MINISTRY OF EDUCATION AND TRAINING
HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION

GRADUATION THESIS

MAJOR: COMPUTER ENGINEERING TECHNOLOGY

FACIAL LANDMARK DETECTION ON MULTIPLE PLATFORMS USING QUANTIZED MOBILENETV2 FOR EDGE AI APPLICATIONS

INSTRUCTOR PHAM VAN KHOA

STUDENT DO MANH DUNG

DANG TRUNG NGHIA

HO CHI MINH CITY – 07/2025

HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY OF INTERNATIONAL EDUCATION

GRADUATION PROJECT

FACIAL LANDMARK DETECTION ON MULTIPLE PLATFORMS USING QUANTIZED MOBILENETV2 FOR EDGE AI APPLICATIONS

ĐỖ MẠNH DŨNG

Student ID: 21119301

ĐẶNG TRUNG NGHĨA

Student ID: 21119313

Major: COMPUTER ENGINEERING TECHNOLOGY

Advisor: PHẠM VĂN KHOA, PhD.

HO CHI MINH CITY – 07/2025

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION
FACULTY OF INTERNATIONAL EDUCATION

GRADUATION PROJECT

FACIAL LANDMARK DETECTION ON MULTIPLE PLATFORMS USING QUANTIZED MOBILENETV2 FOR EDGE AI APPLICATIONS

ĐỖ MẠNH DŨNG
Student ID: 21119301
ĐẶNG TRUNG NGHĨA
Student ID: 21119313
Major: COMPUTER ENGINEERING TECHNOLOGY
Advisor: PHẠM VĂN KHOA, PhD.

HO CHI MINH CITY – 07/2025

THE SOCIALIST REPUBLIC OF VIETNAM
Independence – Freedom– Happiness

Ho Chi Minh City, July …, 2025

GRADUATION PROJECT ASSIGNMENT

Student name: Đỗ Mạnh Dũng	Student ID: 21119301
Student name: Đặng Trung Nghĩa	Student ID: 21119313
Major: Computer Engineering Technology	Class: 21119FIE
Advisor: Phạm Văn Khoa, PhD.	Phone number: 0397079864
Date of assignment: _____	Date of submission: _____

1. Project title: Facial Landmark Detection on Multiple Platforms using Quantized MobileNetV2 for Edge AI Applications

2. Initial materials provided by the advisor:

3. Content of the project:

- Build and train a lightweight facial landmark detection model using MobileNetV2 architecture and the WFLW dataset.
- Integrate a pose-estimation auxiliary branch and train the model with a weighted pose- and class-aware loss function.
- Perform Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) to compress the model to INT8 format using Vitis AI.
- Deploy and evaluate the quantized model on multiple platforms (CPU, GPU, Raspberry Pi, Jetson Nano, FPGA KV260) for real-time performance and energy efficiency.

4. Final product: _____

CHAIR OF THE PROGRAM	**ADVISOR**
(Sign with full name)	*(Sign with full name)*

THE SOCIALIST REPUBLIC OF VIETNAM
Independence – Freedom– Happiness

Ho Chi Minh City, July …, 2025

ADVISOR'S EVALUATION SHEET

Student name: Đỗ Mạnh Dũng Student ID: 21119301

Student name: Đặng Trung Nghĩa Student ID: 21119313

Major: Computer Engineering Technology

Project title: Facial Landmark Detection On Multiple Platforms Using Quantized MobilenetV2 for Edge AI Applications

Advisor: Phạm Văn Khoa, PhD.

EVALUATION

1. Content of the project:

- Build and train a lightweight facial landmark detection model using MobileNetV2 architecture and the WFLW dataset.
- Integrate a pose-estimation auxiliary branch and train the model with a weighted pose- and class-aware loss function.
- Perform Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) to compress the model to INT8 format using Vitis AI.
- Deploy and evaluate the quantized model on multiple platforms (CPU, GPU, Raspberry Pi, Jetson Nano, FPGA KV260) for real-time performance and energy efficiency.

2. Strengths:

.....
.....
.....

3. Weaknesses:

.....
.....
.....

4. Approval for oral defense? *(Approved or denied)*

.....

5. Overall evaluation: (Excellent, Good, Fair, Poor)

.....

6. Mark:…….(*in words*:.....)

Ho Chi Minh City, July …, 2025

ADVISOR

(Sign with full name)

THE SOCIALIST REPUBLIC OF VIETNAM
Independence – Freedom– Happiness

Ho Chi Minh City, July …, 2025

PRE-DEFENSE EVALUATION SHEET

Student name: Đỗ Mạnh Dũng Student ID: 21119301

Student name: Đặng Trung Nghĩa Student ID: 21119313

Major: Computer Engineering Technology

Project title: Facial Landmark Detection On Multiple Platforms Using Quantized MobilenetV2 For Edge AI Applications

Name of Reviewer: .....

EVALUATION

1. Content and workload of the project

- Build and train a lightweight facial landmark detection model using MobileNetV2 architecture and the WFLW dataset.
- Integrate a pose-estimation auxiliary branch and train the model with a weighted pose- and class-aware loss function.
- Perform Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) to compress the model to INT8 format using Vitis AI.
- Deploy and evaluate the quantized model on multiple platforms (CPU, GPU, Raspberry Pi, Jetson Nano, FPGA KV260) for real-time performance and energy efficiency.

2. Strengths:

.....
.....
.....

3. Weaknesses:

.....
.....
.....

4. Approval for oral defense? *(Approved or denied)*

.....

5. Overall evaluation: *(Excellent, Good, Fair, Poor)*

.....

6. Mark:…….(*in words*:.....)

Ho Chi Minh City, July …, 2025

REVIEWER

(Sign with full name)

THE SOCIALIST REPUBLIC OF VIETNAM
Independence – Freedom– Happiness

EVALUATION SHEET OF DEFENSE COMMITTEE MEMBER

Student name: ..... Student ID: .....

Student name: ..... Student ID: .....

Student name: ..... Student ID: .....

Major: .....

Project title: .....

.....

Name of Defense Committee Member:

.....

EVALUATION

1. Content and workload of the project

- Build and train a lightweight facial landmark detection model using MobileNetV2 architecture and the WFLW dataset.
- Integrate a pose-estimation auxiliary branch and train the model with a weighted pose- and class-aware loss function.
- Perform Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) to compress the model to INT8 format using Vitis AI.
- Deploy and evaluate the quantized model on multiple platforms (CPU, GPU, Raspberry Pi, Jetson Nano, FPGA KV260) for real-time performance and energy efficiency.

2. Strengths:

.....

.....

.....

3. Weaknesses:

.....

.....

.....

4. Overall evaluation: *(Excellent, Good, Fair, Poor)*

.....

5. Mark:…….(*in words*:.....)

Ho Chi Minh City, July …, 2025

COMMITTEE MEMBER

(Sign with full name)

DISCLAIMER

The implementation group hereby formally declares that the research and application presented in this thesis represent the outcome of our independent and original work. The methodologies, analytical techniques, and proposed solutions have been developed based on our professional expertise and scientific research capabilities. All aspects of the research process, including data collection, preprocessing, model design, training, and system deployment, have been conducted with scientific rigor and methodological precision.

We categorically affirm that no portion of this work has been plagiarized from any published materials without appropriate citation and acknowledgment. All reference materials have been properly documented and credited according to established academic standards. We have maintained strict adherence to ethical principles and academic integrity throughout the entire research and implementation process.

We acknowledge full responsibility for the accuracy and authenticity of all content presented in this thesis. Any errors or omissions that may have occurred during the implementation process are entirely our responsibility, and we commit to upholding the highest standards of academic and research integrity.

The research findings and results presented in this thesis demonstrate our systematic approach and innovative methodology in the field of license plate recognition. We anticipate that this research will make a meaningful contribution to the advancement of the field and provide a foundation for subsequent research endeavors.

The knowledge and experience gained through this project have been invaluable to our professional development. We are confident that the technical solutions and research outcomes presented herein reflect our dedicated commitment to scientific excellence and innovation in computer vision and pattern recognition applications.

Students

Do Manh Dung

Dang Trung Nghia

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to Ph.D. Phạm Văn Khoa, our esteemed supervisor and mentor, for his invaluable guidance and unwavering support throughout this research project. His profound expertise in hardware-software co-design and FPGA-based AI acceleration has been instrumental in steering our research toward meaningful contributions and empowering us to navigate complex technical challenges with confidence.

This work focused on developing an energy-efficient Edge AI pipeline for real-time facial landmark detection with a comprehensive evaluation across multiple hardware platforms, including GPU, CPU, FPGA, and embedded systems. Under Ph.D. Khoa's expert supervision, we successfully implemented systematic model compression through Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT), achieving INT8 precision with minimal accuracy degradation. The successful deployment on the Kria KV260 platform, achieving high FPS with superior energy efficiency, directly reflects his exceptional mentorship.

Ph.D. Khoa's expertise in quantization strategies and hardware-aware optimization proved invaluable in transforming our theoretical framework into a practical, high-performance prototype. His constructive feedback on performance evaluation and comparative analysis significantly enhanced the rigor of our research contributions. We are deeply grateful for his continuous encouragement and technical guidance, which played a pivotal role in achieving our thesis objectives. His constructive feedback on multi-platform performance evaluation and comparative analysis significantly enhanced the rigor of our research contributions.

Thank you, Ph.D. Phạm Văn Khoa, for believing in our research vision and guiding us through this journey of scientific innovation and discovery..

Sincerely!

STUDENTS

ABSTRACT

Real-time applications such as driver monitoring, face detection, and expression analysis have become increasingly critical in modern life. Deploying Facial Landmark Detection systems to make assessments of state and emotions of user require real-time processing capabilities with stringent power constraints for edge deployment. This work presents a comprehensive evaluation and optimization of facial landmark detection networks across multiple hardware platforms, with particular focus on FPGA implementation for energy-efficient edge computing applications. We demonstrate an end-to-end development workflow that transforms the PyTorch PFLD network into a hardware-accelerated CNN optimized for the Xilinx Kria KV260 platform. Our methodology begins with systematic model compression using Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) pipelines in Vitis AI, converting all weights and activations to 8-bit fixed-point representation. This quantization strategy reduces the model footprint to 1.65 MB achieving 4× compression with minimal accuracy degradation (less than 1%). Performance evaluation was conducted across five hardware platforms: RTX 4060 GPU, i9 CPU, Xilinx Kria KV260 FPGA, Jetson Nano 2GB, and Raspberry Pi 4B. The FPGA implementation demonstrates superior energy efficiency with 10 FPS/W approximately 2.5× more efficient than the RTX 4060 (3.91 FPS/W), 7× more efficient than the Intel i9 (1.35 FPS/W), and over 11× more efficient than embedded alternatives (Jetson Nano: 0.90 FPS/W, Raspberry Pi: 0.79 FPS/W). The FPGA implementation achieves real-time inference at 30 FPS while maintaining competitive accuracy (91.74% vs 92.42% for other platforms) with a total power consumption of just 3W. This represents a ten-fold energy efficiency improvement compared to CPU implementation, making it ideal for battery-powered edge applications.

