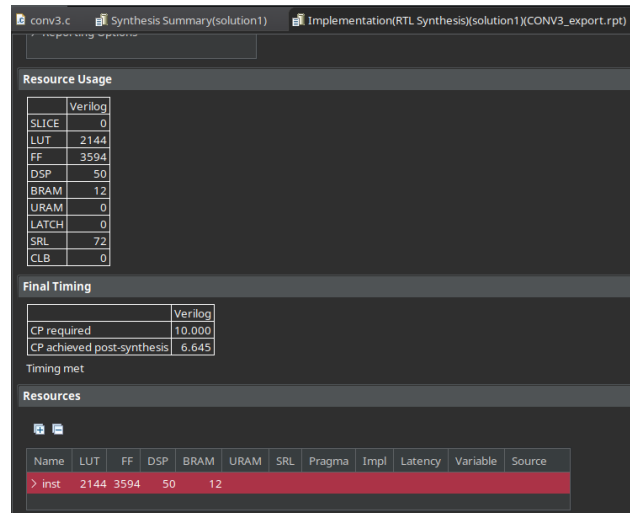


Figure 4.8 : Summary of RTL export and implementation

Once all the synthesis and the RTL export of all the functions we have created is done, we move to the creation of the Block Design of the total architecture.

4.3.3 IP Block Interfacing - Data Transfer

In anticipation of the implementation of the Block Design, it is crucial to familiarize ourselves with certain concepts that will greatly facilitate the comprehension of the next steps. These concepts include Block interfacing and data transfer, which will be explored to ensure a comprehensive understanding of the upcoming explanations.

We observe that in the previous code of the CONV3 layer depicted in Figure 4.4, when utilizing HLS, there are a few additional instructions within the main function (CONV3). These instructions, denoted by `#pragma HLS INTERFACE` which are shown in Figure 4.9, serve as interfacing directives that specify the data transfer protocol utilized by the input or output ports of generated IP Core.

```
#pragma HLS INTERFACE axis port=input
#pragma HLS INTERFACE m_axi port=weights_l3
#pragma HLS INTERFACE m_axi port=biases3
#pragma HLS INTERFACE axis port=output
#pragma HLS INTERFACE s_axilite port=return
```

Figure 4.9 : Interfacing Directives for CONV3

The INTERFACE pragma or directive specifies how RTL ports are created from the function arguments during interface synthesis. The Vitis HLS tool automatically determines the I/O protocol used by any sub-functions. Each function argument can be specified to have its own I/O protocol [57].

The port protocol establishes the mechanisms for managing data flow within the data channel, determining when the data is valid and can be read or can be written. It is determined by the manner in which data is transferred between blocks and its origin.

The protocol used by many SoC today is AXI, or Advanced eXtensible Interface. It is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) specification. It is especially prevalent in Xilinx's Zynq devices, providing the interface between the processing system and programmable logic sections of the chip [58].

AXI Protocol

The protocol establishes a set of rules governing the communication between different modules within a chip. It mandates a handshake-like procedure prior to any transmission, ensuring a coordinated exchange of data. This protocol facilitates the creation of a comprehensive "system" rather than a mere collection of modules, as it serves as an effective medium for data transfer between the chip's components[58].

The specifications of the protocol can be summarized as follows :

- Prior to transmitting any control signal, address, or data, both the master and slave modules engage in a handshake using ready and valid signals.
- The transmission of control signals and addresses occurs in separate phases, each with its dedicated channel.
- Data transmission also utilizes a separate channel distinct from control signals and addresses.
- The protocol supports burst-type communication, enabling continuous data transfer [58].

The Figure 4.10 [55] represents the channel connections of the AXI protocol :

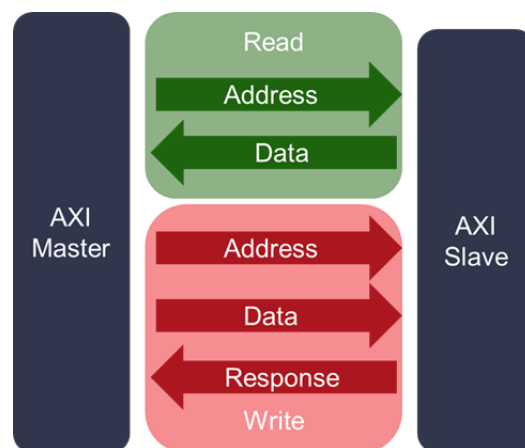


Figure 4.10 : Channel Connections between Master and Slave Interfaces

In the AXI protocol, a channel refers to a distinct set of AXI signals that includes the VALID and READY signals. Each channel operates independently. When both the VALID and READY signals are high, and there is a rising edge of the clock, a piece of data is transmitted on that specific channel. This data transmission event is referred to as a transfer [55].

For instance, in the given Figure 4.11 [55], the transfer is occurring at T3, indicating that on that specific clock cycle, both the VALID and READY signals are active, allowing the data to be successfully transmitted on the channel :

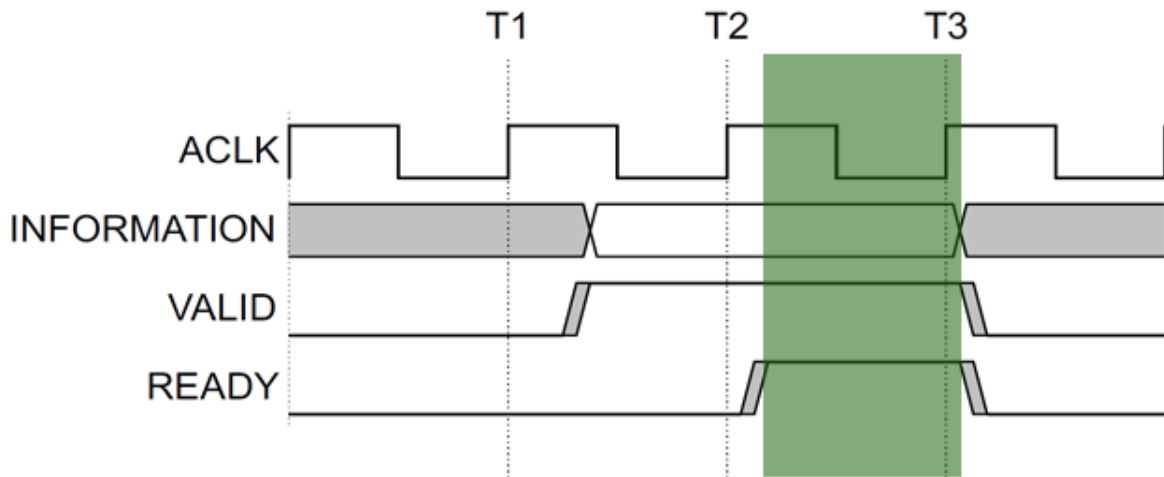


Figure 4.11 : Valid Transfer of Data

The AXI protocol encompasses three main types : AXI Memory Mapped, AXI Stream, and AXI Lite. Each type serves specific purposes and is designed to address different communication requirements.

- **AXI Memory Mapped :**

The AXI Memory Mapped protocol is used for communication between a master and a slave device in a memory-mapped system. This type of communication is based on addressing specific memory locations for read and write operations. The protocol supports burst transfers, allowing for efficient data transfer across multiple memory locations. It utilizes separate channels for read address, write address, read data, write data, and write response. The protocol incorporates handshaking using VALID and READY signals to ensure synchronized data transfers.

- **AXI Stream :**

The AXI Stream protocol is intended for high-speed, unidirectional data streaming applications. It is commonly used for transferring large amounts of data between modules or processing elements. Unlike the memory-mapped protocol, AXI Stream does not involve explicit addressing of memory locations. Instead, it relies on a continuous stream of data without separate channels for address or response. The communication occurs via a single channel using a flow control mechanism based on READY and VALID signals. Data transfers happen continuously as long as both the sender (VALID signal) and receiver (READY signal) are ready for the next transfer.

- **AXI Lite :**

The AXI Lite protocol is a simplified version of the AXI Memory Mapped protocol. It is specifically designed for low-bandwidth applications with reduced resource requirements. AXI Lite supports basic read and write operations to memory-mapped registers. AXI Lite uses a single address channel for both read and write operations, along with separate

channels for read data and write response, reducing this way the number of signals and complexity.

As previously stated, the selection of the protocol type depends on the nature of the data to be transmitted as well. The following table (4.2) illustrates the recommended protocol type for different variable types :

C-argument type	Paradigm	Interface protocol (I/O/Inout)
Scalar (pass by value)	Register	AXI4-Lite (s_axilite)
Array	Memory	AXI4 Memory Mapped (m_axi)
Pointer to array	Memory	m_axi
Pointer to scalar	Register	s_axilite
Reference	Register	s_axilite
hls::stream	Stream	AXI4-Stream (axis)

Table 4.2 : AXI protocol according to the variables types

Overall, the AXI protocol provides flexibility and scalability by offering these three types, enabling efficient and reliable data communication across various system architectures and application requirements.

Protocol Type Selection

Since the Features Extraction system follows a data flow architecture in which each IP Block within the system receives data from a preceding IP block, performs specific computations such as convolutions, max pooling, or padding on the received data and subsequently, the output data is transferred to the next IP block ; and given that the data being transferred between IP blocks is in the form of arrays, the primary protocol utilized for this data transfer is AXI-Stream, denoted as "axis".

We also use the AXI-Memory Mapped protocol denoted "m_axi" in the Convolution Blocks where besides receiving and sending data to the previous and next blocks respectively, we read the Weights needed that are stored in the Memory Block. Additionally to the Convolution Blocks, we use the m_axi in the blocks that communicate with the Memory Block which are the first and the last blocks in the chain : the first one (CONV1) reads the input image from the memory and the last block (MAX3) write its results in the memory as well.

The s_axilite protocol is used in all the blocks for control signals and to generate an interrupt that can be used if needed.

In summary, Figure 4.12 illustrates the data flow diagram of the system, along with the corresponding protocols employed :

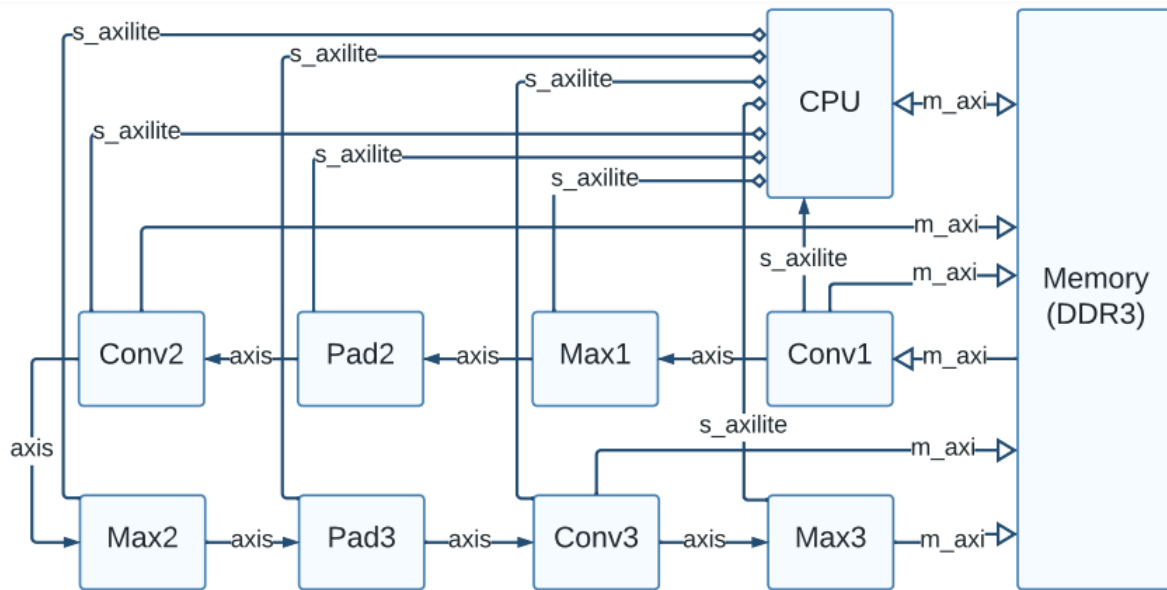


Figure 4.12 : Data Flow of the Features Extraction Sub-stage

4.3.4 Block Design of the Architecture

Block Design Methodology

In FPGA design, a block design refers to a modular approach of designing digital circuits using pre-defined blocks or modules. These blocks are often referred to as Intellectual Property (IP) cores and can be either provided by the FPGA manufacturer or created by the designer.

In a block design, the overall circuit is divided into functional blocks, where each block represents a specific functionality or subsystem of the design. These blocks can include components such as processors, memory controllers, interfaces, mathematical operations, or custom logic circuits.

The blocks are interconnected using standardized interfaces, such as buses or point-to-point connections, enabling communication and data flow between the different blocks. This modular approach offers several advantages, including ease of design, reusability, and efficient utilization of FPGA resources.

Block designs can be created through graphical design tools provided by FPGA development environments such as *Vivado Design Suits*. These tools allow us to visually arrange and connect the blocks, configure their parameters, and generate the necessary HDL code or configuration files required to program the FPGA.

Overall, block designs simplify the FPGA design process by promoting modularity, reusability, and ease of integration, ultimately leading to efficient and scalable FPGA implementations.

Creation of the Block Design of Our System

For our system, once we synthesized and generated all the functions of the different layers using Vitis HLS, we move to the creation of the Block Design for our system.