Keywords: Facial Landmark Detection, Mobilenetv2, FPGA, Quantization-Aware Training, Post Training Quantization, Vitis AI, Edge Computing.

TABLE OF CONTENT

DISCLAIMER i
ACKNOWLEDGEMENT ii
ABSTRACT iii
LIST OF FIGURES vi
LIST OF TABLES viii
LIST OF ABBREVIATIONS ix
CHAPTER 1: INTRODUCTION 1
1.1. Problem Statement 1
1.2. The Role of Research for Science and Practice 2
1.3. Objective 3
1.4. Research Value and Scope 4
1.5. Thesis Outline 5
CHAPTER 2: FUNDAMENTAL CONCEPT 6
2.1. Convolutional Neural Network (CNN) 6
2.1.1. Convolutional Neural Network in Facial Landmark Detection 6
2.1.2. Facial Landmark Dataset 6
2.2. Lightweight CNN Architecture: MobileNetV2 ..... **9**
2.2.1. Core Architectural Features 10
2.2.2. Computational Complexity and Parameter Efficiency 10
2.2.3. Flexibility and Hardware Scaling 11
2.2.4. Structured Pruning and Sparse Optimization 12
2.3. Quantization **12**
2.3.2. Post-Training Quantization (PTQ) 13
2.3.3. Quantization-Aware Training (QAT) 14
2.4. Overview Vitis AI **15**
2.4.1. Vitis AI includes the following features: 15
2.4.2. Quantization In Vitis AI 16
2.5. Edge Computing **19**
2.6. FPGA **20**
2.7. Pytorch Framework **23**
2.8. Overview of Hardware Platforms for Comparison **25**

2.8.1. NVIDIA RTX 4060 GPU 25
2.8.2. Intel Core i9-13900K CPU 26
2.8.3. NVIDIA Jetson Nano 2GB 27
2.8.4. Raspberry Pi 4 Model B 27
2.9. Related works **28**
CHAPTER 3: IMPLEMENTATION **30**
3.1. System Overview **30**
3.1.1 Data Preparation and Augmentation 31
3.1.2. Model Architecture 33
3.1.3. Loss Function 36
3.1.4. Training Configuration 39
3.1.5. Algorithms Explaining the Complete Pipeline 40
3.1.6. Face and Facial Landmark Detection 41
3.2. Quantization **42**
3.2.1. Post-Training Quantization (PTQ) 45
3.2.2. Quantization-Aware Training (QAT) 48
CHAPTER 4: RESULT & DISCUSSION **53**
4.1. Training Model **53**
4.1.1. Experimental Result 53
4.1.2. Evaluation 54
4.2. Quantization **54**
4.2.1. Results 54
4.2.2. Evaluation 56
4.3. Target Platforms Deployment **56**
4.3.1. Experimental Result 63
4.3.2. Evaluation 65
CHAPTER 5: CONCLUSION AND FUTURE WORK **66**
5.1. Conclusion **66**
5.2. Future Work **67**
REFERENCES **69**

LIST OF FIGURES

Figure 2.1. Face region with 98 landmarks [6] 7
Figure 2.2. The architecture of MobileNetV2 DNN [7] 11
Figure 2.3. Quantization strategies with Post-Training Quantization (PTQ) in the left and Quantization-Aware Training (QAT) in the right. 13
Figure 2.4. Vitis AI Integrated Development Environment [20] 15
Figure 2.5: Pruning 17
Figure 2.6. Prunning process flow 18
Figure 2.7. Evolution of accuracy and parameter reduction over iterations in the pruning process 18
Figure 2.8. Quantization process 19
Figure 2.9. Edge Computing 20
Figure 2.10. A typical internal structure of an FPGA [40] 22
Figure 2.11. Pytorch Framwork [41] 24
Figure 2.12. RTX 4060 GPU 26
Figure 2.13. Intel Core i9-13900K CPU 26
Figure 2.14. NVIDIA Jetson Nano 2GB 27
Figure 2.15. Raspberry Pi 4 Model B 28
Figure 3.1. System Overview [42] 30
Figure 3.2. Flow diagram 31
Figure 3.3. 98 Facial Landmarks Images 31
Figure 3.4. Photometric transformation examples 33
Figure 3.5. Pre-processing and Training on Dataset Workflow 40
Figure 3.5. Inference facial landmark detection Pipeline 42
Figure 3.6. Pruning and Quantization Flow 42
Figure 3.7. Quantization Workflow 44
Figure 3.8. PTQ Model Flow 45
Figure 3.9. QAT Model Flow 49
Figure 4.1: Accuracy of MobileNetV2 over epochs 53
Figure 4.2: Visualization images after quantization by QAT and PTQ with green landmarks is groud truth and blue landmarks is predicted 56

Figure 4.3. System workflow 58
Figure 4.4. Raspberry Setup Environment 59
Figure 4.5. Jetson Nano Setup Environment 59
Figure 4.6. KV260 Setup Environment 60
Figure 4.7. Jetson Nano Inference 61
Figure 4.8. Raspberry Inference 62
Figure 4.9. KV260 Inference 62

LIST OF TABLES

Table 1: Dataset comparison 8
Table 2: Vitis AI supports quantization..... 16
Table 3: Related Works Comparison 28
Table 4: The backbone network configuration..... 34
Table 5: The Auxiliary Branch Configuration 36
Table 6: Quantized method results 55
Table 7: Comparison and evaluation of PTQ vs. QAT 56
Table 8: Voltage of platforms..... 57

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
AFLW	Annotated Factual Landmarks in the Wild
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANNs	Artificial Neural Networks
API	Application Programming Interface
AR	Augmented Reality
AXI	Advanced eXtensible Interface
BN	Batch Normalization
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DNNs	Deep Neural Networks
DPU	Deep Learning Processor Unit
FC	Fully Connected
FP	Floating Point
FP32	32-bit Floating Point
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
FPU	Floating point unit
GPU	Graphics Processing Unit
HDMI	High-Definition Multimedia Interface
HLS	High-Level Synthesis
HW	Hardware
IC	Integrated Circuit
INT8	8-bit Integer
IoT	Internet of Things
IP	Intellectual Property
LUT	Lookup Table
ML	Machine Learning
MTCNN	Multi-Task Cascaded Convolutional Networks
NLP	Natural Language Processing
NME	Normalized Mean Error
ONNX	Open Neural Network Exchange
OS	Operation System
PE	Processing Element
PFLD	Practical Facial Landmark Detector
PTQ	Post Training Quantization
QAT	Quantization-aware Training

QM	Quantized Model
QNNs	Quantized Neural Network
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RGB	Red – Green – Blue
RNNs	Recurrent Neural Networks
RTL	Register Transfer Level
SIMD	Single-Instruction Multiple-Data
SoC	System on Chip
STE	Straight Through Estimator
TF	TensorFlow
USB	Universal Serial Bus
VR	Virtual Reality
WFLW	Wider Facial Landmarks in the Wild
YOLO	You Only Look Once

CHAPTER 1: INTRODUCTION

1.1. Problem Statement

The Internet of Things has evolved significantly, with an increasing number of devices being connected to networks and equipped with small computing capabilities [1]. Traditionally, these devices use cloud computing services to process data received from their sensors. However, many emerging applications require computation to be performed at the edge - the connection point between the network and the real world. This necessity has given rise to edge computing as a critical paradigm shift. Edge computing focuses on processing data efficiently at the network's edge rather than relying solely on cloud computing [2]. This approach addresses several critical needs: faster data processing speeds compared to using broadband to transmit data to the cloud and waiting for computation results, and the transformation from data consumers to data producers as more people generate new data rather than simply consuming it on their devices. A prime example of this need is autonomous vehicles, which generate massive amounts of data every second that must be processed in real-time, making cloud service dependency impractical [3].

Within this edge computing context, one of the most important use cases is for traffic accidents caused by driver drowsiness and fatigue represent a critical challenge for road safety. These incidents account for 3–30% of global accidents and approximately 10 - 20% of severe crashes in Vietnam, resulting in thousands of collisions and hundreds of fatalities annually [4]. The urgency of this problem aligns perfectly with edge computing requirements, as driver monitoring systems demand real-time processing with minimal latency to provide immediate safety interventions. Existing solutions, such as steering wheel sensors, basic surveillance cameras, or algorithms running on CPU/GPU platforms, are often limited by high processing latency, significant power consumption, and costly hardware. These limitations make deployment on resource-constrained embedded systems typical of edge computing environments particularly challenging. In response to these challenges, there is an urgent need for a non-intrusive, real-time monitoring approach capable of operating on low-power edge platforms. Facial landmark detection offers a promising solution by accurately identifying key facial features such as eyelids, mouth corners, and head pose. This method enables real-time analysis of blink frequency, mouth opening degree, and head tilt to detect early signs of drowsiness. However, traditional convolutional neural networks (CNNs) typically demand substantial computational resources and memory, posing difficulties for implementation on FPGAs with limited logic capacity and bandwidth [5]- common constraints in edge computing scenarios.

This research proposes a lightweight CNN model specifically tailored for facial landmark detection and optimized for edge deployment. The model incorporates quantization and pruning techniques to significantly reduce model size and computational

load, making it suitable for resource-constrained edge devices. The proposed architecture uses the parallel processing capabilities of FPGAs to achieve real-time inference with minimal latency and optimal power efficiency. The system is designed for deployment on embedded edge platforms, meeting stringent requirements for cost and energy efficiency while eliminating the need for cloud connectivity during critical safety operations. This approach directly addresses the edge computing paradigm's core principles: processing data locally for immediate response, reducing bandwidth requirements, and ensuring system reliability independent of network connectivity.

Upon completion, the system will provide instantaneous facial landmark detection that serves as the foundation for drowsiness alerts, embodying the edge computing principle of real-time local processing. Additionally, the system enables broader edge applications, including attention monitoring, emotion analysis, AR/VR calibration, and facial biometrics. The proposed solution not only enhances road safety through effective driver state monitoring but also establishes a foundation for deploying real-time computer vision systems on edge devices, optimized for performance, cost-effectiveness, and local processing capabilities. This work contributes to the growing ecosystem of edge computing applications that prioritize immediate response times and autonomous operation, essential characteristics for safety-critical systems in connected vehicle environments.

1.2. The Role of Research for Science and Practice

The proposed study advances scientific knowledge in the fields of Edge Computing, Hardware-aware Machine Learning, and Computer Vision by demonstrating systematic evaluation and optimization of facial landmark detection across multiple hardware platforms. From a scientific perspective, it provides a comprehensive methodology for comparing performance trade-offs between different computing architectures (GPU, CPU, FPGA, embedded systems) in the context of real-time AI applications. Our work establishes empirical benchmarks for energy efficiency (FPS/W) across diverse hardware platforms, quantifying the relationship between computational performance, power consumption, and inference accuracy. The research contributes concrete evidence that FPGA-based implementations can achieve superior energy efficiency compared to traditional GPU and CPU approaches, while maintaining competitive accuracy levels. The systematic evaluation of model compression techniques, including Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT), provides valuable insights into accuracy-efficiency trade-offs for edge AI deployment. The study fills a critical gap in comparative hardware evaluation for computer vision applications, offering empirical data on how different platforms perform under real-time constraints and resource limitations. These findings can guide future research in selecting appropriate hardware architectures for specific edge computing scenarios and inform the development of platform-aware optimization strategies.

In practical terms, this research addresses the urgent need for energy-efficient driver drowsiness detection systems in the automotive industry. With traffic accidents caused by driver fatigue accounting for 3-30% of global incidents and 10-20% of serious crashes in Vietnam, the development of low-power, real-time monitoring solutions has a significant societal impact. The demonstrated FPGA implementation, consuming only 3W while maintaining 30 FPS processing capability, enables practical deployment in battery-powered or automotive-grade systems. The comparative evaluation framework provides industry practitioners with concrete guidance for hardware selection in edge AI applications. The 10× energy efficiency improvement demonstrated by the FPGA platform over CPU implementations makes it particularly suitable for always-on monitoring systems where battery life and thermal management are critical concerns. Beyond automotive safety, the energy-efficient landmark detection approach has broad applications in human-computer interaction, telemedicine, and mobile computing where continuous vision processing is required under power constraints. The methodology for systematic platform evaluation can be applied to other computer vision tasks, providing a template for optimizing AI deployment across different hardware architectures. By establishing clear performance benchmarks and energy efficiency metrics, this research enables informed decision-making for edge AI deployment in resource-constrained environments, contributing to the broader adoption of AI-powered safety and monitoring systems in real-world applications.

1.3. Objective

This thesis aims to develop a comprehensive Edge-AI pipeline for real-time facial landmark detection optimized for FPGA deployment. The primary objective is to create an energy-efficient, hardware-accelerated solution for driver drowsiness detection that operates independently on edge devices without cloud connectivity. The work establishes a complete development workflow transforming the PyTorch PFLD network into an optimized hardware accelerator. The network undergoes systematic optimization through Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) using Vitis AI, converting weights and activations to 8-bit fixed-point representation to achieve significant model compression with minimal accuracy degradation. The quantized model is translated into the inference engine for real-time inference.

The thesis seeks to demonstrate the superior energy efficiency of FPGA implementations compared to conventional GPU, CPU, and embedded computing platforms. The goal is to achieve real-time inference capabilities while maintaining minimal power consumption, making the solution suitable for battery-powered automotive applications and continuous monitoring systems. Systematic performance evaluation across multiple hardware platforms will establish empirical benchmarks for edge AI deployment decisions. Beyond addressing driver drowsiness detection, which causes 3-30% of global traffic accidents, the pipeline is designed to serve as a versatile foundation

for edge-vision applications including human-computer interaction, telemedicine, AR/VR calibration, and biometric authentication. The resulting system embodies core edge computing principles: local processing for immediate response, minimal power consumption, and autonomous operation independent of network connectivity. This work provides a practical template for deploying quantized CNNs in resource-constrained environments while advancing energy-efficient edge AI applications.

1.4. Research Value and Scope

This research advances the scientific understanding of edge computing through systematic evaluation and optimization of facial landmark detection across multiple hardware architectures, establishing empirical benchmarks for energy efficiency (FPS/W) across diverse platforms including GPU, CPU, FPGA, and embedded systems while providing quantifiable evidence of the relationship between computational performance, power consumption, and inference accuracy in real-time AI applications. The study contributes significant scientific knowledge by demonstrating that FPGA-based implementations can achieve superior energy efficiency compared to traditional GPU and CPU approaches while maintaining competitive accuracy levels, with systematic evaluation of model compression techniques including Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT), providing valuable insights into accuracy-efficiency trade-offs for edge AI deployment. From a practical standpoint, this research addresses the urgent societal need for energy-efficient such as: expression analasis or driver drowsiness detection systems in the automotive industry, where traffic accidents caused by driver fatigue account for 3-30% of global incidents and 10-20% of serious crashes in Vietnam, making the development of low-power, real-time monitoring solutions critically important for societal impact. The demonstrated FPGA implementation consumes only 3W while maintaining 30 FPS processing capability, enabling practical deployment in battery-powered or automotive-grade systems, with the 10× energy efficiency improvement over CPU implementations making it particularly suitable for always-on monitoring systems where battery life and thermal management are critical concerns.

The research operates within defined boundaries focusing on facial landmark detection as the primary computer vision task, FPGA as the primary target hardware platform for optimization, real-time processing requirements as the primary performance constraint, and energy efficiency as the primary optimization criterion within edge computing deployment contexts, ultimately providing a practical template for deploying quantized CNNs in resource-constrained environments while advancing energy-efficient edge AI applications and establishing a foundation for future work in hardware-aware machine learning and edge computing optimization.

1.5. Thesis Outline

The project consists of five main chapters, each detailing specific aspects as follows:

CHAPTER 1 - INTRODUCTION:

Introduces the context and motivation for deploying real-time facial landmark detection on edge platforms. It discusses the challenges of driver drowsiness detection and the need for energy-efficient AI models. The chapter outlines the research objectives, scope, and contributions of the thesis.

CHAPTER 2 – FUNDAMENTAL CONCEPT:

Provides theoretical background on key technologies, including Convolutional Neural Networks (CNNs), MobileNetV2 architecture, facial landmark datasets, quantization techniques (PTQ, QAT), and the Vitis AI framework. It also explores edge computing principles, FPGA architecture, and relevant hardware platforms for deployment.

CHAPTER 3 – IMPLEMENTATION:

Describes the complete development workflow: from data preprocessing and augmentation to model training and architectural design. It includes the quantization process (PTQ and QAT), loss function formulation, training strategies, and model exportation for FPGA deployment using PyTorch and Vitis AI..

CHAPTER 4 – RESULTS AND DISCUSSION:

Presents experimental results from model training, quantization impact, and deployment across five hardware platforms: GPU, CPU, FPGA, Jetson Nano, and Raspberry Pi. Metrics such as accuracy, FPS, power consumption, and energy efficiency are analyzed. Comparative evaluations highlight the trade-offs between PTQ and QAT methods and between platforms.

CHAPTER 5 - CONCLUSION AND FUTURE WORK:

Summarizes the key findings: the effectiveness of MobileNetV2 + QAT on FPGA for energy-efficient landmark detection, especially for automotive applications. It outlines future directions, including deeper quantization, custom RTL accelerators, ASIC integration, and expanded use cases like telemedicine and human-computer interaction.

CHAPTER 2: FUNDAMENTAL CONCEPT

2.1. Convolutional Neural Network (CNN)

2.1.1. Convolutional Neural Network in Facial Landmark Detection

Convolutional Neural Networks (CNNs) are a powerful class of deep learning models well suited for spatially structured data, such as facial images. In facial landmark detection, CNNs can learn hierarchical representations of facial features through layers of convolutions, activations, and pooling. By applying small filters across an image, CNNs are capable of identifying key visual structures such as eye contours, nose edges, and mouth corners.

Two crucial properties of human faces that CNN-based models exploit are symmetry and fixed spatial relationships. Facial symmetry allows the network to generalize patterns across the midline of the face, enhancing robustness against occlusion and lighting imbalance. Fixed geometric relationships, like the consistent distance between the nose and the eyes serve as implicit constraints that help the network predict anatomically plausible landmark positions. These properties are reinforced through architectural components such as weight sharing, pooling layers, and data augmentation strategies (e.g., image flipping or rotation).

2.1.2. Facial Landmark Dataset

❖ **98 landmarks Dataset**

To train models capable of precise landmark localization under real-world conditions, a dataset with rich annotations and high variability is essential. The Wider Facial Landmarks in the Wild (WFLW) dataset fulfills this requirement by providing:

- **Total images:** 10,000 facial images which have 7,500 for training and 2,500 for testing.
- **Annotations:** 98 manually labeled facial landmarks.
- **Regions covered:** Eyes, eyebrows, nose, lips, jawline, facial contour.
- **Annotated attributes**: Occlusion, pose, illumination, blur , expression and makeup.

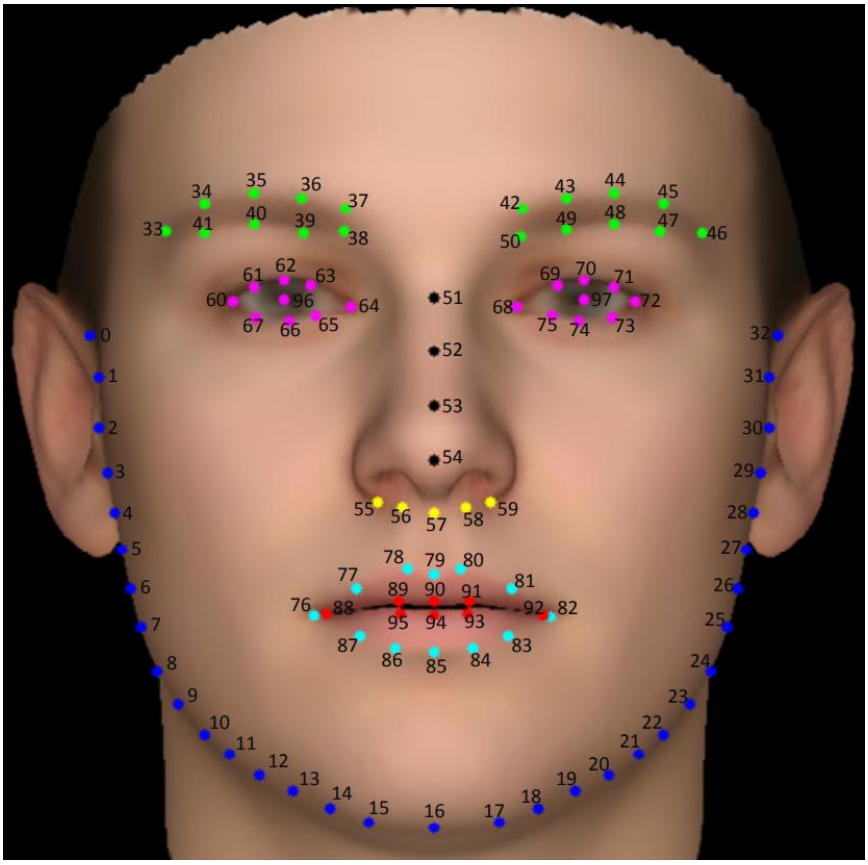

Figure 2.1. Face region with 98 landmarks [6]

The motivation behind WFLW is to serve as a more realistic and challenging benchmark that reflects the complexity of facial landmark detection in practical scenarios. Applications benefiting from this dataset include:

- Drowsiness detection
- Head pose estimation
- Emotion recognition
- Augmented reality (AR) facial tracking
- Driver monitoring systems

This dataset is particularly suitable for fatigue and drowsiness detection, as the dense landmark layout enables accurate measurement of eye aspect ratio (EAR), mouth aspect ratio (MAR), head tilt, and subtle facial expressions.