For this purpose, we follow the next steps :

- First, we **create a new project** for our system in Vivado Design Suit, specifying in it the FPGA (SoC or Board) that we will use. Once the project is created we will see this interface (Figure 4.13) :

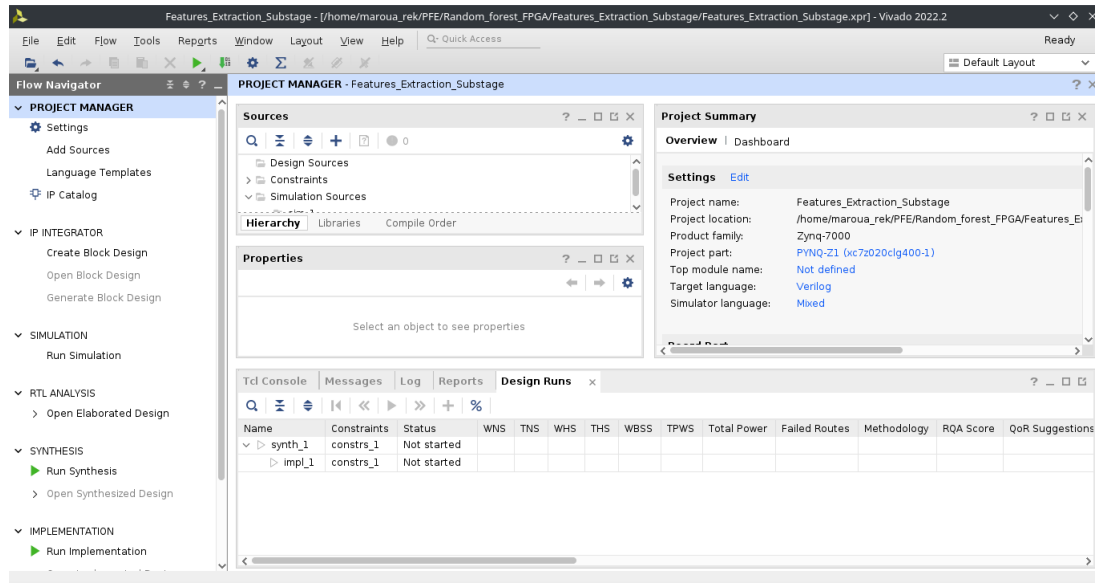


Figure 4.13 : Vivado Interface

- Second, we **add all the repositories of the different IP Blocks** previously created with Vitis HLS to the "IP Repositories" in "Project Manager Settings" (Figure 4.14) :

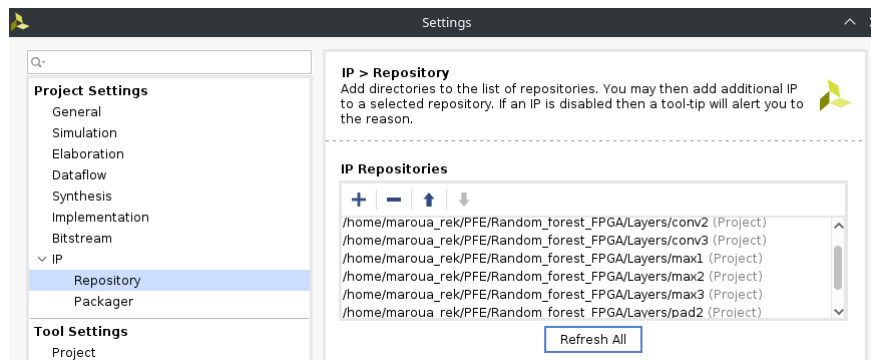


Figure 4.14 : IP Repositories

- After this, we **create a new Block Design** for our project and we add all the necessary blocks which are : the IP blocks of the different layers, and the Processor that will control the Memory Block and contain the rest of the sub-stages of the system, as follows (Figure 4.15) :

We can notice that there are a DDR and a FIXED_IO outputs of the processor, they were automatically generated by Vivado when running the *Block Automation* of the Zynq processor. We will provide in the next section a detailed description of the FPGA used and its specification but the important thing to know for now is that the Zynq IP Core is our processor.

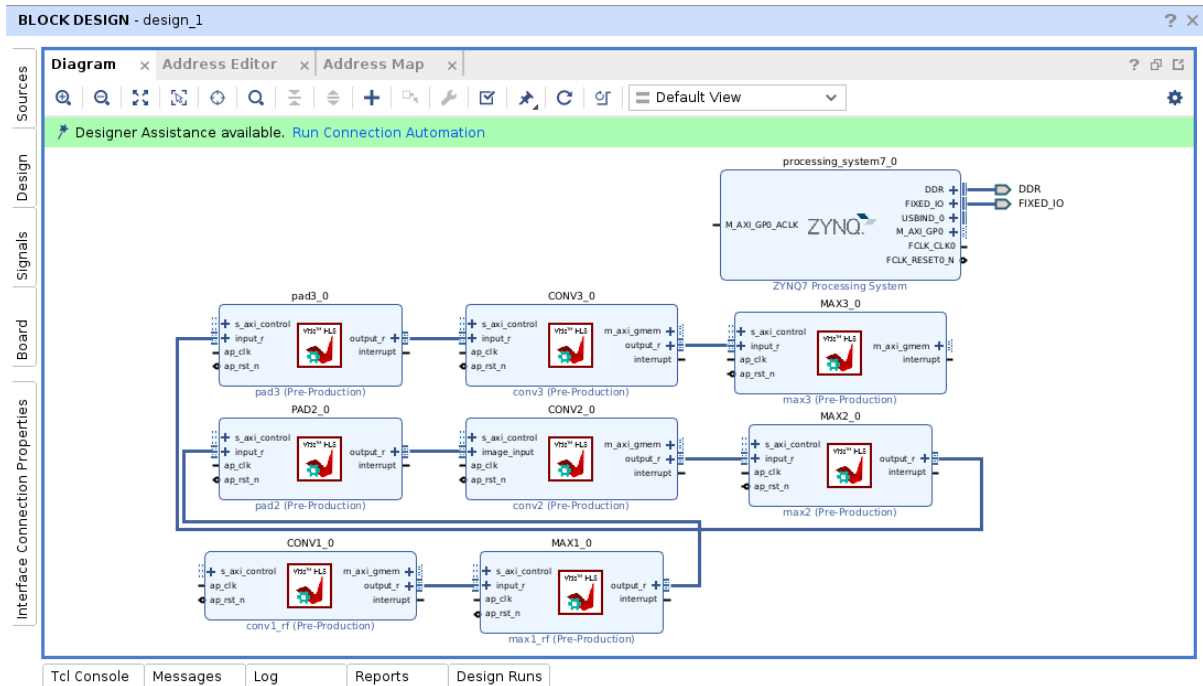


Figure 4.15 : Addition of the different IP Cores to the new Block Design

We can see as well that each IP block previously generated has different ports, we take the example of the CONV3 block that is shown in the Figure 4.16 :

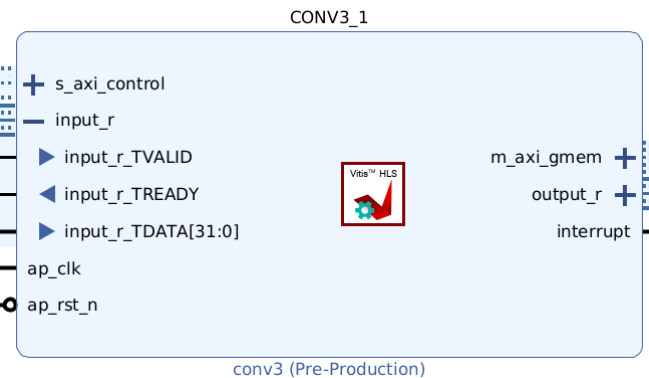


Figure 4.16 : CONV3 IP Block

This block has seven ports :

- s_axi_control : used for the control signals or the variables declared with s_axilite protocol,
- input_r : used for the input variables declared with axis protocol,
- m_axi_gmem : attributed to the variables that needs a memory access (the ones declared with m_axi protocol),
- output_r : used for the output variables declared with axis protocol,
- ap_clk : for the clock signal,
- ap_rst_n : for the reset signal,
- interrupt : used for the interrupt generated by the IP Block, using the port *return* that is triggered when the computation performed by the specific IP block is completed.

- After incorporating all the essential blocks and configuring the Processor to establish communication with the FPGA, the next step involves executing the **"Connection Automation"** process to facilitates the interconnection of all the blocks, ensuring their seamless integration, as follows (Figure 4.17) :

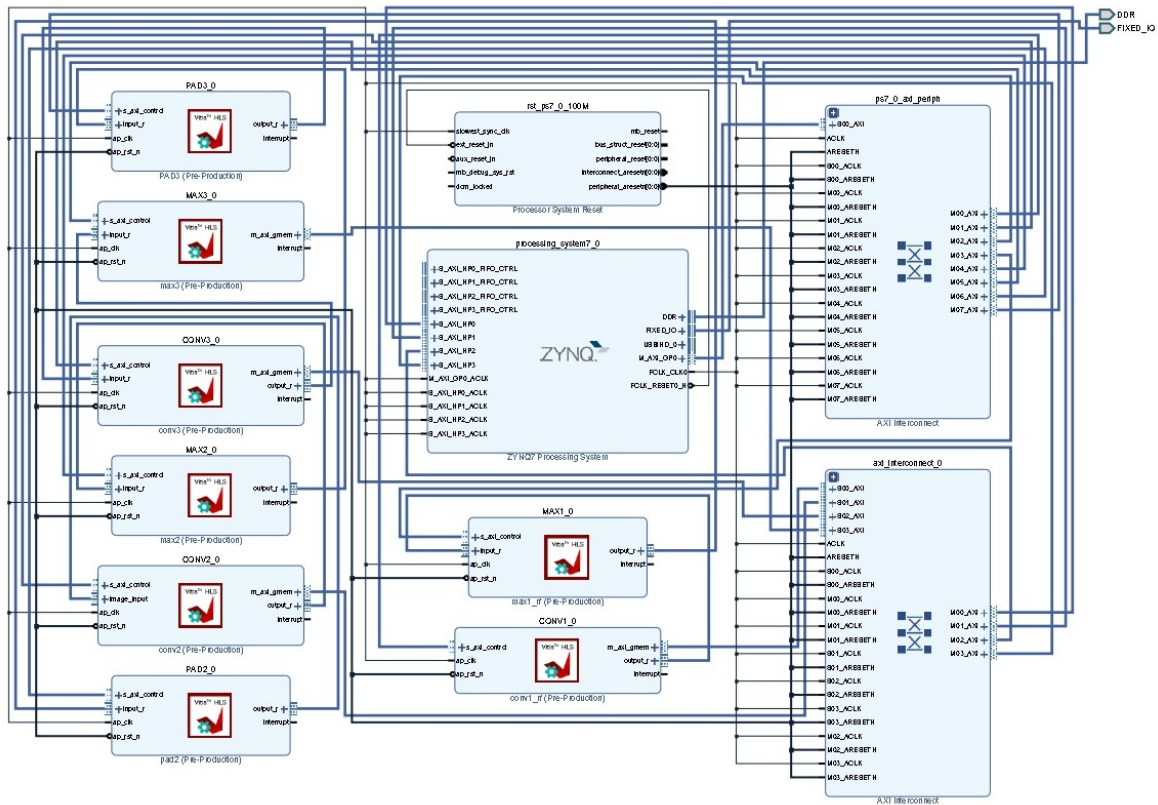


Figure 4.17 : Block Design of the Features Extraction sub-stage

Note :

- 1- In case there are some missing connections after running the "Connection Automation", we need to add them manually.
 - 2- We can see that there are some other blocks that have been added after the Connection Automation process. These blocks are :
 - *Processor System Reset* : which is responsible for initiating system-wide resets that specifically involve the processor.
 - two blocks of *AXI Interconnect* : used to adapt the clock, width or protocol used by the different ports of the different IP Blocks and the clock, width or protocol used by the processor. There are two blocks since we use two different protocols : m_axi and s_axilite.
- After establishing all the necessary connections, the next step is to **check our design**. When the design is validate, we can check and edit the addresses that have been attributed to each Slave and Master in the architecture in the window "Address Editor" as follows (Figure 4.18) :

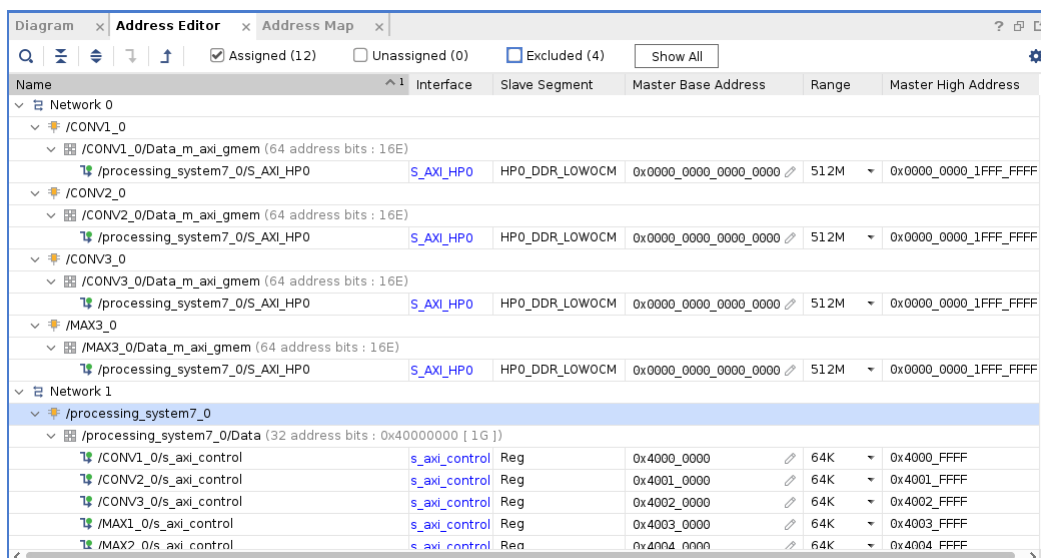


Figure 4.18 : Address Editor in Vivado Design Suit

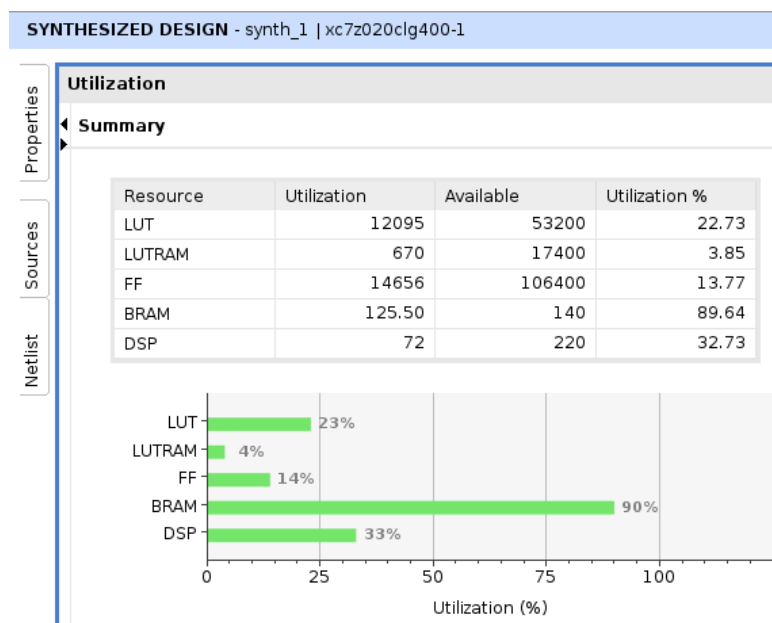
It is worth emphasizing that the presence of assigned addresses exclusively for ports with `m_axi` or `s_axi` interfaces is directly related to the inherent characteristics of the AXI Memory Mapped protocol. As this protocol operates on a Master-Slave architecture, it necessitates the allocation of addresses for both the Masters and Slaves within the system. Consequently, only ports that employ the `m_axi` or `s_axi` interface are designated with addresses in the system.