❖ **Comparison between WFLW and 300W**

Table 1: Dataset comparison

Criterion	WFLW (98 landmarks)	300W (68 landmarks)
Number of Landmarks	98 points – includes detailed annotations of eyes, mouth, jawline, and nose bridge	68 points – standard coverage, lacks detail in facial contour and forehead region
Eye & Mouth Detail	High – supports precise computation of EAR, MAR, yawning, and prolonged eye closure	Moderate – suitable for EAR, but limited precision for MAR and mouth expressions
Data Diversity	Extensive – includes occlusion, blur, pose, illumination, expression, and makeup	Present – but fewer explicitly categorized challenges
Dataset Size	10,000 images (7,500 train / 2,500 test)	3,800 images (3,148 train / 689 test)
Suitability for Drowsiness Tasks	Highly suitable – enables analysis of fatigue indicators: eye state, yawning, head tilt	Basic suitability – limited expression and head pose variability
Drowsiness Detection Accuracy	92–98% (with CNN-based models using WFLW landmarks)	85–90% (with similar models using 300W landmarks)

Given its richer annotations and robustness across variable conditions, WFLW is selected in this work as the primary dataset for training and evaluation.

❖ **Dataset Augmentation**

Data augmentation represents a methodology employed to enhance dataset diversity without requiring the acquisition of additional original data. This approach functions by implementing diverse modification techniques to existing datasets, generating new, altered versions that improve a model's ability to generalize effectively. Throughout this discussion, we will explore data augmentation concepts in greater detail.

Image-based data augmentation operates through the application of multiple transformation methods to source images. These modification techniques are implemented

while preserving the original data labels, simultaneously producing enhanced training datasets. Several of these transformation approaches include:

Geometric Transformations

- Geometric transformations alter the spatial properties of an image. It includes:
- Rotation: It rotates the image to a certain angle like 90° or 180°.
- Flipping: It flips the image horizontally or vertically.
- Scaling: Helps in zooming in or out on the image.
- Translation: Shifting the image along the x or y axis.
- Shearing: Slanting the shape of the image.

Color Space Augmentations

Color space transformations alter the chromatic characteristics of images through several methods:

- Brightness Adjustment: The image's overall brightness can be enhanced or reduced.
- Contrast Adjustment: This technique adjusts the tonal range between light and dark areas.
- Saturation Adjustment: This process changes the vividness and richness of colors within the image.
- Hue Adjustment: Colors are shifted by modifying their hue values across the spectrum.

Kernel Filters

Kernel filters apply convolutional operations to enhance or suppress specific features in the image. It includes:

- Blurring: Applying Gaussian blur to smooth the image.
- Sharpening: Enhancing the edges to make the image sharper.
- Edge Detection: Highlighting the edges in the image using filters like Sobel or Laplacian.

2.2. Lightweight CNN Architecture: MobileNetV2

The increasing need for deploying deep learning models on mobile and embedded devices has led to the development of efficient CNN architectures. MobileNetV2, introduced by Google, is a highly optimized architecture designed to perform well on platforms with limited computational power, such as smartphones, IoT hardware, or edge devices.

2.2.1. Core Architectural Features

MobileNetV2 improves on its predecessor by combining multiple design principles that reduce both model size and computational load:

- Depthwise Separable Convolutions: Decomposes standard convolutions into depthwise and pointwise operations. This significantly reduces the number of multiplications and additions by up to 8–9× compared to traditional convolutional layers [7].

- Inverted Residual Blocks: Rather than expanding and contracting feature maps arbitrarily, MobileNetV2 connects narrow bottleneck layers via expansion–depthwise–projection blocks. These blocks increase dimensionality only temporarily during computation and project back to lower-dimensional embeddings using 1×1 convolutions [8].

- Linear Bottlenecks: To prevent information loss, non-linear activations (e.g., ReLU) are removed from the final layer of bottleneck blocks, ensuring that low-dimensional features can be passed through intact.

- Expansion Layers: Each block begins with a dimensional expansion (using a hyperparameter-defined ratio), increasing the number of channels. This allows the network to learn richer features before compressing them again via projection.

These architectural choices enable MobileNetV2 to maintain competitive accuracy while remaining lightweight and fast key criteria for real-time vision systems.

2.2.2. Computational Complexity and Parameter Efficiency

One of the standout benefits of MobileNetV2 is its impressive computational efficiency:

- A fully trained MobileNetV2 model typically requires only 4.3 million parameters

- The number of multiply–accumulate (MAC) operations needed for inference is approximately 13 million

- The inference stage, which consists of a forward pass only, consumes significantly less energy and time than training, since it avoids backward propagation or weight updates

This makes MobileNetV2 highly suitable for low-power, latency-sensitive applications, including drowsiness detection on embedded platforms.

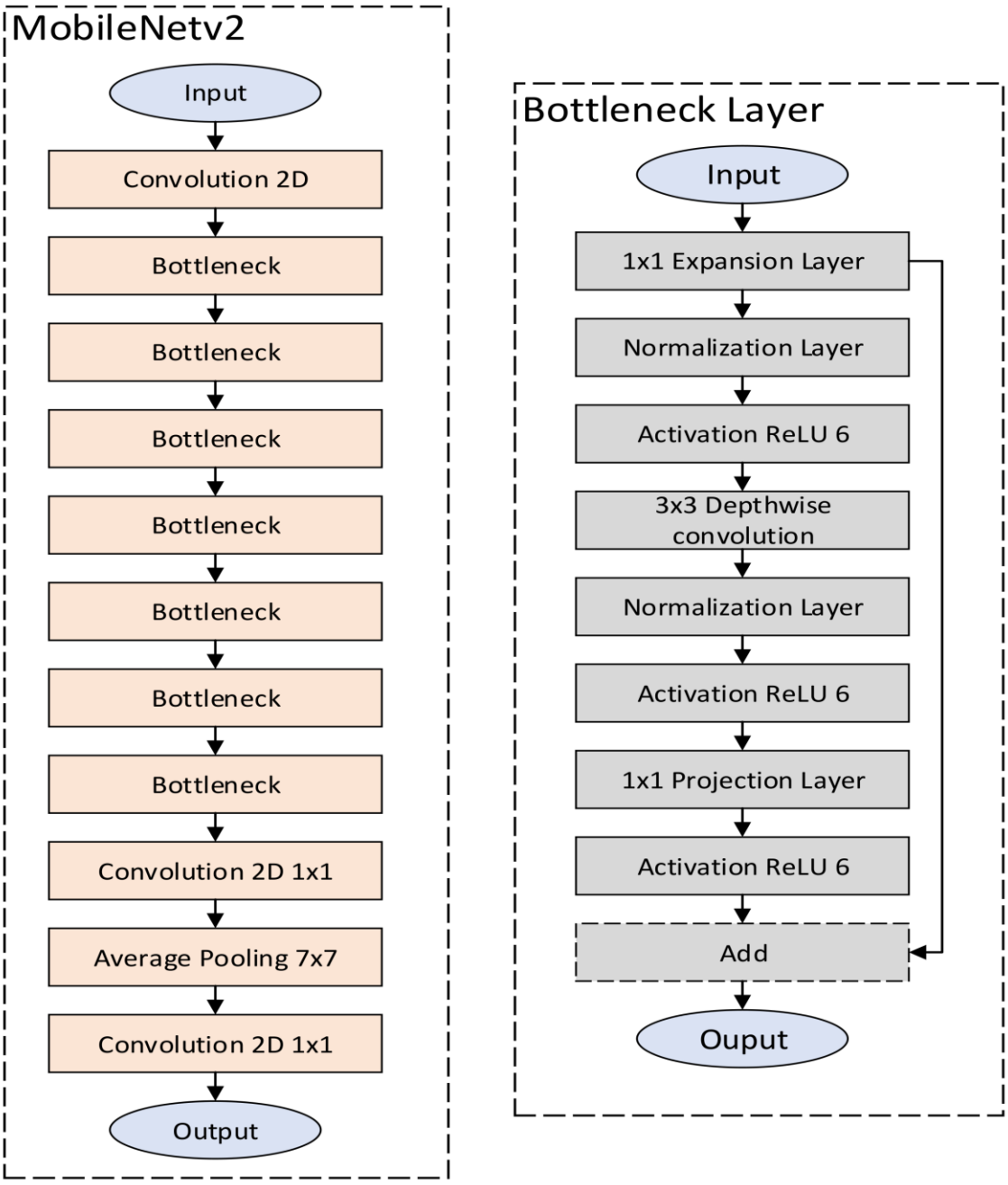

Figure 2.2. The architecture of MobileNetV2 DNN [7]

2.2.3. Flexibility and Hardware Scaling

MobileNetV2 is designed with two scaling hyperparameters:

- Width multiplier (α): Adjusts the number of channels in each layer. Lower α values lead to smaller, faster models at the cost of reduced capacity.
- Input resolution multiplier (ρ): Rescales the resolution of input images for a further trade-off between accuracy and speed.

Thanks to these factors, practitioners can easily tailor the model to the capabilities of their target hardware, making it ideal for FPGA, Raspberry Pi, or mobile NPU deployment.

2.2.4. Structured Pruning and Sparse Optimization

To further compress the model and accelerate execution, structured pruning techniques can be applied to MobileNetV2. Unlike unstructured pruning which removes arbitrary weights structured pruning eliminates entire filters or channels, producing more hardware-friendly sparsity:

- Two strategies can be employed:
  - Conservative pruning to retain most of the model's accuracy
  - Aggressive pruning for maximum compression, used for comparative analysis
- Compressed storage formats (e.g., compressed rows and columns) are used to reduce memory usage and increase data throughput
- Uniform compression schemas are applied across layers to reduce architectural irregularity and improve implementation simplicity

By applying sparse design principles, it becomes possible to maintain high landmark detection accuracy while reducing the MobileNetV2 model size and inference latency even further an especially important consideration for real-time drowsiness monitoring systems running on constrained devices.

2.3. Quantization

2.3.1. Overview Of Quantized Neural Network (QNNs)

Quantization has emerged as a leading strategy in addressing the memory and computational demands of deep learning (DL) models. Traditionally, DL networks represent parameters using 32-bit floating-point (FP32) precision. Quantization compresses these representations by substituting them with low-bitwidth values, thereby reducing both memory consumption and computational overhead. When integrated into convolutional neural networks (CNNs), the result is a Quantized Neural Network (QNN) a specialized model family with a growing record of practical successes [9].

A notable benefit of QNNs lies in their ability to enhance inference speed and energy efficiency, owing to the reduced complexity of quantized arithmetic operations. However, this compression introduces a trade-off: loss in numerical precision may lead to degraded accuracy. It is therefore essential to minimize such degradation when designing quantized models.

In the literature, two primary quantization strategies have been established: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). These methodologies are illustrated in *Figure 2.3*.