- Following this, we proceed with the generation of the **HDL wrapper**. The primary objective of this wrapper is to transform the design into a hardware description format that can be subsequently synthesized and implemented in the designated FPGA device.
- Once the HDL Wrapper is generated we can **synthesize** our system. The synthesis converts an HDL design into a gate-level representation. This gate-level representation consists of interconnected logic gates, flip-flops, and other digital components that can be physically implemented in the target FPGA device.

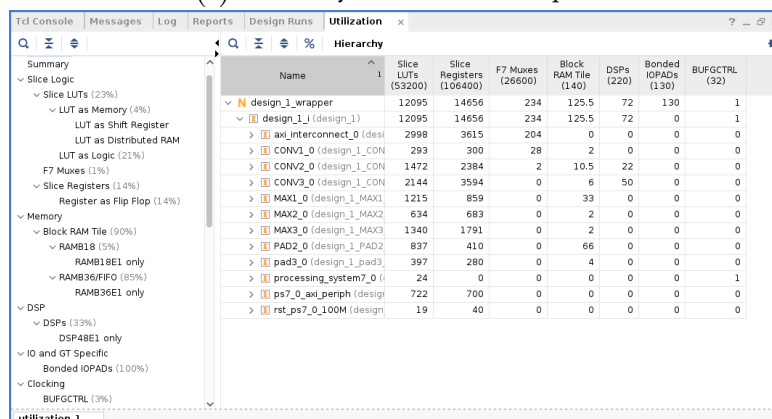
During synthesis, Vivado analyzes the RTL description of the design and performs various transformations to optimize it for the target FPGA. These transformations include :

- *Technology mapping* : The RTL design is mapped to specific logic elements available in the target FPGA, taking into consideration the device's architecture and resources.
- *Combinational logic optimization* : Vivado optimizes the combinational logic by minimizing the number of logic gates and reducing unnecessary logic operations. This helps to improve the overall performance and minimize resource utilization.
- *Sequential logic optimization* : The sequential logic elements, such as flip-flops and registers, are optimized to reduce the number of clock cycles required for data processing and improve the timing characteristics of the design.
- *Resource allocation* : Vivado assigns physical resources of the FPGA, such as lookup tables (LUTs), flip-flops, block RAM, and DSP slices, to different parts of the design to ensure efficient resource utilization and meet the design constraints [57].

When the synthesis is finished, we can analyze the different reports that we get, such as Utilization Report, a detailed and a summary report (Figure 4.19) and Power Report (Figure 4.20), as well :



(a) Summary of Utilization Report



(b) Detailed Utilization Report

Figure 4.19 : Utilization Report

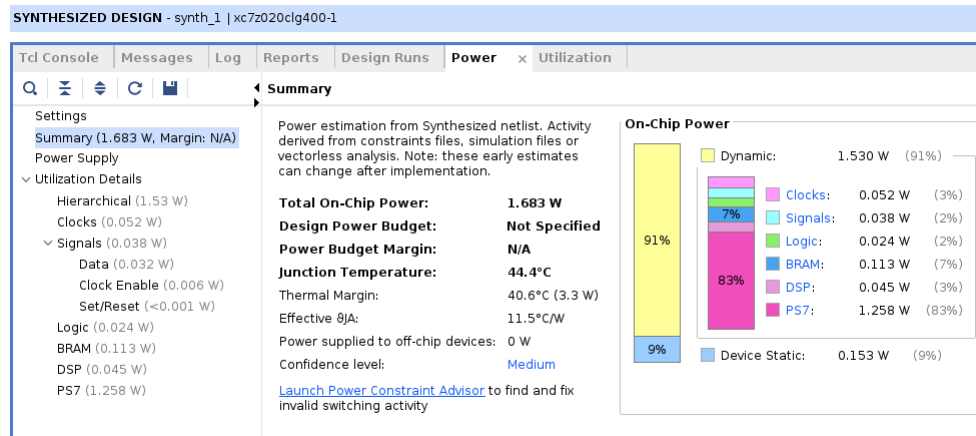


Figure 4.20 : Power Report

- The final step is to **implement the system and generate its Bitstream file**. In Vivado, implementation refers to the process of mapping and physically placing the synthesized design onto the target FPGA device. It involves several stages and tasks to transform the gate-level netlist generated during synthesis into a configuration bitstream that can be loaded onto the FPGA.

The main steps involved in the implementation process in Vivado are :

- *Placement* : it determines the physical location of each logic element in the FPGA's programmable fabric. It aims to find an optimal placement that minimizes delays, optimizes performance, and satisfies constraints.

- *Routing* : it establishes the physical connections between the logic elements based on the design's interconnections specified in the netlist.

- *Clock Planning* : it identifies clock domains, applies clock constraints, and optimizes clock routing to ensure proper synchronization and timing.

- *I/O Planning* : it maps the design's input/output signals to the appropriate physical pins on the device and applying I/O constraints for proper signaling and voltage levels.

- *Bitstream Generation* : The bitstream file contains the necessary instructions and configuration data to program the FPGA, enabling it to behave according to the desired functionality specified in the design.

- *Timing Analysis and Closure* : to ensure that the design meets the specified timing constraints.

- *Design Rule Check (DRC)* : to ensure that the design adheres to the manufacturing rules and guidelines specified by the target FPGA device. It checks for any violations such as short circuits, excessive congestion, or overlapping components[57].

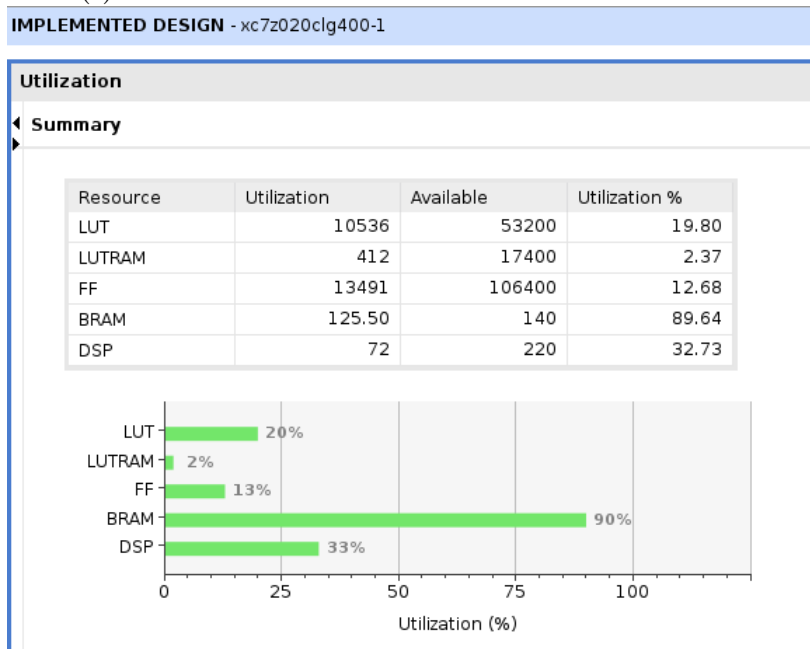
Once the implementation process is complete, the resulting Bitstream file can be loaded onto the FPGA, configuring it to operate based on the desired functionality defined in the original design.

Similar to the synthesis stage, the implementation process in Vivado produces various reports that offer valuable insights into the resource utilization of our system on the FPGA. These reports serve as crucial sources of information, enabling us to analyze and understand how the FPGA resources are utilized by our design. The most important ones for us are the Utilization Report (Figure 4.21) and the Power Report (Figure 4.22) :

UTILIZATION

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Title (140)	DSPs (220)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	10536	13491	30	4586	10124	412	125.5	72	130	1
design_1_i(de)	10536	13491	30	4586	10124	412	125.5	72	0	1
aw_iintercor	2243	2622	0	835	2144	99	0	0	0	0
CONV1_0(d)	289	301	28	111	289	0	2	0	0	0
CONV2_0(d)	1354	2381	2	734	1301	53	10.5	22	0	0
CONV3_0(d)	1886	3591	0	1155	1782	104	6	50	0	0
MAX1_0(de)	1164	797	0	460	1164	0	33	0	0	0
MAX2_0(de)	595	683	0	240	595	0	2	0	0	0
MAX3_0(de)	1194	1774	0	540	1100	94	2	0	0	0
PAD2_0(de)	798	410	0	336	798	0	66	0	0	0
pad3_0(de)	356	280	0	171	356	0	4	0	0	0
processing_	0	0	0	0	0	0	0	0	0	1
ps7_0_axi_p	641	619	0	309	580	61	0	0	0	0
rst_ps7_0_1	17	33	0	12	16	1	0	0	0	0

(a) Table Summary of Resources Usage - Detailed Report



(b) Graph Summary of Resources Usage

Figure 4.21 : Resources Usage Report

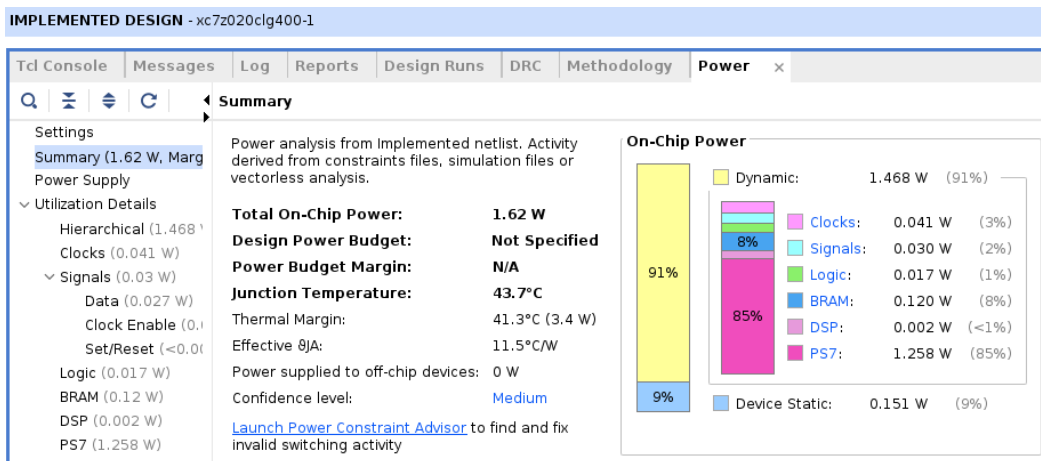


Figure 4.22 : Summary of Power Report

4.3.5 CPU Programming

After generating the Bitstream file, the next step is to deploy it on the FPGA. However, prior to that, we must program the CPU within the design to establish communication with the FPGA. To proceed, it is essential to familiarize ourselves with the board we are working with. This involves understanding the CPU programming procedure and conducting simulations to verify the proper functioning of our architecture. By introducing the board, we can gain the necessary knowledge to program the CPU and execute simulations for quality assurance.

A. PYNQ-Z1 Board

The PYNQ-Z1 board is a Xilinx Zynq All Programmable SoCs (APSoCs). The AMD-Xilinx® Zynq® All Programmable device is a SOC based on a dual-core ARM® Cortex®-A9 processor (referred to as the Processing System or PS), integrated with FPGA fabric (referred to as Programmable Logic or PL). The PS subsystem includes a number of dedicated peripherals (memory controllers, USB, Uart, IIC, SPI etc) and can be extended with additional hardware IP in a PL hardware libraries [59]. The Figure 4.23[59] gives sn overview of the architecture of the board :

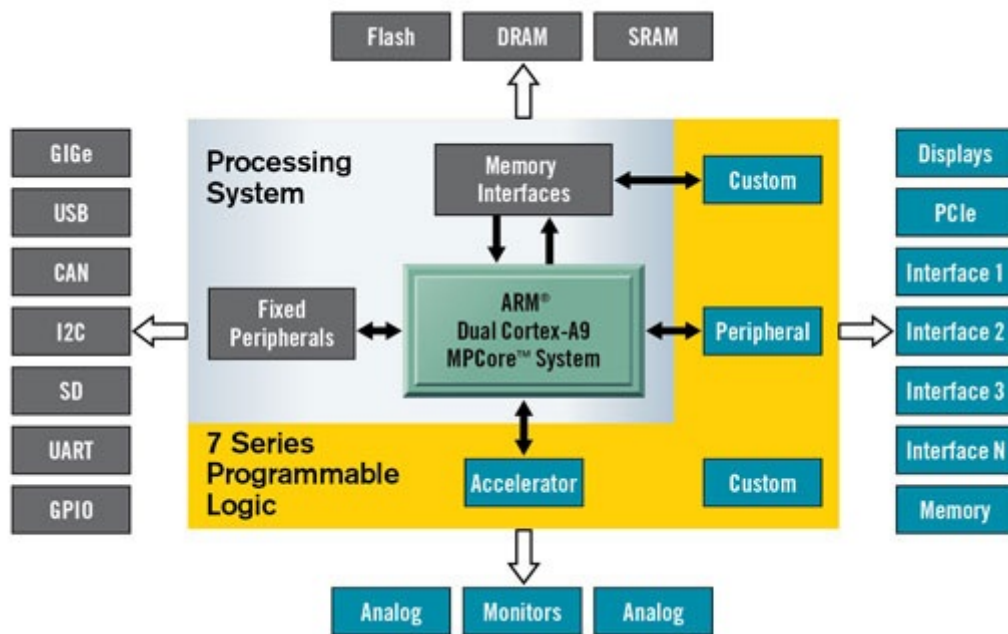


Figure 4.23 : Architecture of the PYNQ-Z1 Board

The PYNQ-Z1 can be programmed using Python and the Jupyter Notebook, which is a software that allows interactive computing. The programmable logic circuits that are implemented in the FPGA are imported as hardware libraries called "Overlays" and controlled through their APIs, mirroring the process of importing and programming software libraries [59].

Overlays

Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the application of the Zynq processing system to the programmable logic. These overlays can be used to accelerate a software application or customize the hardware platform for a specific application.

PYNQ provides a Python interface to allow overlays in the PL to be controlled from Python running in the PS. The `pynq.Overlay` module is used to manage the state and content of a PYNQ overlay. This module adds additional functionality to the programmable logic (PL) module.