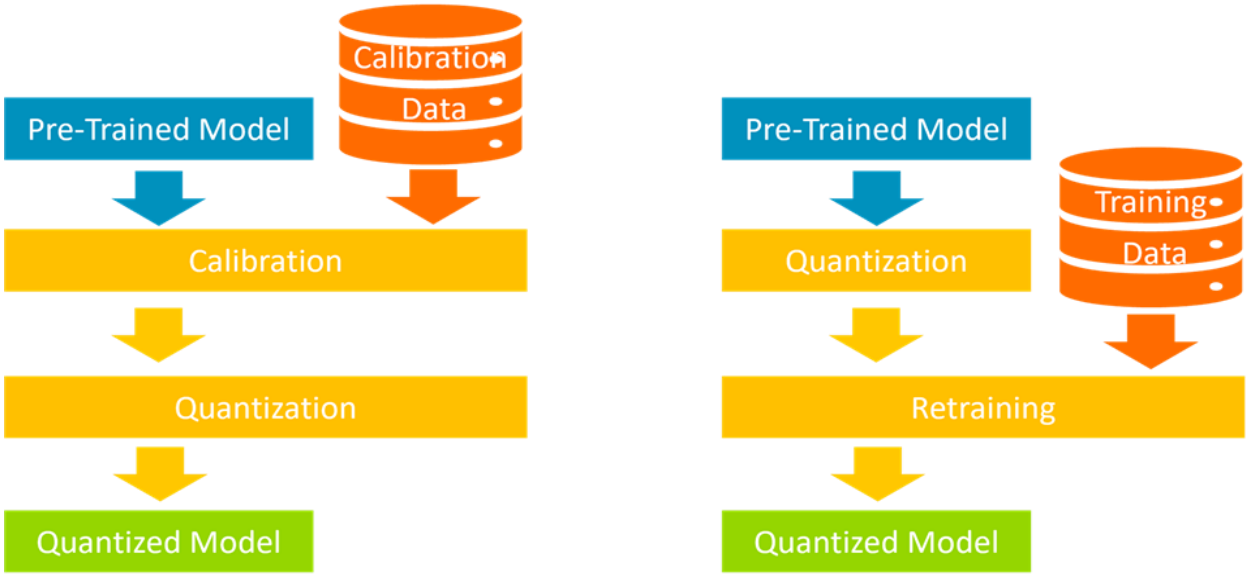

Figure 2.3. Quantization strategies with Post-Training Quantization (PTQ) in the left and Quantization-Aware Training (QAT) in the right.

2.3.2. Post-Training Quantization (PTQ)

PTQ is a method that applies quantization directly to pre-trained FP32 models without requiring retraining or access to labeled data, enabling rapid deployment and cost-effectiveness but being more prone to quantization-induced errors, particularly in aggressive low-bit scenarios [10]. PTQ operates on the principle that pre-trained models contain sufficient statistical information to determine appropriate quantization parameters through analysis of weight distributions and activation patterns.

The PTQ process centers around calibration using a representative subset of training data (typically 100-1000 samples) to characterize activation distributions, involving statistical analysis with range estimation, distribution analysis to optimize quantization schemes, and outlier handling to manage values that may skew quantization parameters [11]. Modern PTQ techniques employ sophisticated parameter optimization methods including percentile-based scaling to handle outliers, KL-divergence minimization to reduce information loss, and MSE-based optimization to minimize error between full-precision and quantized activations [12]. Advanced approaches perform layer-wise quantization with independent calibration for each layer, cross-layer optimization considering interactions between adjacent layers, and adaptive bit-width selection based on sensitivity analysis [13].

PTQ offers advantages of no retraining requirements, fast deployment suitable for production environments, and lower computational overhead, but generally achieves lower accuracy compared to QAT and depends heavily on representative calibration data. Recent developments have explored hybrid quantization strategies combining QAT and PTQ

elements, including progressive quantization, layer-specific strategies, and distillation-enhanced PTQ incorporating knowledge distillation principles [14].

Post-Training Quantization (PTQ) is applied to pre-trained models without retraining. It involves:

- ❖ Calibration: Analyzing activation distributions using a calibration dataset
- ❖ Scale Factor Determination: Computing optimal quantization parameters for weights and activations
- ❖ Model Conversion: Transforming the model to use quantized operations

PTQ is the most straightforward approach and requires minimal data (typically 100-1000 samples) for calibration.

2.3.3. Quantization-Aware Training (QAT)

QAT incorporates quantization effects during the training phase by simulating low-precision operations throughout training, enabling the model to adapt to quantized weights and activations and generally leading to improved performance relative to PTQ, though with increased training complexity, resource consumption, and the need for dataset availability during optimization [15].

QAT allows the model to learn representations that are robust to quantization noise by simulating quantization during forward passes while maintaining full precision for gradient computation, enabling superior accuracy preservation compared to post-training methods [16]. The theoretical foundation of QAT relies on the straight-through estimator (STE) to handle the non-differentiability of quantization functions, where "fake quantization" is applied during training by rounding values to discrete levels while maintaining full precision in the computational graph [17]. The typical QAT procedure follows a multi-stage approach including initial pre-training with full precision, quantization initialization with careful scale factor setup, and fine-tuning with quantization effects to optimize for quantized inference [16, 18].

Advanced QAT techniques incorporate learnable quantization parameters that adapt during training, mixed-precision strategies using different bit-widths across layers, and specialized regularization methods including quantization noise injection and knowledge distillation [16]. QAT offers superior accuracy preservation and supports aggressive quantization schemes including 4-bit, 2-bit, and binary quantization [19], but requires increased training time and computational resources, access to training data, careful hyperparameter tuning for quantization-specific parameters, and faces training stability challenges with very low bit-widths.

QAT incorporates quantization effects during the training process by:

1. Simulating quantization in the forward pass
2. Training with gradients calculated with respect to the quantized values
3. Storing weights in floating point during training, but constraining them to simulate quantization effects

QAT typically achieves better accuracy than PTQ at the cost of longer training time.

2.4. Overview Vitis AI

AMD's Vitis™ AI development platform serves as a comprehensive solution for enhancing artificial intelligence inference performance across various AMD hardware architectures, encompassing both edge computing devices and AMD Versal™ acceleration cards. This integrated ecosystem provides a complete suite of components, including refined intellectual property cores, flexible development tools, robust software libraries, varied AI models, and demonstrative design examples. By prioritizing operational efficiency and user accessibility, the Vitis AI platform effectively harnesses the complete capabilities of AI acceleration within AMD System-on-Chips and Adaptive System-on-Chips. The development environment streamlines the creation of deep learning inference applications for users who may lack extensive field-programmable gate array expertise, achieving this by simplifying the complexities associated with underlying programmable logic architectures [20].

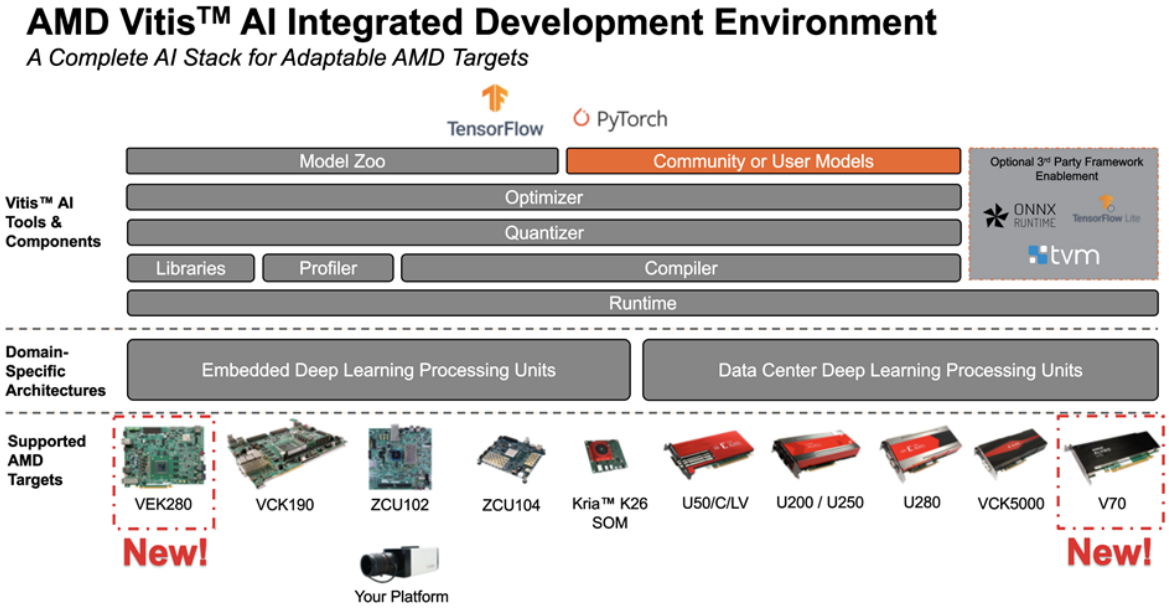

Figure 2.4. Vitis AI Integrated Development Environment [20]

2.4.1. Vitis AI includes the following features:

- Compatible with popular deep learning frameworks and cutting-edge models designed for various artificial intelligence applications.

- Delivers an extensive collection of pre-optimized neural network models that are deployment-ready for AMD hardware platforms.
- Features a robust quantization tool that enables model compression, calibration processes, and fine-tuning capabilities. For experienced developers, AMD provides an additional AI optimization utility that can reduce model size by as much as 90% while maintaining acceptable performance levels.
- Enables detailed layer-wise performance analysis to identify and resolve computational bottlenecks.
- Delivers standardized high-level programming interfaces in both C++ and Python languages, ensuring seamless portability across Edge computing and data center environments.
- Develops tailored, high-performance intellectual property cores that address specific application demands while optimizing critical performance parameters including processing speed, response time, and energy efficiency.
- Engineers customized and scalable IP core solutions to accommodate varied application specifications and enhances system performance across essential benchmarks such as data throughput, response latency, and power utilization [20].

2.4.2. Quantization In Vitis AI

The Vitis AI framework utilizes precision reduction methodologies to transform deep learning models from high-precision floating-point data types (typically FP32) to lower-precision fixed-point numerical formats. This transformation process is designed to enable enhanced inference performance on Deep-learning Processing Units (DPUs), which are engineered specifically for fixed-point arithmetic operations.

The Vitis AI development environment supports precision conversion procedures for deep neural network models derived from various machine learning frameworks:

Table 2: Vitis AI supports quantization

Framework	Post-Training Quantization	Quantization-Aware Training	Supported Data Types
PyTorch	✓	✓	INT8
TensorFlow 1.x	✓	✓	INT8
TensorFlow 2.x	✓	✓	INT8

These quantized models can be deployed on various AMD/Xilinx hardware platforms including Versal VEK280, Alveo V70 data center cards, and embedded platforms [20].

To achieve the optimization the FPGA will have multiple processes used [21].The multiple steps that we followed are:

Pruning:

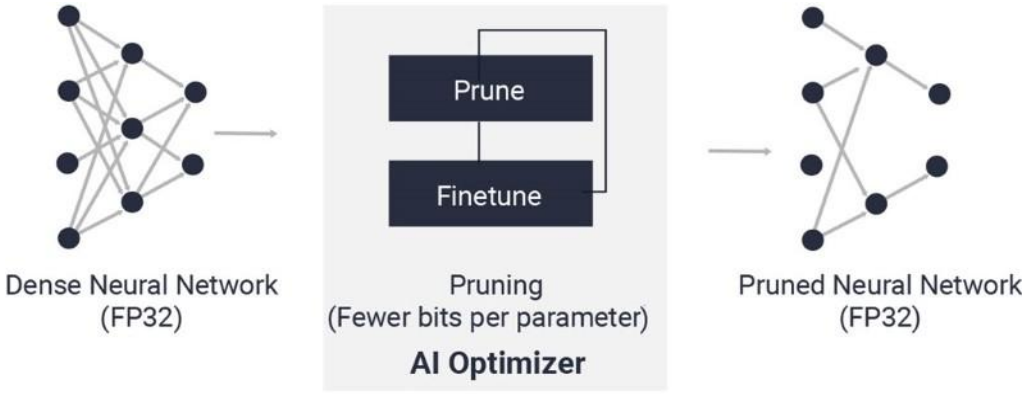

Figure 2.5: Pruning

AMD's AI-Optimizer receives a neural network model and implements a network reduction methodology [22], as illustrated in *Figure 2.5*. The optimization process begins with a sensitivity evaluation designed to assess how individual convolution filters within each layer contribute to the final prediction [23]. Subsequently, the system identifies filter weights designated for elimination, meaning they will be set to zero values. This identification is accomplished through performance assessment of the original trained model prior to any component removal.

Following this, weight parameters undergo adjustment across multiple training cycles to restore model accuracy. The pruning procedure typically operates through successive iterations, where each cycle allows examination of the network's current state, accuracy degradation, and the quantity of neurons selected for removal. Based on these metrics, decisions can be made regarding when to halt the pruning process or revert to previous states as necessary [24].

Upon completion of the final stage, which eliminates the neurons designated for removal, the resulting model contains fewer neurons and consequently requires reduced computational resources. For example, a layer that originally required processing 128 feature channels might only need to handle 87 channels following the pruning operation [23]. *Figure 2.6* presents a visual representation of the pruning workflow.

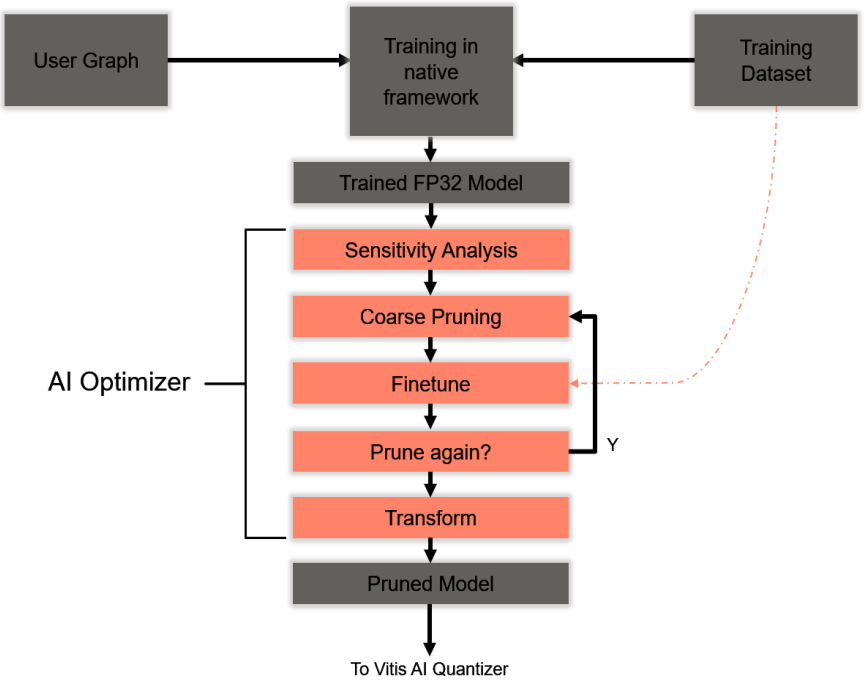

Figure 2.6. Prunning process flow

Figure 2.7 shows the accuracy and parameter reduction modification during the loop of pruning.

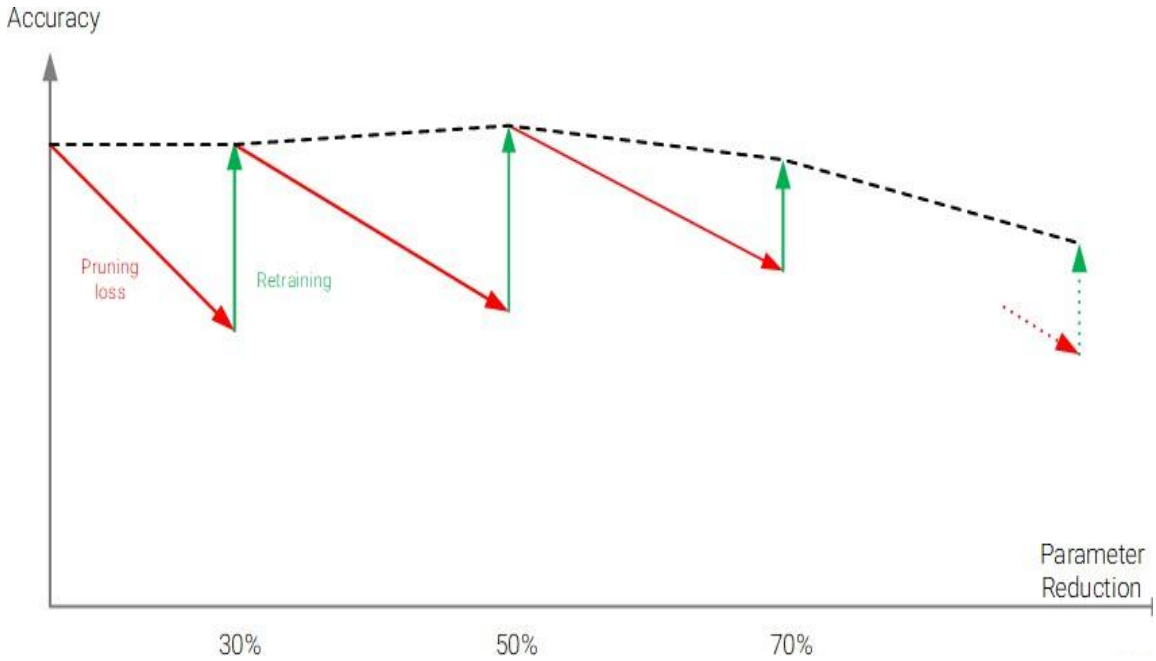

Figure 2.7. Evolution of accuracy and parameter reduction over iterations in the pruning process

Quantization

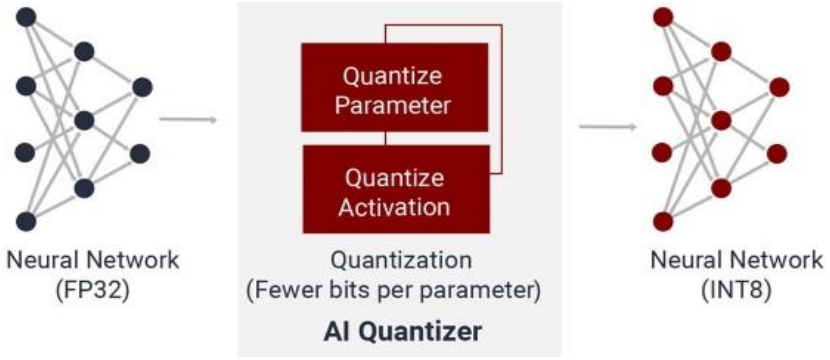

Figure 2.8. Quantization process

This approach improves deployment performance through the application of integer-based precision reduction to decrease data transmission requirements, power consumption, and memory utilization [25]. In this implementation, we apply INT8 precision conversion to the previously compressed network. The Vitis AI Quantization tool initially executes a calibration phase, during which it selects a subset of training data samples and processes them through the network architecture to monitor the activation patterns of individual channels. The network parameters and activation values are subsequently converted to 8-bit integer representations. This procedure is commonly referred to as Post-Training Quantization [26].

2.5. Edge Computing

Edge computing represents a decentralized computational architecture that positions business applications in closer proximity to data generation points, including Internet of Things sensors or localized edge processing nodes. This spatial closeness to data origins can provide significant organizational advantages, such as accelerated analytical insights, enhanced system responsiveness, and improved network capacity utilization [27]. Edge computing encompasses both data manipulation processes and real-time local information processing. The key edge infrastructure elements that can be relied upon include Data manipulation capabilities, Decision-making engines, and Localized storage systems [28].

Figure 2.9. Edge Computing

Why Edge Computing?

- This approach optimizes bandwidth utilization by performing data analysis at peripheral locations, contrasting with cloud computing that necessitates data transmission from IoT devices requiring substantial network capacity, making it advantageous for deployment in remote areas with cost-effective solutions [29].
- It enables intelligent applications and devices to process information with near-instantaneous response times, which is crucial for business operations and autonomous vehicle systems [30].
- The technology possesses the capability to handle data processing without relying on public cloud infrastructure, thereby ensuring comprehensive data security and privacy [31].
- Information may become compromised during transmission across extended network infrastructures, potentially impacting data integrity for industrial applications [32].
- Edge-based data computation provides an alternative that reduces dependency on cloud computing resources [33].

2.6. FPGA

Field Programmable Gate Arrays (FPGAs) are reconfigurable digital integrated circuits (ICs) that allow designers to program custom hardware logic after manufacturing. Unlike

ASICs (Application-Specific Integrated Circuits), the logic within an FPGA is not fixed at fabrication but can be configured by the end-user to meet application-specific requirements. This field programmability makes FPGAs particularly valuable in domains requiring flexibility and rapid prototyping [34].

In the context of edge computing and AI, FPGAs are emerging as a promising solution for deploying resource-intensive deep learning models on power- and latency-constrained platforms. Their ability to perform highly parallel computations and optimize hardware-level dataflows makes them well-suited for accelerating Convolutional Neural Networks (CNNs) [35].

Internal structure of an FPGA

The accompanying diagram illustrates a representative internal architecture of an FPGA from a high-level perspective. The structural design of an FPGA consists of three fundamental elements [36]:

- **Configurable Logic Blocks (CLBs):** These units incorporate Look-Up Tables (LUTs), storage elements, and switching circuits, facilitating the realization of both combinational and sequential digital logic. In artificial intelligence acceleration applications, CLBs serve to construct specialized processing units (PUs) for computational tasks such as matrix operations, convolution processes, and activation functions [37].
- **Programmable Interconnect Network:** An adaptable routing infrastructure that connects the CLBs, enabling dynamic reconfiguration of signal pathways. This capability permits designers to optimize data movement between AI processing modules, reducing processing delays and enhancing computational efficiency [38].
- **Input/Output (I/O) Interface Blocks:** These components establish communication with peripheral devices (such as imaging sensors, storage systems, display interfaces) and serve a vital function in time-critical applications like driver monitoring systems, where sensor data requires processing and visualization with minimum latency [39].