4.3.6 Programming Approach

We included a CPU in our design primarily because the Memory Block can only be controlled by the CPU. Since we require the Memory Block to store essential data such as the input image, model weights and biases, and the output vector, it became necessary to incorporate the CPU. Consequently, we had to follow the following essential steps for programming the CPU in our application :

- .1 Create a project for our system (a folder in which we create a Notebook file),
- .2 Upload the Bitstream file, the Tool Command Language (TCL) file as well as the Metadata file (having the .hwh extension) to the folder of our project. It is primary to know that the TCL file is a script for controlling and extending software applications, it helps in building the system (design or architecture) from sources. In other hand, the HWH file includes a complete set of register values, interfaces, and connections of the design. This file is generated automatically by Vivado during implementation. This two files helps the CPU recognize the architecture implemented in the FPGA and how it works.
- .3 Moving to the code now, we need first to create the overlay of our architecture using the *"Overlay" module* and make sure that all the IP Blocks of our system figure in the list of IP Blocks recognized by the CPU. The Figure 4.24 shows the list of IP Blocks for the created overlay :

```
Type: Overlay
String form: <pynq.overlay.Overlay object at 0xab2ff3b8>
File: /usr/local/share/pynq-venv/lib/python3.10/site-packages/pynq/overlay.py
Docstring:
Default documentation for overlay design_1.bit. The following
attributes are available on this overlay:

IP Blocks
-----
CONV1_0 : pynq.overlay.DefaultIP
CONV2_0 : pynq.overlay.DefaultIP
CONV3_0 : pynq.overlay.DefaultIP
MAX1_0 : pynq.overlay.DefaultIP
MAX2_0 : pynq.overlay.DefaultIP
MAX3_0 : pynq.overlay.DefaultIP
PAD2_0 : pynq.overlay.DefaultIP
PAD3_0 : pynq.overlay.DefaultIP
processing_system7_0 : pynq.overlay.DefaultIP
```

Figure 4.24 : Description of The Overlay created and the IP Blocks of this last

Before moving forward, we can check the details of the different IP Blocks. The Figure 4.25 shows a part of the description of the CONV1 Block, its registers and the fields of each one :

```
print(ol.ip_dict["conv1_0"])
{'type': 'xilinx.com:hls:conv1:1.0', 'mem_id': 's_axi_control', 'memtype': 'REGISTER', 'gpio': {}, 'interrupts':
 {}, 'parameters': {'C_S_AXI_CONTROL_ADDR_WIDTH': '6', 'C_S_AXI_CONTROL_DATA_WIDTH': '32', 'C_M_AXI_GMEM_ID_WIDTH':
 '1', 'C_M_AXI_GMEM_ADDR_WIDTH': '64', 'C_M_AXI_GMEM_DATA_WIDTH': '32', 'C_M_AXI_GMEM_AUSER_WIDTH': '1', 'C_M_AXI_G
MEM_ARUSER_WIDTH': '1', 'C_M_AXI_GMEM_WUSER_WIDTH': '1', 'C_M_AXI_GMEM_RUSER_WIDTH': '1', 'C_M_AXI_GMEM_BUSER_WIDT
H': '1', 'C_M_AXI_GMEM_USER_VALUE': '0x00000000', 'C_M_AXI_GMEM_PROT_VALUE': '000', 'C_M_AXI_GMEM_CACHE_VALUE':
'0011', 'C_M_AXI_GMEM_ENABLE_ID_PORTS': 'true', 'C_M_AXI_GMEM_ENABLE_USER_PORTS': 'false', 'Component Name': 'des
ign_1_conv1_0_0', 'clk_period': '10', 'machine': '64', 'combinational': '0', 'latency': '172808', 'II': 'x', 'EDK_I
PTYPE': 'PERIPHERAL', 'C_S_AXI_CONTROL_BASEADDR': '0x40000000', 'C_S_AXI_CONTROL_HIGHADDR': '0x4000ffff', 'ADDR_WID
TH': '6', 'DATA_WIDTH': '32', 'PROTOCOL': 'AXI4LITE', 'READ_WRITE_MODE': 'READ_WRITE', 'FREQ_HZ': '100000000', 'ID
WIDTH': '0', 'AUSER_WIDTH': '0', 'ARUSER_WIDTH': '0', 'WUSER_WIDTH': '0', 'RUSER_WIDTH': '0', 'BUSER_WIDTH': '0',
'HAS_BURST': '0', 'HAS_LOCK': '0', 'HAS_PROT': '0', 'HAS_CACHE': '0', 'HAS_QOS': '0', 'HAS_REGION': '0', 'HAS_WSTR
B': '1', 'HAS_BRESP': '1', 'HAS_RRESP': '1', 'SUPPORTS_NARROW_BURST': '0', 'NUM_READ_OUTSTANDING': '1', 'NUM_WRITE
OUTSTANDING': '1', 'MAX_BURST_LENGTH': '1', 'PHASE': '0.0', 'CLK_DOMAIN': 'design_1_processing_system7_0_0_FCLK_CLK
0', 'NUM_READ_THREADS': '1', 'NUM_WRITE_THREADS': '1', 'RUSER_BITS_PER_BYTE': '0', 'WUSER_BITS_PER_BYTE': '0', 'INS
ERT_VIP': '0', 'MAX_READ_BURST_LENGTH': '16', 'MAX_WRITE_BURST_LENGTH': '16', 'TDATA_NUM_BYTES': '4', 'TUSER_WIDT
H': '0', 'LAYERED_METADATA': None, 'TDEST_WIDTH': '0', 'TID_WIDTH': '0', 'HAS_TREADY': '1', 'HAS_TSTRB': '0', 'HAS
_TKEEP': '0', 'HAS_TLAST': '0', 'registers': {'CTRL': {'address_offset': 0, 'size': 32, 'access': 'read-write', 'de
scription': 'Control signals', 'fields': {'AP_START': {'bit_offset': 0, 'bit_width': 1, 'access': 'read-write', 'de
scription': 'Control signal Register for 'ap_start'.'}, 'AP_DONE': {'bit_offset': 1, 'bit_width': 1, 'access': 'rea
d-only', 'description': 'Control signal Register for 'ap done'.'}, 'AP_IDLE': {'bit_offset': 2, 'bit_width': 1, 'ac
cess': 'read-only', 'description': 'Control signal Register for 'ap idle'.'}, 'AP_READY': {'bit_offset': 3, 'bit_wi
dth': 1, 'access': 'read-only', 'description': 'Control signal Register for 'ap ready'.'}, 'RESERVED_1': {'bit_offs
```

Figure 4.25 : Details of the CONV1 Block

4. After that, we need to map **each IP Block** to addresses using the MMIO class. This class allows Python objects (like IP Blocks of an Overlay) to access to the System Memory Mapped. The Figure 4.26 is an example of the mapping of the CONV1 block :

```
ip_input = ol.CONV1_0
mmio_in = ip_input.mmio
register_map_in = ip_input.register_map
registers_in = register_map_in._register_classes
```

Figure 4.26 : Mapping of CONV1 Block Using MMIO Class

We can check the Register Map as well to identify the registers of Control Signals that we have, the Figure 4.27 is the Register Map of CONV1 Block :

```
for name, reg in registers_in.items():
    print(name, reg)
CTRL (<class 'pynq.registers.RegisterCTRL'>, 0, 32, None, None, 'read-write')
GIER (<class 'pynq.registers.RegisterGIER'>, 4, 32, None, None, 'read-write')
IP_IER (<class 'pynq.registers.RegisterIP_IER'>, 8, 32, None, None, 'read-write')
IP_ISR (<class 'pynq.registers.RegisterIP_ISR'>, 12, 32, None, None, 'read-write')
ap_return (<class 'pynq.registers.Registerap_return'>, 16, 32, None, None, 'read-only')
image_input_1 (<class 'pynq.registers.Registerimage_input_1'>, 24, 32, None, None, 'write-only')
image_input_2 (<class 'pynq.registers.Registerimage_input_2'>, 28, 32, None, None, 'write-only')
weights_l1_1 (<class 'pynq.registers.Registerweights_l1_1'>, 36, 32, None, None, 'write-only')
weights_l1_2 (<class 'pynq.registers.Registerweights_l1_2'>, 40, 32, None, None, 'write-only')
biases_l1 (<class 'pynq.registers.Registerbiases_l1'>, 48, 32, None, None, 'write-only')
biases_l2 (<class 'pynq.registers.Registerbiases_l2'>, 52, 32, None, None, 'write-only')
```

Figure 4.27 : Register Map of CONV1

5. Once the mapping is done, we allocate the memory for our weights and biases as well as the input and output using the *Allocate module* and assign the allocated memory to each variable we have which are : input image, output features vector, three weights vectors and three biases vectors (Figure 4.28) :

```

# Allocated buffer for input image (m_axi)
input_buffer_size = 128*128
input_buffer1 = allocate(shape=(input_buffer_size), dtype=np.uint8, cacheable=False)
register_map_in.image_input_1.image_input = input_buffer1.device_address

# Allocated buffers for input weights (m_axi)
input_buffer_size = 96*11*11
input_buffer2 = allocate(shape=(input_buffer_size), dtype=np.int16, cacheable=False)
register_map_in.weights_l1_1.weights_l1 = input_buffer2.device_address

input_buffer_size = 16*3*3
input_buffer3 = allocate(shape=(input_buffer_size), dtype=np.int16, cacheable=False)
register_map_22.weights_l2_1.weights_l2 = input_buffer3.device_address

input_buffer_size = 8*5*5
input_buffer4 = allocate(shape=(input_buffer_size), dtype=np.int16, cacheable=False)
register_map_32.weights_l3_1.weights_l3 = input_buffer4.device_address

# Allocated buffers for input biases (m_axi)
input_buffer_size = 96
input_buffer5 = allocate(shape=(input_buffer_size), dtype=np.int16, cacheable=False)
register_map_in.biases1_1.biases1 = input_buffer5.device_address

input_buffer_size = 16
input_buffer6 = allocate(shape=(input_buffer_size), dtype=np.int16, cacheable=False)
register_map_22.biases2_1.biases2 = input_buffer6.device_address

input_buffer_size = 8
input_buffer7 = allocate(shape=(input_buffer_size), dtype=np.int16, cacheable=False)
register_map_32.biases3_1.biases3 = input_buffer7.device_address

# Allocated buffer for output (m_axi)
input_buffer_size = 6*6*8
input_buffer8 = allocate(shape=(input_buffer_size), dtype="u1", cacheable=False)
register_map_out.output_r_1.output_r = input_buffer8.device_address

```

Figure 4.28 : Memory Allocation for all the IP blocks

It is worth noting that we allocate Buffers to our variables due to their significant size, as they are in vector form. In contrast, if we were working with scalars, we would allocate registers, resulting in a considerably different program structure.

- Once the Buffers are allocated, we proceed to write our inputs into them. This includes the weights, biases, which have been saved after the model training process, as well as the input image. It is of utmost importance to guarantee the proper update of these Buffers by executing the "flush" function. Neglecting to do so can lead to erroneous outputs.
- Now having the input data stored in the Memory Block, we can launch the computations by setting the control signal START of each IP Block to 1 and waiting until all the computations of each block are accomplished as follow (Figure 4.29) :

```

# Write to input buffer
input_buffer1[:len(image)] = image
input_buffer2[:len(weights1)] = weights1
input_buffer3[:len(weights2)] = weights2
input_buffer4[:len(weights3)] = weights3
input_buffer5[:len(bias1)] = bias1
input_buffer6[:len(bias2)] = bias2
input_buffer7[:len(bias3)] = bias3

#to ensure that the buffers are updated
input_buffer1.flush()
input_buffer2.flush()
input_buffer3.flush()
input_buffer4.flush()
input_buffer5.flush()
input_buffer6.flush()
input_buffer7.flush()

# Send start signal
register_map_in.CTRL.AP_START = 1
register_map_12.CTRL.AP_START = 1
register_map_21.CTRL.AP_START = 1
register_map_22.CTRL.AP_START = 1
register_map_23.CTRL.AP_START = 1
register_map_31.CTRL.AP_START = 1
register_map_32.CTRL.AP_START = 1
register_map_out.CTRL.AP_START = 1

# Wait until algorithm has completed
while (register_map_in.CTRL.AP_DONE == 0): pass
while (register_map_12.CTRL.AP_DONE == 0): pass
while (register_map_21.CTRL.AP_DONE == 0): pass
while (register_map_22.CTRL.AP_DONE == 0): pass
while (register_map_23.CTRL.AP_DONE == 0): pass
while (register_map_31.CTRL.AP_DONE == 0): pass

```

Figure 4.29 : Strating the computation's code

The Control Signals, also called Handshake Signals, of the blocks works as follows (Figure 4.30) so we need to make sure to respect this behavior when running the computations :

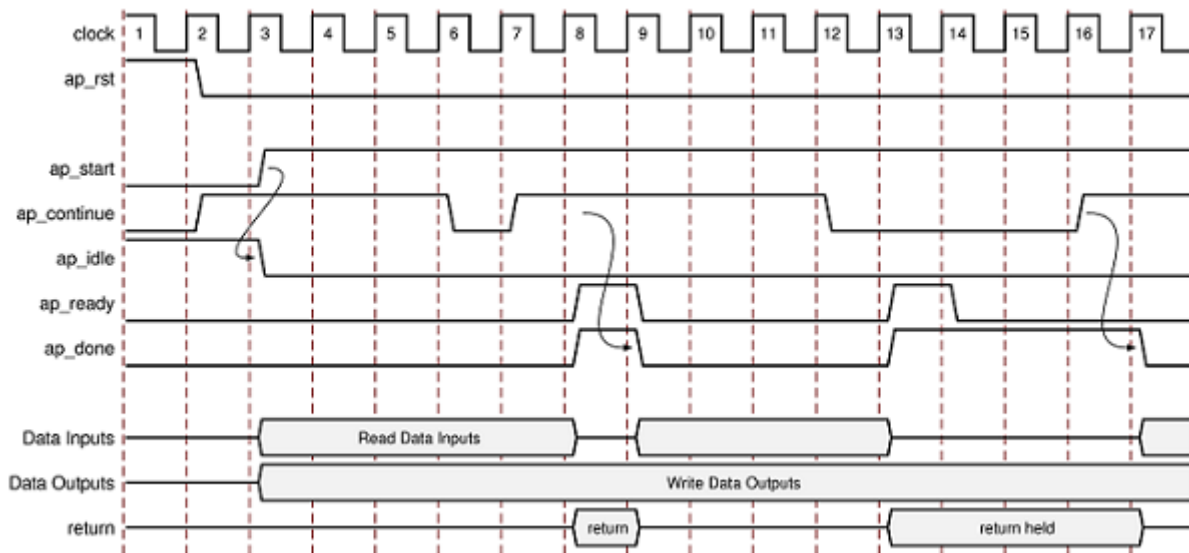


Figure 4.30 : The Behavior of the Block-Level Handshake Signals

Note : The detailed explanation of the handshake signals is described in the Vitis HLS User Guide in the Block-Level Control section [60].

- Once all the IP Blocks finish their respective computations, the output will be written in the **Buffer** 8 since we mapped it to the output of the last layer which is MAX3. So by printing this vector, we can check if our architecture gives the desired results and it's

working or not. The Figure 4.31 is the output of a random image generated, with weights all equals to 1 and null biases just to test the architecture :

```
print(input_buffer8)
[ 78 237 230 122 159 247 225 164  36  67  79 238  77  37  91 247  26 116
 215 192 191 220  85 156  97 232 207 116 207 231  41 252 239 203  65  78
 139 218 199  29  52  22 102 127 183 201  19  87 133 190 178 212 220 124
 119 168 132 248 205 126 196 191  44 235  56  89  5 105 218 233 205 231]
```

Figure 4.31 : Output Buffer

We can ensure this way that our architecture is working well since the output that we got matches the one we calculated ourselves.

Now since the architecture is working we can move to evaluate the results obtained.

4.3.7 Results

Given that our primary focus is on implementing the Features Extraction sub-stage on an FPGA and closely monitoring its resource utilization; the PYNQ-Z1 board with its combination of programmable logic (PL) and processing system (PS) proved to be an excellent choice. Overall, the capabilities offered by the PYNQ-Z1, with its PL and PS components, perfectly aligned with our project requirements.

In order to discuss our results regarding *resource usage* and *power consumption* of the Features Extraction sub-stage implemented in the PL of the board, let's consider some of the pertinent specifications of the Pynq-Z1 board that are relevant to our project :

- **Processing System (PS) :**

- Xilinx Zynq-7000 SoC (XC7Z020-1CLG400C),
- Dual-core ARM Cortex-A9 processor,
- 512 MB DDR3 RAM,
- 16 MB Quad-SPI Flash memory,
- MicroSD card slot for additional storage.

- **Programmable Logic (PL) :**

- Xilinx 7-series FPGA (XC7Z020-1CLG400C),
- 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops,
- 220 DSP slices,
- 630 KB of fast block RAM,

- **Power :**

USB or external power supply (7V to 15V DC)

- **Software Support :**

- PYNQ framework for Python-based development,

- Vivado Design Suite for FPGA synthesis and implementation,
- Vitis for embedded software development.

Now, we present a summary of the resources consumed by the Features Extraction sub-stage (Table 4.3) :

Resources	LUT	LUTRAM	FF	DSP	BRAM
Available	53200	17400	106400	220	140
Used	10536	412	13491	72	125.5
Consumption (%)	20	2	13	33	90

Table 4.3 : Resources Consumption of the Features Extraction sub-stage on PYNQ Z1

With the exception of BRAM (Block RAM) usage, which is necessary due to the storage of weights and biases, we clearly see that the utilization of other resources is deemed acceptable.

4.4 Matlab approach

The Deep Learning on FPGA solution provides with an end-to-end workflow to compile, deploy, profile and debug our custom pretrained deep learning networks. It also generates a custom deep learning processor IP core that we can integrate into our custom reference design.

The implementation of a CNN model on an FPGA is indeed a popular approach for accelerating deep learning inference tasks. One of the methodologies out there is Matlab by using Deep Learning Toolbox [27] and Deep Learning HDL Toolbox [28] which are part of the MATLAB environment and can aid in the design and deployment of CNN models on FPGAs.

The figure 4.46 shows the MATLAB based deep learning on FPGA solution

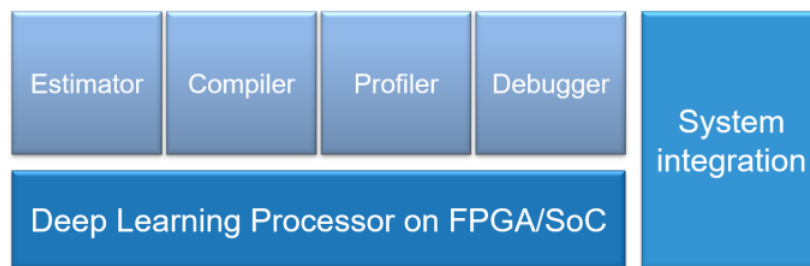


Figure 4.32 : Matlab solution for Deep learning on FPGA

Here's a general overview of the process involved in implementing a CNN model on an FPGA based on the documentation of their official website [29] :

- **CNN Model Design** : We start by designing and training our CNN model using frameworks like TensorFlow, PyTorch, or MATLAB's Deep Learning Toolbox. This step

involves selecting appropriate network architectures, configuring layers, and training the model on a suitable dataset.

- **Fixed-Point Conversion** : Since FPGAs typically operate using fixed-point arithmetic, it is necessary to convert the floating-point parameters of the trained model into fixed-point representation. This conversion is crucial to ensure numerical accuracy during inference.
- **Code Generation** : Once the model is trained and converted to fixed-point representation, we can use MATLAB's Deep Learning HDL Toolbox to generate synthesizable Verilog or VHDL code for the FPGA implementation. The toolbox assists in generating optimized HDL code tailored for the target FPGA platform.
- **IP Core Generation** : The generated HDL code is then used to create an Intellectual Property (IP) core, which is a reusable hardware component representing the CNN model. The IP core encapsulates the functionality and connectivity of the CNN layers and can be instantiated multiple times within the FPGA design.
- **Bitstream Generation** : After generating the IP core, the next step is to integrate it into the overall FPGA design and create a bitstream—a binary file that configures the FPGA with the desired functionality. The bitstream contains the necessary instructions for the FPGA to implement the CNN model.
- **FPGA Deployment and Inference** : Finally, we can load the generated bitstream onto the FPGA and perform CNN inference on input data. The FPGA executes the CNN model in a highly parallel manner, leveraging the inherent parallelism of the FPGA fabric to accelerate the computation.

4.4.1 Quantisation for FPGA Target

The core idea behind quantization is the resiliency of neural networks to noise; deep neural networks, in particular, are trained to pick up key patterns and ignore noise. This means that the networks can cope with the small changes in the weights and biases of the network resulting from the quantization error and there a growing number of works indicating the minimal impact of quantization on the accuracy of the overall network. This, coupled with significant reduction in memory footprint, power consumption, and gains in computational speed, makes quantization an efficient approach for deploying neural networks to embedded hardware.

The Deep Network Quantizer app presented by matlab typically provides the following functionalities :

- A. **Model analysis** : allows the loading of pre-trained deep neural network model and analyze its structure, layer-wise configurations, and parameter sizes.
 - .1 **Save the model** : The `.h5` file format is commonly used to save Keras models in Python. It is a hierarchical data format **HDF5** that allows for efficient storage and retrieval of large amounts of numerical data, including deep learning models.

By loading the `.h5` file into MATLAB, we added the layers, weights, and other attributes of the Keras model. This allows us to perform various operations on the model, such as making predictions, extracting intermediate layer outputs.

So, in order to import the model to the Matlab Workspace, the model needs to be converted and saved to the `.h5` format as shown in figure 4.34

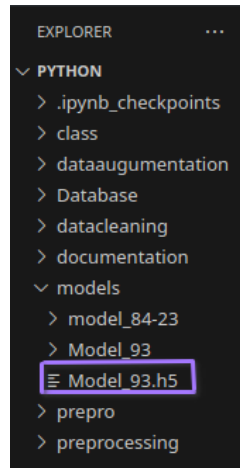


Figure 4.33 : Model saved

- .2 **Loading the Keras model in MATLAB** : MATLAB provides a function called **"importKerasNetwork"** to load our Keras model. We used this function to load the .h5 file containing our Keras model.

```

### The network includes the following layers:
1 'input_1' Image Input 128x128x1 images (SW Layer)
2 'conv2d' 2-D Convolution 96 11x11x1 convolutions with stride [4 4] and padding [0 0 0 0] (HW Layer)
3 'conv2d_relu' ReLU (HW Layer)
4 'max_pooling2d' 2-D Max Pooling 2x2 max pooling with stride [2 2] and padding [0 0 0 0] (HW Layer)
5 'conv2d_1' 2-D Convolution 16 3x3x96 convolutions with stride [1 1] and padding [2 2 2 2] (HW Layer)
6 'conv2d_1_relu' ReLU (HW Layer)
7 'max_pooling2d_1' 2-D Max Pooling 2x2 max pooling with stride [2 2] and padding [0 0 0 0] (HW Layer)
8 'conv2d_2' 2-D Convolution 8 5x5x16 convolutions with stride [1 1] and padding [1 1 1 1] (HW Layer)
9 'conv2d_2_relu' ReLU (HW Layer)
10 'max_pooling2d_2' 2-D Max Pooling 2x2 max pooling with stride [2 2] and padding [0 0 0 0] (HW Layer)
11 'dense' Fully Connected 4096 fully connected layer (HW Layer)
12 'dense_relu' ReLU (HW Layer)
13 'dense_1' Fully Connected 2048 fully connected layer (HW Layer)
14 'dense_1_relu' ReLU (HW Layer)
15 'dense_2' Fully Connected 64 fully connected layer (HW Layer)
16 'dense_2_softmax' Softmax softmax (SW Layer)
17 'ClassificationLayer_dense_2' Classification Output crossentropyex with '1' and 63 other classes (SW Layer)

```

Figure 4.34 : Architecture of our model imported to Matlab

- .3 **Define calibration and validation data** : In the context of deep learning, calibration and validation data are commonly used for quantization which are previously explained. It is important to note that the calibration and validation data should be chosen carefully to ensure they are representative of the real-world scenarios our model will face during inference. Biases or inadequacies in the calibration and validation datasets can affect the quantization process and the accuracy of the quantized model. Therefore, it is recommended to carefully curate and validate these datasets to achieve reliable and accurate quantization results.
- .4 **deepNetworkQuantizer** : The **"deepNetworkQuantizer"** function in MATLAB is a part of the Deep Learning Toolbox, we used it for quantizing deep neural network. It allowed us to quantize the weights and activations of our pretrained network to lower precision in order to reduce memory footprint and computational requirements. The quantizable layers is shown in the Figure 4.35

Layer Name	Min Value	Max Value	Quantize Layer
▼ input_1			<input checked="" type="checkbox"/>
Activations	0.0000	255.0000	
▼ conv2d			<input checked="" type="checkbox"/>
Weights	-0.1779	0.1206	
Bias	-0.5622	0.2465	
Activations	-1374.79...	380.3713	
▼ conv2d_relu			<input checked="" type="checkbox"/>
Activations	0.0000	380.3713	
▼ max_pooling2d			<input checked="" type="checkbox"/>
Activations	0.0000	380.3713	
▼ conv2d_1			<input checked="" type="checkbox"/>
Weights	-0.2770	0.2515	
Bias	-0.2661	0.1399	
Activations	-432.9354	112.5018	
▼ conv2d_1_relu			<input checked="" type="checkbox"/>
Activations	0.0000	112.5018	
▼ max_pooling2d_1			<input checked="" type="checkbox"/>
Activations	0.0000	112.5018	
▼ conv2d_2			<input checked="" type="checkbox"/>
Weights	-0.2215	0.2178	
Bias	-0.0194	0.0750	
Activations	-98.0411	35.8768	
▼ conv2d_2_relu			<input checked="" type="checkbox"/>
Activations	0.0000	35.8768	
▼ max_pooling2d_2			<input checked="" type="checkbox"/>
Activations	0.0000	35.8768	
▼ flatten			<input checked="" type="checkbox"/>
Activations	0.0000	35.8768	
▼ dense			<input checked="" type="checkbox"/>
Weights	-0.3175	0.2940	
Bias	-0.3712	0.4915	
Activations	-17.8413	7.2400	
▼ dense_relu			<input checked="" type="checkbox"/>
Activations	0.0000	7.2400	
▼ dense_1			<input checked="" type="checkbox"/>
Weights	-0.3309	0.3482	
Bias	-0.2085	0.2949	
Activations	-29.5429	29.9695	
▼ dense_1_relu			<input checked="" type="checkbox"/>
Activations	0.0000	29.9695	
▼ dense_2			<input checked="" type="checkbox"/>
Weights	-0.8047	0.2179	
Bias	-0.1969	0.1373	
Activations	-205.2808	32.9379	
▼ dense_2_softmax			<input checked="" type="checkbox"/>
Activations	0.0000	1.0000	
▼ ClassificationLayer_...			<input checked="" type="checkbox"/>
Activations	0.0000	1.0000	

Figure 4.35 : Quantifiability of each layer

When it comes to the allocation for each variable, we have figure 4.36 :

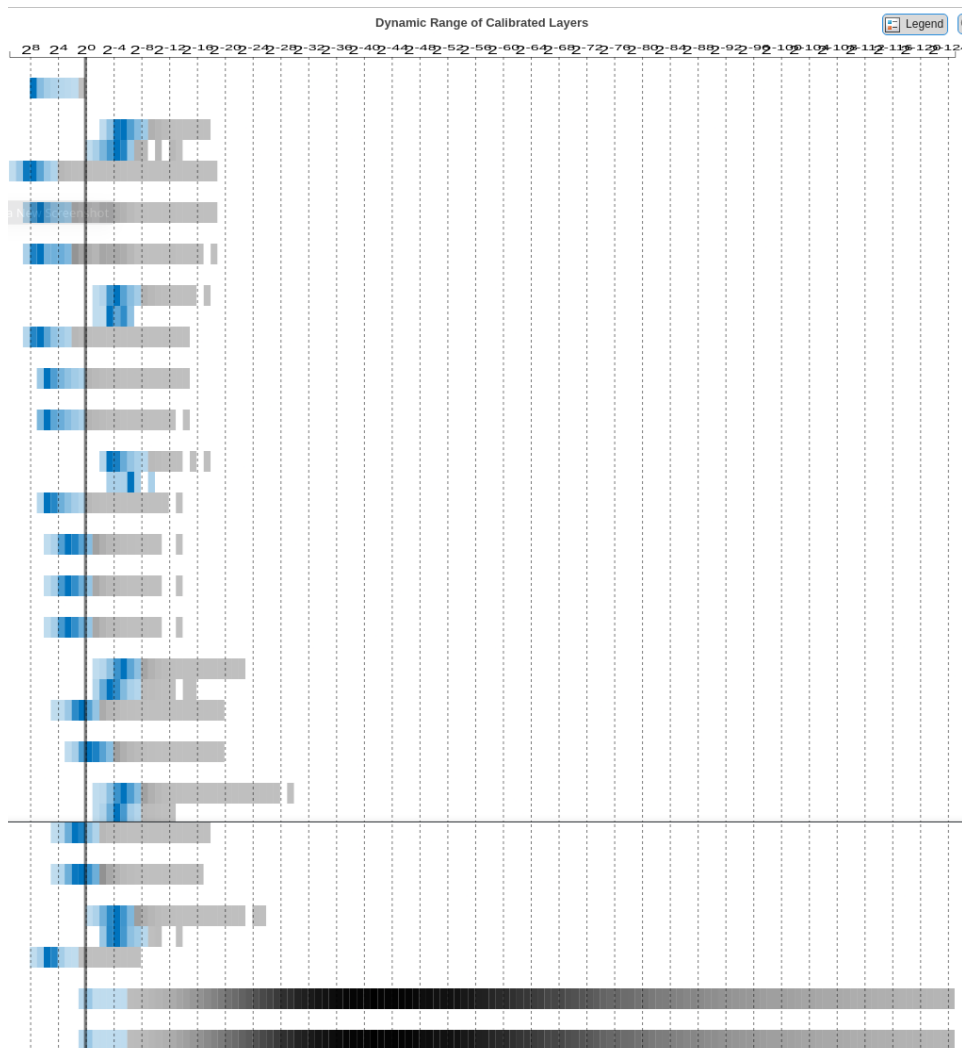


Figure 4.36 : Dynamic Range of Calibration Layers

- B. **Quantization options** : The application offers various quantization options, allowing us to choose the precision and bit-width for weights and activations. It may provide options for fixed-point or integer quantization, specifying the number of bits for each value.
- C. **Quantization performance evaluation** : We evaluated the performance of a quantized model by measuring its accuracy on a validation dataset. It provides metrics such as top-1 and top-5 accuracy, loss, and other relevant performance indicators but we choose accuracy because its the best metric to evaluate image classification.
- D. **Exporting quantized models** : Once the quantization process is complete, the application allows us to export the quantized model in a format suitable for deployment on various platforms or devices, in our case, an FPGA.