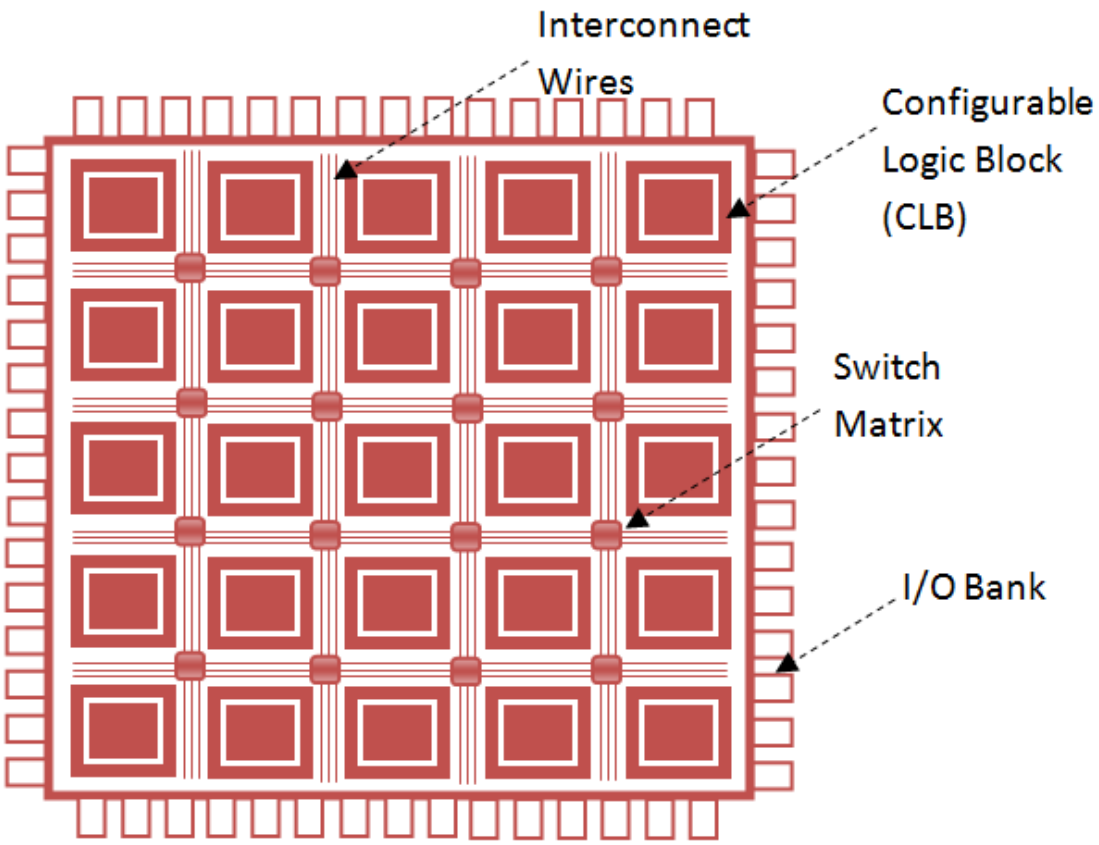

Figure 2.10. A typical internal structure of an FPGA [40]

The fundamental architecture of an FPGA is constructed from adaptable logic units and reconfigurable interconnect networks. These central components are encompassed by multiple programmable input/output modules, which facilitate communication with peripheral systems.

FPGA encompasses three essential elements:

- Configurable Logic Units (or Logic Modules) – tasked with executing primary computational functions
- Reconfigurable Interconnect Network – responsible for establishing connections between Logic Modules
- Input/Output Interface Modules – which link to the Logic Modules via the interconnect system and enable external system integration [40].

FPGA for Edge AI and This Research

In this thesis, the FPGA's flexibility and parallel processing capabilities are leveraged to accelerate a quantized MobileNetV2 for facial landmark detection. Compared to CPU and GPU platforms:

- **Energy Efficiency:** The Kria KV260 FPGA achieves real-time inference with low power, significantly outperforming CPU and GPU implementations.
- **Low Latency:** Hardware-level pipelines on the FPGA eliminate software overheads, ensuring prompt detection of driver drowsiness cues like eyelid closure and head pose.
- **Custom Dataflow:** The design uses PTQ + QAT INT8 quantized models mapped to the FPGA's reconfigurable fabric, enabling memory-efficient and high-throughput operations tailored for edge devices.

Contribution of FPGA in the Proposed System

The proposed system utilizes the Xilinx Kria KV260 platform to demonstrate how FPGA-based acceleration addresses the challenges of deploying CNNs on edge devices:

- **Compactness:** Supports ultra-lightweight models after quantization.
- **Power Efficiency:** Enables continuous, always-on monitoring without thermal constraints.
- **Real-Time Performance:** Meets stringent latency requirements critical for driver safety applications.

This approach highlights FPGA as a practical alternative to GPUs and CPUs for edge AI deployment, particularly in automotive environments where power and space are limited.

2.7. Pytorch Framework

PyTorch represents a machine learning framework built upon the Torch library, designed for applications including computer vision and natural language processing tasks. Initially created by Meta AI and currently maintained under the Linux Foundation's governance, it stands as one of the leading deep learning platforms alongside frameworks like TensorFlow, providing open-source software distributed under the modified BSD license. While the Python interface receives primary development attention and offers enhanced functionality, PyTorch also supports C++ programming interfaces [41].

Figure 2.11. Pytorch Framwork [41]

PyTorch provides two high-level features:

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPUs)
- Deep neural networks built on a tape-based automatic differentiation system [41]

In this thesis, PyTorch plays a central role in the design, training, and optimization of the facial landmark detection model. PyTorch was selected as the primary deep learning framework due to its flexibility, dynamic computational graph support, and extensive community ecosystem, which align perfectly with the iterative development requirements of edge AI systems.

The research workflow utilizing PyTorch consists of the following key stages:

Model Development and Training

- The backbone network (MobileNetV2) and auxiliary network were implemented in PyTorch.
- Training was conducted on the 300W, AFLW, and WFLW datasets, with preprocessing, data augmentation, and performance tracking all handled using PyTorch utilities.
- PyTorch's seamless GPUs acceleration facilitated efficient training on large datasets and enabled rapid experimentation with different architectures and hyperparameters.

Quantization Pipeline

- PyTorch served as the foundation for applying Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) using the Brevitas library.

- This integration made it possible to produce highly compressed models suitable for deployment on fixed-point hardware such as FPGAs.

Export for Hardware Deployment

- After training and quantization in PyTorch, the models were converted into Xilinx's .xmodel format using the Vitis AI compiler.
- The compatibility of PyTorch with Vitis AI's toolchain simplified the transition from software training environments to hardware-accelerated inference on the Kria KV260 FPGA.

2.8. Overview of Hardware Platforms for Comparison

To comprehensively evaluate the proposed CNN accelerator, this thesis examines five representative hardware platforms, each selected for its distinct architectural characteristics, deployment scenarios, and performance profiles. This diverse selection provides a holistic view of the trade-offs in AI inference across varying computational environments. A detailed performance comparison is presented in *Chapter 4*, while this section introduces each platform and its relevance.

2.8.1. NVIDIA RTX 4060 GPU

The NVIDIA RTX 4060 is a high-performance consumer GPU built on a 5 nm process node. Designed for massively parallel workloads, it delivers exceptional throughput for AI inference tasks. In this study, it serves as a baseline for ultra-low-latency and high-throughput scenarios.

- **Use Case:** Data center inference, cloud-edge hybrid AI systems, video analytics.
- **Key Feature:** Massive parallelism with CUDA cores optimized for deep learning.

Figure 2.12. RTX 4060 GPU

2.8.2. Intel Core i9-13900K CPU

The Intel Core i9-13900K, part of the Raptor Lake generation, features a hybrid architecture combining performance and efficiency cores. It represents a general-purpose inference engine capable of CPU-only deployments, suitable for scenarios where GPU acceleration is unavailable.

- **Use Case:** Desktop/workstation inference, CPU-based optimization testing.
- **Key Feature:** Balanced performance for AI workloads without dedicated accelerators.

Figure 2.13. Intel Core i9-13900K CPU

2.8.3. NVIDIA Jetson Nano 2GB

The Jetson Nano 2GB is an entry-level edge AI platform featuring a quad-core ARM Cortex-A57 CPU and a 128-core Maxwell GPU. Despite its limited power envelope, it supports lightweight frameworks such as TensorRT and ONNX Runtime, making it suitable for low-demand AI applications.

- **Use Case:** Prototyping, hobbyist projects, simple computer vision tasks.
- **Key Feature:** Compact and power-efficient design for edge AI prototyping.

Figure 2.14. NVIDIA Jetson Nano 2GB

2.8.4. Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B is a widely adopted single-board computer (SBC) with broad support for embedded applications. Included in this study to represent resource-constrained systems, it helps assess the feasibility of CNN inference without hardware acceleration.

- Use Case: Educational purposes, proof-of-concept deployments, simple embedded vision.
- Key Feature**:** Versatile SBC platform with extensive community support.

Figure 2.15. Raspberry Pi 4 Model B

This cross-platform selection enables a balanced assessment of performance, energy efficiency, and deployment feasibility. By comparing architectures ranging from high-end GPUs to resource-limited SBCs, this study illustrates the trade-offs and highlights the advantages of the proposed FPGA-based solution in real-world edge AI scenarios.

2.9. Related works

Facial landmark detection has been widely studied due to its applications in driver monitoring, human-computer interaction, and biometric authentication. However, deploying these systems on resource-constrained edge platforms introduces challenges in computational efficiency, energy consumption, and real-time performance. Table 2.1 summarizes recent works, highlighting their model architectures, datasets, target hardware, quantization strategies, and key performance metrics.

This comparison reveals that while some studies achieve high accuracy, few address the stringent power and latency requirements for edge AI deployment. In particular, FPGA-based solutions remain underexplored despite their potential for energy-efficient inference.

Guo et al. (2019) demonstrated PFLD, a lightweight facial landmark detection model achieving high accuracy (92.5%) and very high throughput (120 FPS). However, this system relies on a high-performance GPU (RTX 2080 Ti), consuming approximately 250 W of power, which is unsuitable for battery-powered edge devices. Moreover, although the model size is compact (2.1 MB), no quantization techniques were applied to further optimize deployment for embedded systems.

Wu et al. (2020) explored the deployment of MobileNetV2 on the NVIDIA Jetson TX2 using post-training quantization (PTQ). This approach reduced the model's power consumption to 7 W, making it more energy-efficient than GPU-based implementations. However, the model size (14 MB) remains relatively large, and the system achieved only

15 FPS, which is insufficient for real-time applications in dynamic environments like driver monitoring.

Yang et al. (2021) implemented MTCNN on a general-purpose CPU (Intel i7-8700K). While achieving moderate accuracy (89.7%), this system suffered from significant limitations in both performance and energy efficiency. The model size (68 MB) and lack of quantization resulted in a low frame rate (8 FPS) and high power usage (80 W), making it impractical for edge deployment scenarios.