4.4.2 Prototype Deep Learning Networks on FPGA and SoC Devices

To accelerate the performance of deep learning tasks, specialized hardware accelerators called deep learning processors have emerged. These processors are designed to efficiently handle the

complex computations required by deep neural networks. In order to tailor these processors to the specific requirements of a custom board, the generation of a deep learning processor IP core becomes crucial. By creating a custom IP core, we can optimize the hardware architecture to maximize performance, power efficiency, and flexibility for their custom board. In this substage, we will explore the process of generating a deep learning processor IP core, covering the key steps involved in designing, implementing, and integrating the IP core into a custom board.

The figure 4.37 shows the process of deploying a network to a custom board and retrieving a prediction from the deployed network.

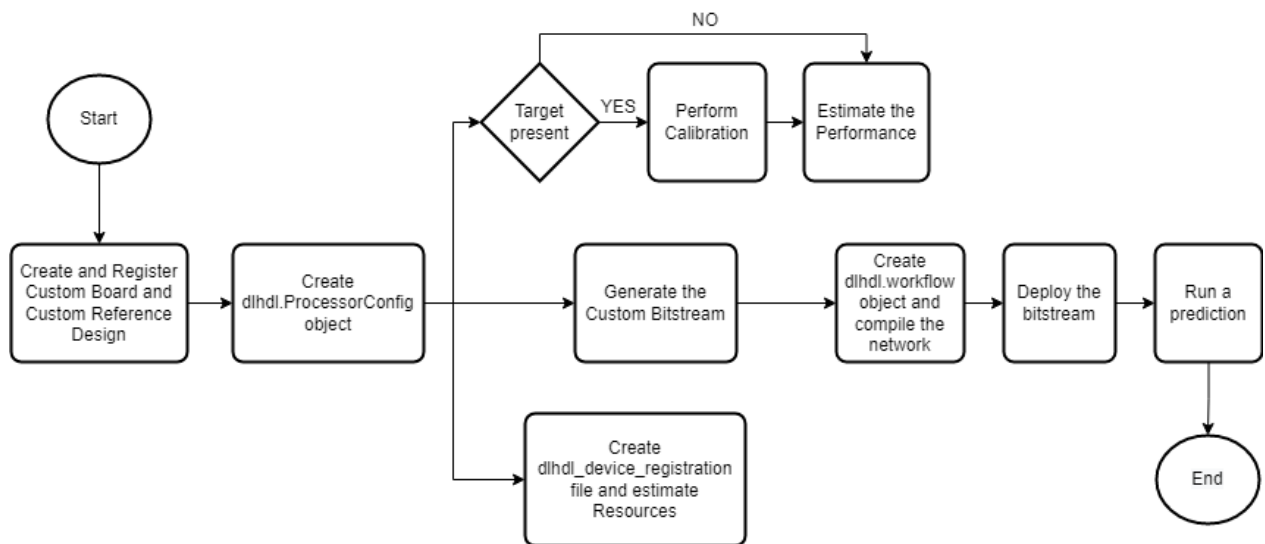
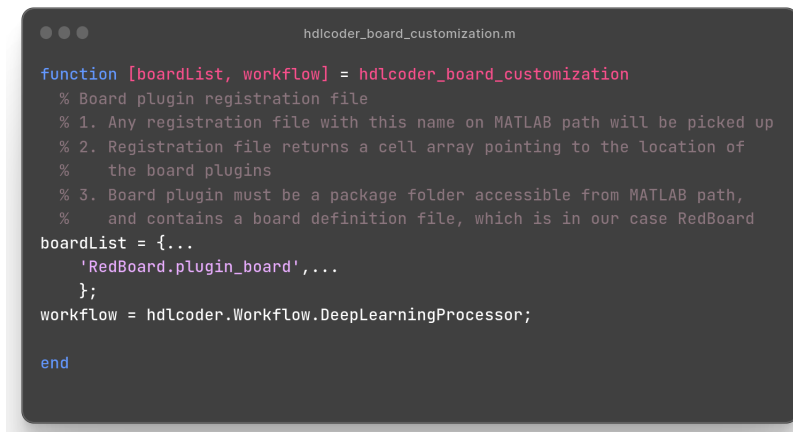


Figure 4.37 : The process of deploying a network to a custom board

A. Register Custom Board

Define the interface and attributes of a custom SoC board. To register the Xilinx® Kintex® UltraScale™ zc102 board.

- **Create a board registration file :** We define a function that returns a list of board plugins as a cell array of character vectors. the following figure 4.38 defines a board registration function :



```
hdlcoder_board_customization.m

function [boardList, workflow] = hdlcoder_board_customization
% Board plugin registration file
% 1. Any registration file with this name on MATLAB path will be picked up
% 2. Registration file returns a cell array pointing to the location of
%    the board plugins
% 3. Board plugin must be a package folder accessible from MATLAB path,
%    and contains a board definition file, which is in our case RedBoard
boardList = {...
    'RedBoard.plugin_board',...
};
workflow = hdlcoder.Workflow.DeepLearningProcessor;

end
```

Figure 4.38 : board registration function

- **Create the board definition file** : this step consist of creating an `hdlcoder.Board` object and specify its properties and interfaces according the characteristics of our FPGA.

B. Register Custom Reference Design

To define and register a reference design, we must have a reference design definition, a reference design plugin, and a reference design registration file.

- **Reference Design Definition** : A reference design definition is a file that defines the characteristics of a reference design, including its associated board and interfaces. We can define multiple custom reference designs per board. To generate a deep learning processor IP core, we must define these three AXI4 Master Interfaces :
 - **AXI4 Master Activation Data** :AXI4 Master Interface for the layer activation data with max data bit-width of 512
 - **AXI4 Master Weight Data** : AXI4 Master Interface for the layer weight data with max data bit-width of 512 `hRD.addAXI4MasterInterface(...`
 - **AXI4 Master Debug** : AXI4 Master Interface for the debugger with max data bit-width of 512
- **Reference Design Plugin** : A reference design plugin is a package folder that contains :
 - The reference design definition.
 - Files that are part of the embedded system design project, and are specific to our third-party synthesis tool, including Tcl, project, and design files.
- **Reference Design Registration Function** : A reference design registration function contains a list of reference design functions and the associated board name. to do that, we must define function that returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors. as shown in figure 4.39.

```
hdlcoder_ref_design_customization.m

function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file

rd = {'RedBoard.vivado_base_ref_design.plugin_rd',...
      };

boardName = 'Xilinx® Kintex® UltraScale™ zc102 board';

end
```

Figure 4.39 : Reference Design Registration Function

C. Performance Estimation

We reduce the time required to design and deploy a custom deep learning network that meets performance requirements by analyzing the layer-level latencies before deploying the network.

- Generating a calibration bitstream
 - Create a Processor Configuration object.
 - Specify the TargetPlatform
 - Reduce the number of parallel convolution processor kernel threads for the conv module
 - Set the Xilinx Vivado toolpath to our design tool. Set the Xilinx Vivado toolpath to our design tool using the `hdlsetuptoolpath` function, then build the calibration bitstream.
- Deploying the calibration bitstream to the target custom board Deploy the bitstream to the hardware and obtain the external- to-internal memory transaction latencies. We can use these values to get better estimates for the layer-level latencies.as shown in figure 4.40

```
deploy_bitstream.m

deployCalibrationBitstream(hPC, bitstreamPath);
```

Figure 4.40 : Calibration bitstream

- Retrieving the external to internal memory transaction latencies Estimate the performance of the network for the custom processor configuration. as shown in figure 4.41

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total Latency	Frames/s
Network	4137958	0.02069	1	4137958	48.3
conv2d	407387	0.00204			
max_pooling2d	73010	0.00037			
conv2d_1	52225	0.00026			
max_pooling2d_1	5266	0.00003			
conv2d_2	4925	0.00002			
max_pooling2d_2	994	0.00000			
dense	120674	0.00060			
dense_1	3384182	0.01692			
dense_2	89295	0.00045			

* The clock frequency of the DL processor is: 200MHz

Figure 4.41 : Internal memory transaction latencies

D. Resource Estimation

To estimate the resource utilization for our custom board that has a Kintex® Ultrascale chip family. The resource utilization is as shown in figure 4.42 :

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
Available	2520	912	274080
Total	384(16%)	586(65%)	251119(92%)
ReferenceDesign	3(1%)	78(9%)	35000(13%)
DL_Processor	381(16%)	508(56%)	216119(79%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

Figure 4.42 : Resource Estimation of original model

Compared to the quantified model in figure 4.43

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
Available	2520	912	274080
DL_Processor	33(2%)	50(6%)	32302(12%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

Figure 4.43 : Resource Estimation of quantified model

Based on our study, we observed a significant reduction in resource usage of over 80% after applying quantification, which was our primary goal. This remarkable decrease can be attributed to the implementation of dynamic quantization in MATLAB. Dynamic quantization, a specific type of quantization that dynamically adjusts quantization levels based on the characteristics of the input data, optimizes resource utilization and enhances overall efficiency.

E. IP Core Generation

The figure 4.44 shows the deep learning processor IP core architecture :

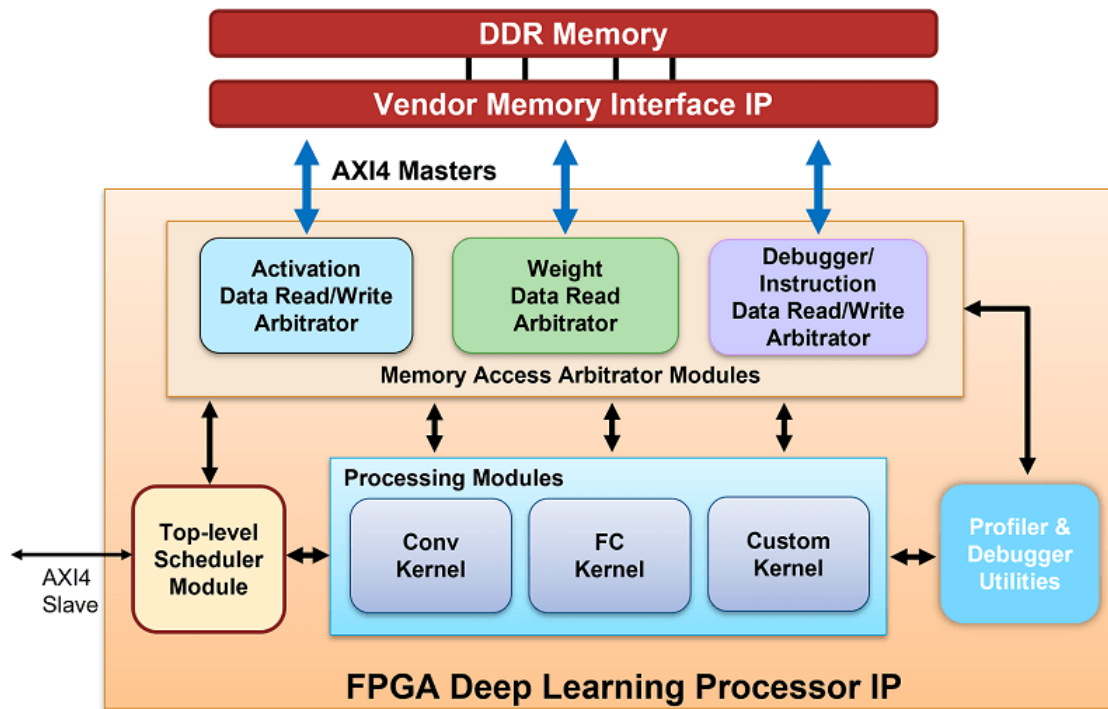


Figure 4.44 : Custom Bitstream for Custom Processor Configuration

We describe the architecture of the deep learning processor IP core using an image classification as following :

1. **DDR Memory** : The external DDR memory serves as a storage space for input images, weights, and output images. The processor incorporates three AXI4 master interfaces to communicate with the external memory. One of these interfaces is utilized to load input images into the processing modules. The compile method generates weight data, and the activation data can be retrieved from DDR using the External Memory Data Format. To initialize the deep learning processor, the weight data can be written to a deployment file, which eliminates the need for a MATLAB connection during deployment. For detailed information, refer to the "Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection" documentation.
2. **Memory Access Arbitrator Modules** : The activation and weight memory access arbitrator modules utilize the AXI Master interface to read and write weights and activation data to and from the processing modules. The profiler AXI Master interface handles reading and writing profiler timing data and instructions to the profiler module.
3. **Convolution Kernel** : The Conv Kernel is responsible for implementing layers with a convolution layer output format. It receives weights and activations for the layer through two AXI4 master interfaces. The Conv Kernel performs the necessary operations on the input image according to the implemented layer. This kernel is designed to be generic, supporting tensors and shapes of various sizes.
4. **Top-Level Scheduler Module** : The top-level scheduler module plays a vital role in

determining which instructions to execute, which data to read from DDR, and when to read that data. It acts as the central computer in a distributed computer architecture, distributing instructions to the processing modules. For example, if the network includes a convolution layer, fully connected layer, and a multiplication layer.

5. **Fully Connected Kernel** : The fully connected (FC) kernel is responsible for implementing layers with a fully connected layer output format. It receives weights and activations through two AXI4 master interfaces and performs the fully connected layer operation on the input image. Similar to the Conv Kernel, this kernel is also designed to be generic, supporting tensors and shapes of various sizes.
6. **Custom Kernel** : The custom kernel module is responsible for implementing layers that have been registered as custom layers using the "registerCustomLayer" method. Examples of such layers include addition layer, multiplication layer, resize2d layer, and more. To learn about creating, registering, and validating custom layers, refer to the "Register, Validate, and Deploy Custom Natural Logarithm Layer Network to FPGA" documentation. The custom kernel supports various layers.
7. **Profiler Utilities** : When the Profiler argument of the predict or predictAndUpdateState methods is set to "on," the profiler module collects information from the kernel, such as start and stop times for the Conv Kernel, FC Kernel, and more. This information is used to generate a profiler table with the collected results.

This figure 4.45 shows the generated deep learning processor IP core :

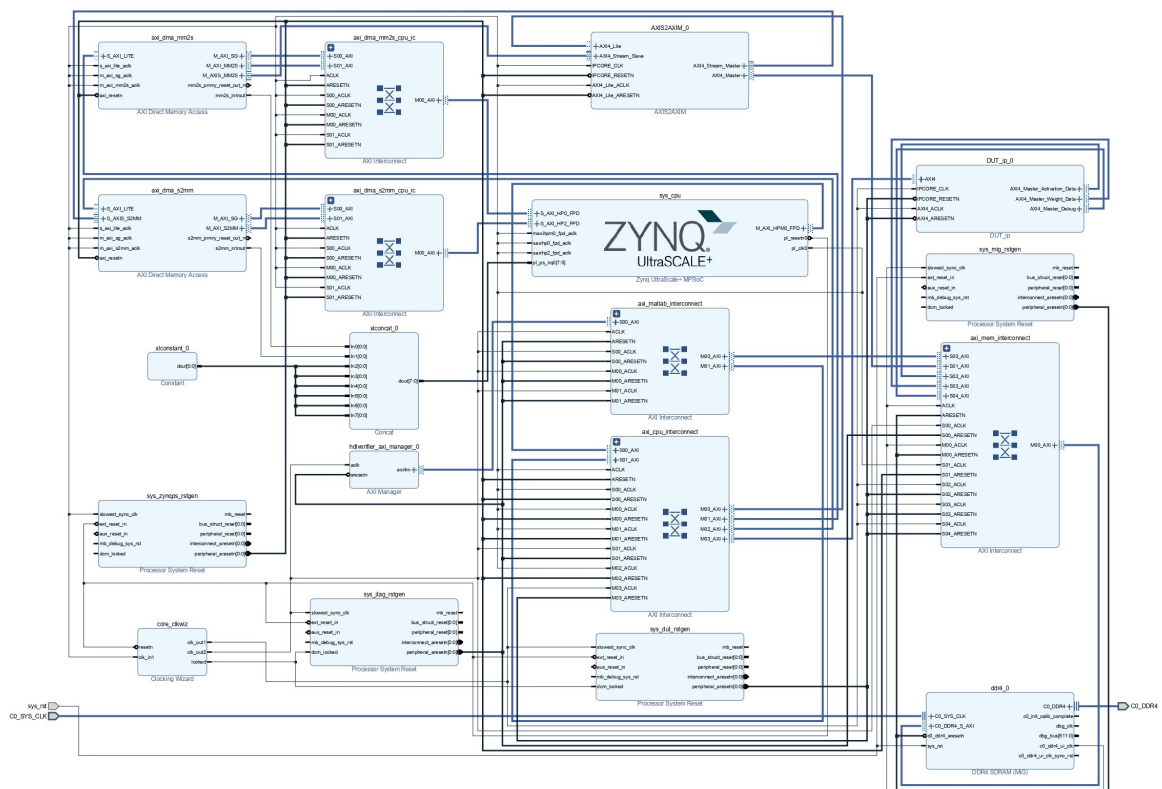


Figure 4.45 : Custom Bitstream for Custom Processor Configuration

Where it contains the following blocks :

1. **One DDR4 SDRAM** offers faster data transfer rates compared to previous generations,

- enabling quicker access to memory and improved overall system performance.
2. AXI DMA is designed to handle large amounts of data efficiently, providing high-speed transfer rates and reducing system latency. It is capable of moving data between memory and peripherals or between different memory locations.
 3. **Five blocks of AXI Interconnect** : used to adapt the clock, width or protocol used by the different ports of the different IP Blocks and the clock, width or protocol used by the processor.
 4. **Four Processor System Reset** : which is responsible for initiating system-wide resets that specifically involve the processor.
 5. **One concat** : responsible of concatenate the 8 inputs for one output.
 6. **One DUT-IP** : responsible for stimulus or test vectors to observe its behavior and verify that it performs as expected.

F. Generate Custom Bitstream for Custom Processor Configuration

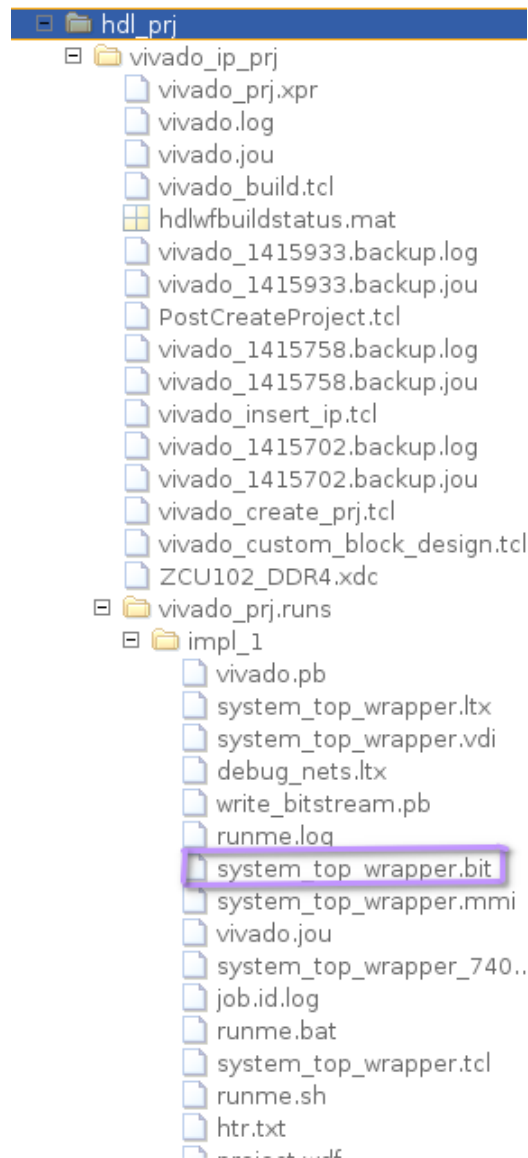


Figure 4.46 : Custom Bitstream for Custom Processor Configuration

G. Deploy the Custom Bitstream and Run Predictions on the Network

- **Create Target Object** : Create a target object with the vendor name of the target device. Specify the interface to connect the target device to the host using the Interface name-value pair. in our case it connects to the target using the JTAG interface.
- **Create Workflow Object for our network** : Create an object of the dlhdl.Workflow class. Specify the network, the bitstream name, and the target object.
- **Compile the Network** : Run the compile function of the dlhdl.Workflow object.
- **Deploy the Bitstream to the FPGA** : To deploy the network on the Xilinx Zc102 Kintex hardware, run the deploy function of the dlhdl.Workflow object.

4.4.3 Results and evaluation

The results of our image classification task after implementation are highly promising, showcasing the effectiveness of the employed methodology. We achieved an impressive accuracy rate of **97,20%** in correctly classifying images into their respective categories, demonstrating the robustness of our classification model. An exemple test of our implimented model in figure 4.47

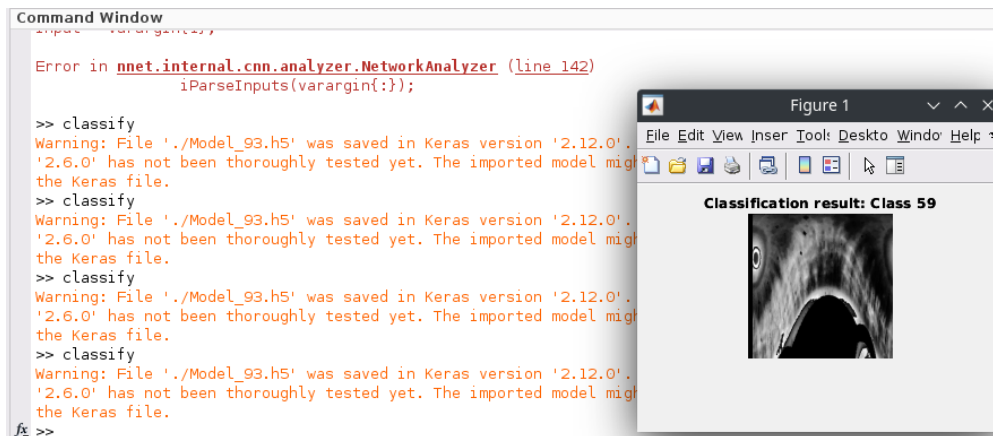


Figure 4.47 : Final result of the Matlab approach

4.5 Results Comparison

After implementing the system using the two approaches, it is important to note that the proposed architecture and application have not been previously implemented, which means there is no benchmark available for comparison. Therefore, the focus of our evaluation will be on comparing the performance of the two approaches directly.

In Figure 4.4, we present a comparative analysis of the two approaches based on various performance metrics. Since we don't have a benchmark to reference, our evaluation will primarily rely on the relative performance between the two approaches rather than absolute values.

Although lacking a benchmark for comparison, this evaluation of the two approaches will allow us to gain insights into their relative advantages and limitations. By analyzing the

performance metrics, we can determine which approach exhibits better resource utilization and execution time for our specific architecture and application.

Resources	Matlab Approach		HLS Approach	
	Utilization	Available	Utilization	Available
LUT	32302	274080	10536	53200
BRAM	50	912	125,5	140
DSP	33	2520	72	220

Table 4.4 : Resources utilization comparison between the two approaches

Upon analysis, we observe that the hls method outperforms the MATLAB approach for several reasons. Firstly, one notable distinction lies in the type of quantization utilized in each method.

In the MATLAB approach, dynamic quantization is employed, which means that the quantization process adapts dynamically to the input data. This approach ensures that no informative variables are discarded during quantization. While this adaptability may seem advantageous at first, it can potentially lead to a larger memory footprint and slower execution due to the need for additional processing to handle varying precision requirements.

In contrast, the hls method we used python for quantization which adopts a static pre-defined width of 16 bits for quantization. This fixed-width approach provides several benefits. Firstly, it allows for more efficient memory utilization since each variable is allocated a consistent number of bits. Additionally, the static nature of the quantization process simplifies hardware implementation, as the FPGA can be optimized specifically for the fixed-width representation.

By employing static quantization with a predefined width of 16 bits, the Python method achieves a more streamlined and efficient implementation compared to the MATLAB approach. This static approach eliminates the need for additional processing to handle varying precision requirements, resulting in improved execution time and reduced resource utilization.

Another key advantage of the HLS (High-Level Synthesis) approach is that it offers more efficient resource utilization compared to the MATLAB method. This can be attributed to a specific design choice we made in the HLS implementation.

In the HLS approach, we choose to encapsulate the model as a function, effectively considering the model itself as a function. This decision has several benefits, including reduced resource requirements. By treating the model as a function, we eliminate the need for redundant metadata and minimize the overhead associated with managing and coordinating information between different functions.

In contrast, MATLAB treats the model as a separate entity with distinct functions. As a result, additional resources are consumed due to the necessary inclusion of metadata and the communication overhead required between these functions.

The HLS approach's decision to consider the model as a function allows for more efficient resource allocation on the FPGA platform. By minimizing the metadata and communication overhead, the HLS approach optimizes resource utilization, leading to improved efficiency and reduced resource requirements.

Overall, the HLS approach outperforms MATLAB in terms of resource optimization. By

treating the model as a function, the HLS approach minimizes the need for additional resources, such as metadata and communication overhead. This resource optimization is a logical choice that contributes to an efficient and dependable biometric authentication solution when implemented on an FPGA platform.

4.6 Conclusion

In conclusion, the final stage of our system implementation focused on developing an iris recognition system on an FPGA with the aim of creating an efficient and reliable biometric authentication solution. The study began with optimizing the model and architecture, resulting in impressive outcomes. After pruning and quantization techniques were applied, the model achieved an impressive accuracy rate of **97.20%**.

Furthermore, significant efforts were made to reduce the FPGA resource utilization, resulting in notable improvements. The resource utilization was successfully optimized to utilize only **20%** of Look-Up Tables (LUT), **2%** of Random Access Memory (LUTRAM), **13%** of Flip-Flops (FF), **90%** of Block RAM (BRAM), and **33%** of Digital Signal Processors (DSP).

These achievements demonstrate the successful integration of the iris recognition system on the FPGA platform, ensuring high accuracy while efficiently utilizing the available resources. The reduced resource usage not only enhances the overall performance of the system but also offers potential cost savings and scalability benefits.

By achieving these outcomes, the project has contributed to the advancement of biometric authentication solutions and their practical implementation on FPGA technology. The optimized system's efficiency and reliability make it suitable for a wide range of applications, including access control, surveillance, and identity verification.

In conclusion, the successful implementation of the iris recognition system on an FPGA, coupled with the achieved accuracy rate and optimized resource utilization, provides a strong foundation for further advancements in biometric authentication systems. The findings of this project contribute to the development of more efficient and dependable solutions in the field of iris recognition, paving the way for enhanced security and reliability in various domains.

Conclusion and Perspectives

Iris footprints refer to the distinct patterns and characteristics present in a person's eye iris, which are used by iris recognition systems for biometric identification and authentication. Despite the benefits of this technology, there are several challenges associated with its implementation. The iris recognition algorithms employ intricate image processing and pattern matching techniques, imposing significant computational demands. Attaining good performance presents a challenge that necessitates efficient implementation of these algorithms, often through strategies such as algorithm simplification or hardware acceleration, which were the focal points of our project.

In addition to successfully implementing an End-to-End iris recognition system from scratch, this project has made several significant contributions, which can be summarized as follows :

- **New approach for reflections reduction :**

Unlike existing methods in the literature that handle reflections by either setting affected pixels to black or replacing them with a statistical value, our proposed approach aims to reduce reflections by adjusting the brightness of the reflective areas using Histogram Equalization. Our method focuses on minimizing information loss by extracting valuable details that may be obscured by high intensities in the presence of reflections.

- **New architecture for iris matching :**

After conducting numerous tests, we successfully developed a modified CNN architecture based on IRISNet. Our architecture incorporates three convolutional layers with filter sizes of $11 \times 11 \times 96$, $3 \times 3 \times 16$, and $5 \times 5 \times 8$ respectively, followed by max pooling layers with a 2×2 kernel after each convolutional layer. For classification, we employed a random forest classifier. This optimized and lightweight architecture demonstrates excellent performance, achieving an accuracy of 97.20% on the Phoenix dataset.

- **Accelerating the system on a SoC FPGA :** We successfully implemented the Features Extraction sub-stage, which involves the convolutional layers, on a PYNQ-Z1 platform using the HLS (High-Level Synthesis) approach. The implementation consumed acceptable resources on the platform. Additionally, we explored the implementation using Matlab and its dedicated toolbox for Machine and Deep Learning.

It is worth noting that our work represents the pioneering implementation of an iris recognition system, as there were no prior works in this specific area. Therefore, we did not have existing benchmarks to compare our results against.

These contributions collectively contribute to the advancement of iris recognition technology, addressing challenges such as reflections, optimizing architectures, exploring FPGA implementations and setting a baseline for future research in the domain. Our work also opens up exciting perspectives and potential future directions for research and application. Building on our achievements, we can explore various avenues to further enhance the field of iris recognition and its practical implications such as :

- **Refining Reflection Handling Techniques :**

While our approach of using Histogram Equalization to mitigate reflections in iris images has shown promising results, there is room for further improvement. Exploring advanced image processing techniques, such as deconvolution algorithms or adaptive brightness adjustment methods, could lead to even more effective reflection removal strategies. Additionally, investigating deep learning-based approaches specifically designed for handling reflections may offer new insights and improved performance in this area.

- **Multimodal Biometrics and Privacy Considerations :**

Exploring the integration of iris recognition with other biometric modalities, such as face or fingerprint, holds potential for improving authentication systems' accuracy and security. Investigating fusion techniques and developing multimodal biometric systems can provide enhanced identification capabilities while considering privacy concerns and ensuring compliance with data protection regulations.

- **Software and Hardware Optimization** : Utilizing advanced software techniques for computation optimization, particularly in the Features Extraction sub-stage that involves convolutional layers, becomes crucial.

In conclusion, iris footprints play a vital role in iris recognition systems for biometric identification and authentication. While this technology offers numerous benefits, its implementation presents several challenges. The complex image processing and pattern matching techniques used in iris recognition algorithms require substantial computational resources. Overcoming these challenges and achieving optimal performance requires efficient implementation strategies such as algorithm simplification and hardware acceleration.

Our project focused on addressing these issues, emphasizing the importance of finding effective solutions to ensure the successful integration of iris recognition technology in various applications.

Bibliography

1. KUMAR, Deepanshu; SASTRY, Mahati; MANIKANTAN, K. Iris recognition using contrast enhancement and spectrum-based feature extraction. In : 2016 *International Conference on Emerging Trends in Engineering, Technology and Science (ICETETS)*. IEEE, 2016, pp. 1–7.
2. CHANG, Yuan-Tsung; SHIH, Timothy K; LI, Yung-Hui; KUMARA, WGCW. Effectiveness evaluation of iris segmentation by using geodesic active contour (GAC). *The Journal of Supercomputing*. 2020, vol. 76, pp. 1628–1641.
3. LIU, Ming; ZHOU, Zhiqian; SHANG, Penghui; XU, Dong. Fuzzified image enhancement for deep learning in iris recognition. *IEEE Transactions on Fuzzy Systems*. 2019, vol. 28, no. 1, pp. 92–99.
4. AK, Tahir Ahmed; STELUTA, Anghelus, et al. An iris recognition system using a new method of iris localization. *International Journal of Open Information Technologies*. 2021, vol. 9, no. 7, pp. 67–76.
5. OMRAN, Maryim; ALSHEMMARY, Ebtessam N. An iris recognition system using deep convolutional neural network. In : *Journal of Physics : Conference Series*. IOP Publishing, 2020, vol. 1530, p. 012159. No. 1.
6. NANAYAKKARA, Samitha; MEEGAMA, RGM. A Review of Literature on Iris Recognition. *International Journal of Research*. 2020, vol. 9, pp. 106–120.
7. KAMBOJ, Preeti. IRIS Recognition Segmentation Signal Processing System. *Applied Science & Engineering Journal for Advanced Research (2022) ISSN (Online)*. 2022, pp. 2583–2468.
8. WEI, Yinyin; ZHANG, Xiangyang; ZENG, Aijun; HUANG, Huijie. Iris Recognition Method Based on Parallel Iris Localization Algorithm and Deep Learning Iris Verification. *Sensors*. 2022, vol. 22, no. 20, p. 7723.
9. GUO, Qiaoli; ZHENG, Junsheng. An iris recognition algorithm for identity authentication. In : 2018 *International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)*. IEEE, 2018, pp. 621–624.
10. AHMED, Hanaa M; TAHA, Mohammed A. A brief survey on modern iris feature extraction methods. *Engineering and Technology Journal*. 2021, vol. 39, no. 1, pp. 123–129.
11. LEE, Min Beom; KIM, Yu Hwan; PARK, Kang Ryoung. Conditional generative adversarial network-based data augmentation for enhancement of iris recognition accuracy. *IEEE Access*. 2019, vol. 7, pp. 122134–122152.

12. ZHAO, Tianming; LIU, Yuanning; HUO, Guang; ZHU, Xiaodong. A deep learning iris recognition method based on capsule network architecture. *IEEE Access*. 2019, vol. 7, pp. 49691–49701.
13. HAFNER, Andrej; PEER, Peter; EMERŠIČ, Žiga; VITEK, Matej. Deep iris feature extraction. In : *2021 International Conference on Artificial Intelligence in Information and Communication (ICAIIIC)*. IEEE, 2021, pp. 258–262.
15. WANG, Kuo; KUMAR, Ajay. Cross-spectral iris recognition using CNN and supervised discrete hashing. *Pattern Recognition*. 2019, vol. 86, pp. 85–98.
16. KHOTIMAH, C; JUNIATI, D. Iris recognition using feature extraction of box counting fractal dimension. In : *Journal of Physics : Conference Series*. IOP Publishing, 2018, vol. 947, p. 012004. No. 1.
17. RAFFEI, Anis Farihan Mat; DZULKIFLI, Siti Zulaikha; RAHMAN, Nur Shamsiah Abdul. Template matching analysis using neural network for mobile iris recognition system. In : *IOP Conference Series : Materials Science and Engineering*. IOP Publishing, 2020, vol. 769, p. 012024. No. 1.
18. EL-SAYED, Mohamed A; ABDEL-LATIF, Mohammed A. Iris recognition approach for identity verification with DWT and multiclass SVM. *PeerJ Computer Science*. 2022, vol. 8, e919.
19. RANA, Humayan Kabir; AZAM, Md Shafiul; AKHTAR, Mst Rashida; QUINN, Julian MW; MONI, Mohammad Ali. A fast iris recognition system through optimum feature extraction. *PeerJ Computer Science*. 2019, vol. 5, e184.
20. MITTAL, Sparsh. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*. 2016, vol. 48, no. 4, pp. 1–33.
21. MITTAL, Sparsh. A survey of FPGA-based accelerators for convolutional neural networks. *Neural computing and applications*. 2020, vol. 32, no. 4, pp. 1109–1139.
22. PAGE, Adam; JAFARI, Ali; SHEA, Colin; MOHSENIN, Tinoosh. Sparcnet : A hardware accelerator for efficient deployment of sparse convolutional networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*. 2017, vol. 13, no. 3, pp. 1–32.
23. QIU, Jiantao; WANG, Jie; YAO, Song; GUO, Kaiyuan; LI, Boxun; ZHOU, Erjin; YU, Jincheng; TANG, Tianqi; XU, Ningyi; SONG, Sen, et al. Going deeper with embedded fpga platform for convolutional neural network. In : *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*. 2016, pp. 26–35.
24. ALWANI, Manoj; CHEN, Han; FERDMAN, Michael; MILDER, Peter. Fused-layer CNN accelerators. In : *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
25. MOSS, Duncan JM; NURVITADHI, Eriko; SIM, Jaewoong; MISHRA, Asit; MARR, Debbie; SUBHASCHANDRA, Suchit; LEONG, Philip HW. High performance binary neural networks on the Xeon+ FPGA platform. In : *2017 27th International conference on field programmable logic and applications (FPL)*. IEEE, 2017, pp. 1–4.
26. LIU, Bing; ZHOU, Yanzhen; FENG, Lei; FU, Hongshuo; FU, Ping. Hybrid CNN-SVM Inference Accelerator on FPGA Using HLS. In : MDPI, 2022, vol. 11. No. 14.

35. ESTREBOU, César Armando; FLEMING, Martín; SAAVEDRA, Marcos David; ADRA, Federico. A neural network framework for small microcontrollers. In : *XXVII Congreso Argentino de Ciencias de la Computación (CACIC) (Modalidad virtual, 4 al 8 de octubre de 2021)*. 2021.
36. AHMED, Hossam O; GHONEIMA, Maged; DESSOUKY, Mohamed. Concurrent MAC unit design using VHDL for deep learning networks on FPGA. In : *2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE, 2018, pp. 31–36.
37. PODILI, Abhinav; ZHANG, Chi; PRASANNA, Viktor. Fast and efficient implementation of convolutional neural networks on FPGA. In : *2017 IEEE 28Th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 11–18.
38. ZHANG, Chen; LI, Peng; SUN, Guangyu; GUAN, Yijin; XIAO, Bingjun; CONG, Jason. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In : *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 2015, pp. 161–170.
39. MITTAL, Sparsh. A survey of FPGA-based accelerators for convolutional neural networks. *Neural computing and applications*. 2020, vol. 32, no. 4, pp. 1109–1139.
44. RAFAEL C. GONZALEZ, Richard E. Woods. Digital Image Processing. In : Pearson Education, Inc, 2008.
50. ASHFAQ, Muniba; MINALLAH, Nasru; ULLAH, Zahid; AHMAD, Arbab Masood; SAEED, Aamir; HAFEEZ, Abdul. Performance analysis of low-level and high-level intuitive features for melanoma detection. *Electronics*. 2019, vol. 8, no. 6, p. 672.
51. PS, Shijin Kumar; VS, D. Extraction of texture features using GLCM and shape features using connected regions. *International journal of engineering and technology*. 2016, vol. 8, no. 6, pp. 2926–2930.
53. ASHFAQ, Muniba; MINALLAH, Nasru; ULLAH, Zahid; AHMAD, Arbab Masood; SAEED, Aamir; HAFEEZ, Abdul. <https://www.intel.com/>. *Electronics*. 2019, vol. 8, no. 6, p. 672.
56. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep residual learning for image recognition. In : *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
60. Proceedings of the IEEE conference on computer vision and pattern recognition. In : [n.d.]. Available also from : <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Block-Level-Control-Protocols>.

Webography

14. DAVID, Davis. *Random Forest classifier tutorial : How to use tree-based algorithms for machine learning*. freeCodeCamp.org, 2020. Available also from : <http://www.freecodecamp.org/news/how-to-use-the-tree-based-algorithm-for-machine-learning/>.
27. MARK HUDSON BEALE Martin T.H, Howard B.D. *Deep Learning Toolbox*. 2020. Available also from : <https://www.mathworks.com/products/deep-learning.html>.
28. MARK HUDSON BEALE, Martin T.H. *Deep Learning HDL Toolbox*. 2020. Available also from : <https://www.mathworks.com/products/deep-learning-hdl.html>.
29. *MATLAB for Artificial Intelligence*. 1994-2023. Available also from : https://www.mathworks.com/?s_tid=gn_logo.
30. *Convert Simulink Model to Fixed-Point*. 2020. Available also from : <https://www.mathworks.com/help/stats/deploy-neural-network-regression-model-to-fpga-platform.html>.
31. *Prepare Simulink Model for HDL Code Generation*. 2020. Available also from : <https://www.mathworks.com/help/stats/deploy-neural-network-regression-model-to-fpga-platform.html>.
32. *Deep Learning Processor IP Core Generation for Custom Board*. 2020. Available also from : <https://www.mathworks.com/help/deep-learning-hdl/ug/define-custom-board-and-reference-design-for-dl-ip-core-workflow.html>.
33. *Generate IP Core and Bitstream*. 2020. Available also from : <https://www.mathworks.com/help/hdlcoder/generate-ip-core-and-bitstream.html>.
34. *Prototype Deep Learning Networks on FPGA*. 2020. Available also from : <https://shorturl.at/bcgLW>.
40. PATTERN RECOGNITION, National Laboratory of. [N.d.]. Available also from : <http://biometrics.idealtest.org/>.
41. [N.d.]. Available also from : https://www4.comp.polyu.edu.hk/~csajaykr/IITD/Database_Iris.htm.
42. 193070017. *MMU Iris dataset*. 2020. Available also from : <https://www.kaggle.com/datasets/naureenmohammad/mmu-iris-dataset>.
43. DEPT. COMPUTER SCIENCE, Palacky University in Olomouc. *Phoenix dataset*. [N.d.]. Available also from : <https://phoenix.inf.upol.cz/iris/>.
45. [N.d.]. Available also from : https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html.

46. BAELDUNG, Written by : *How to use Gabor filters to generate features for machine learning*, 2023. Available also from : <https://www.baeldung.com/cs/ml-gabor-filters>.
47. [N.d.]. Available also from : <https://www.mathworks.com/discovery/wavelet-transforms.html>.
48. TALEBI, Shawhin. *The wavelet transform*. Towards Data Science, 2023. Available also from : <https://towardsdatascience.com/the-wavelet-transform-e9cfa85d7b34>.
49. *SVM Classification*. [N.d.]. Available also from : <https://scikit-learn.org/stable/modules/svm.html#svm-classification>.
52. MISHRA, Mayank. *Convolutional Neural Networks, explained*. Towards Data Science, 2020. Available also from : <https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>.
54. MISHRA, Mayank. *Call functions*. Towards Data Science, 2020. Available also from : <https://www.freecodecamp.org/news/how-to-automate-call-graph-creation/#what-is-a-call-graph>.
55. MISHRA, Mayank. *AXI reference*. Towards Data Science, 2020. Available also from : https://support.xilinx.com/s/article/1053914?language=en_US.
57. MISHRA, Mayank. *AXI reference*. Towards Data Science, 2020. Available also from : <https://docs.xilinx.com>.
58. MISHRA, Mayank. *AXI reference*. Towards Data Science, 2020. Available also from : <https://www.aldec.com/en/company/blog/122--introduction-to-axi-protocol>.
59. MISHRA, Mayank. *PYNQ docs*. Towards Data Science, 2020. Available also from : https://pynq.readthedocs.io/en/latest/pynq_overlays.html.