Table 3: Related Works Comparision

Author	**Model (Size)**	**Hardware**	**Quantization**	**Accuracy (%)**	**FPS**	**Power (W)**	**Application**
Guo et al. (2019)	PFLD (2.1 MB)	GPU (RTX 2080 Ti)	None	92.5	120	~250	Face alignment in AR systems
Wu et al. (2020)	MobileNet V2 (14 MB)	Jetson TX2	PTQ (INT8)	91.2	15	7	Driver monitoring
Yang et al. (2021)	MTCNN (68 MB)	CPU (i7-8700K)	None	89.7	8	80	Real-time video conferencing
Proposed (This Work)	MobileNet V2 – Quantized (1.65 MB)	FPGA (Kria KV260)	QAT (INT8)	91.74	30	3	Driver drowsiness detection

CHAPTER 3: IMPLEMENTATION

3.1. System Overview

An incoming RGB frame is first processed by a face detector (e.g., Haar Cascade or a lightweight CNN), which yields a tight bounding box around the face. The detected patch is converted to grayscale to reduce input dimensionality and normalized to the [0, 1] range. Resizing to 112 × 112 pixels standardizes spatial resolution across all samples.

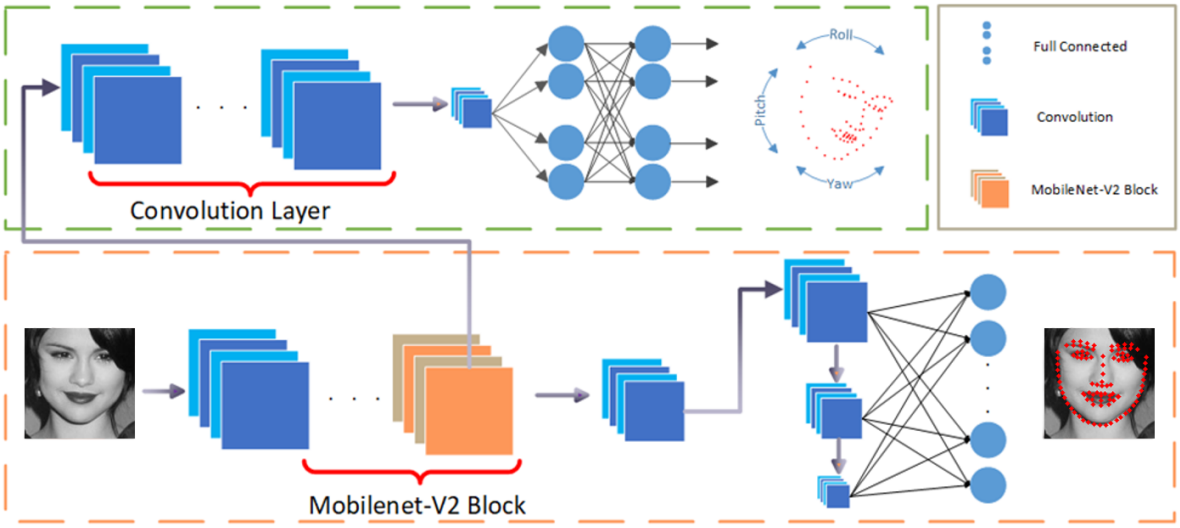

Figure 3.1. System Overview [42]

Figure 3.1. System overview pipeline. First, the input RGB frame undergoes face detection, grayscale conversion, normalization, and resizing to 112×112. During training, online augmentation (geometric transforms, photometric adjustments, and filtering) is applied. The augmented image is passed through the Backbone module (MobileNetV2), then the Neck for multi-scale feature fusion, and the Head to regress 98 landmark coordinates. A training-only Auxiliary Pose branch predicts Euler angles for geometric regularization; this branch is removed during inference to ensure low latency.

During training, each crop undergoes a sequence of geometric augmentations (rotation, translation, shear, flip), photometric (brightness, contrast, saturation, hue), and kernel-based filtering (Gaussian blur, sharpening, edge enhancement) to simulate real-world variations such as head tilt, illumination shifts, and motion blur. Augmented images are fed into the core model, which consists of:

- A Backbone (MobileNetV2-inspired) extracting hierarchical features.
- A Neck module fusing multi-scale descriptors into a unified representation.
- A Head regressing 98 landmark coordinate pairs via a lightweight fully connected layer.
- An Auxiliary Pose Branch (training-only) predicting yaw, pitch, and roll for geometric regularization.

At inference, the auxiliary branch is pruned to preserve low latency, allowing real-time landmark detection on embedded hardware.

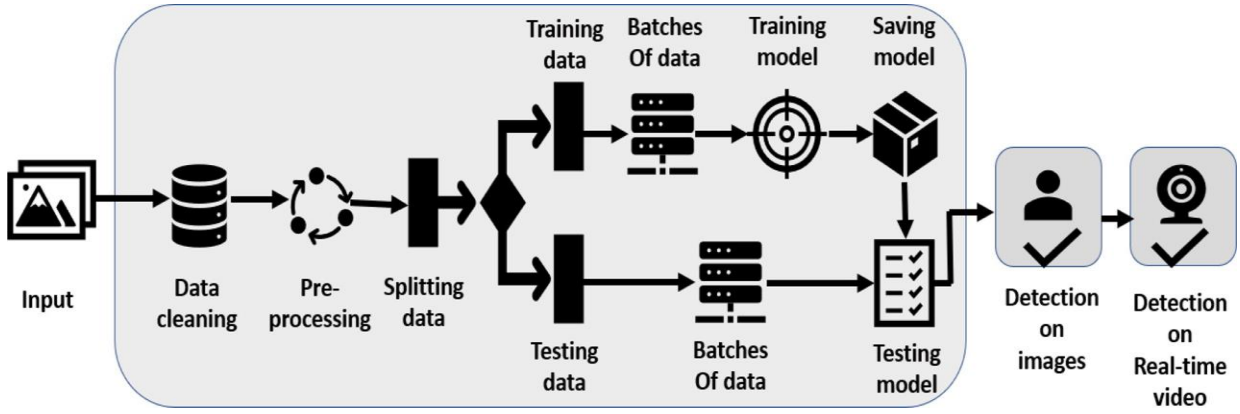

Figure 3.2. Flow diagram

3.1.1 Data Preparation and Augmentation

❖ **Pre-processing**

Face detection employs a cascade classifier or a small CNN to produce a bounding box. The face region is cropped with a small margin to include peripheral features (outline of jaw, hairline). Grayscale conversion, followed by histogram equalization or contrast normalization, standardizes appearance across lighting conditions. Finally, resizing to 112 × 112 uses bilinear interpolation, and pixel values are scaled to [0, 1]. Landmark annotations, originally in text format, are loaded as NumPy arrays of shape (98, 2).

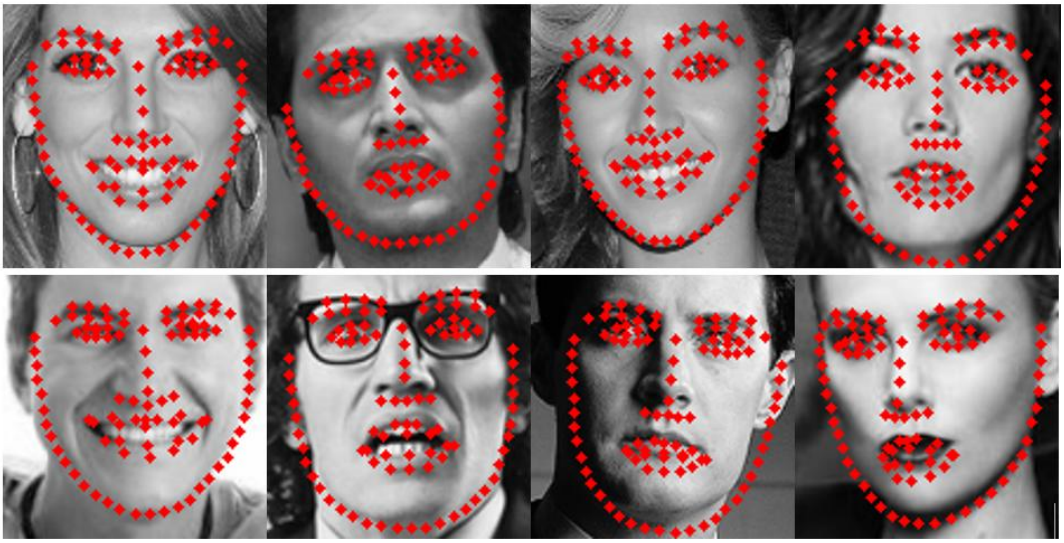

Figure 3.3. 98 Facial Landmarks Images

Facial feature points (alternatively referred to as facial landmarks) are represented by the small magenta-colored dots visible on each face within the above image. Every image in both the training and testing sets contains a single face accompanied by 98 landmark

points, each having coordinate values (x, y) for that particular face. These landmark points identify crucial facial regions including the eye areas, mouth corners, nasal region, and other significant facial components. These feature points play an essential role in numerous applications, including facial filters, emotion detection, posture identification, and various other domains. The points are sequentially numbered, and it can be observed that specific point ranges correspond to different facial segments.

The facial landmark detection and application dataset exhibited significant amounts of noise and contained numerous duplicate images throughout the collection. Given that high-quality datasets are fundamental to achieving optimal model performance during training, addressing these issues becomes crucial. Unprocessed data typically suffers from various challenges including absent values, corrupted information, partial datasets, conflicting entries, and anomalous data points. Therefore, it becomes imperative to process this information thoroughly prior to implementing data mining techniques. Data preprocessing represents a critical phase for improving overall data effectiveness. The preprocessing pipeline encompasses multiple methodologies such as data cleansing, integration procedures, transformation techniques, and dimensionality reduction approaches.

Prior to implementing CNN training for facial landmark detection tasks, we execute a comprehensive sequence of preprocessing procedures to guarantee data integrity, uniformity, and adequate diversity across the dataset.

Following these initial steps, data augmentation techniques are implemented to enhance model accuracy during the training phase.

❖ **Data Augmentation**

A substantial volume of data is required for effective training processes, particularly when facing limitations in available training data for the proposed model architecture. Data augmentation methodology serves as a solution to address this challenge. This approach employs various techniques, including image rotation, scaling operations, positional shifts, shearing transformations, and horizontal/vertical flipping to create multiple variations of identical images. Within our proposed framework, image augmentation serves as the primary mechanism for expanding the dataset. An image data generation function is developed to facilitate the augmentation process, producing both training and validation data batches.

To enhance model robustness, the following augmentations are applied on the GPU during training:

- **Geometric transforms**
  - Rotation: random ±15° around image center
  - Translation: random shift ±10 pixels horizontally/vertically

• Shear: random ±10° • Horizontal flip: with 50% probability

- **Photometric jitter**
  - Brightness: ±30%
  - Contrast: ±30%
  - Saturation & hue: jitter within ±10%
- **Kernel-based filtering**
  - Gaussian blur: σ ~ Uniform(0.5, 1.5)
  - Sharpening: unsharp mask parameters randomized
  - Sobel edge enhancement: combine the edge map with the original

Each augmentation preserves landmark correspondence by applying identical transformations to coordinate pairs.

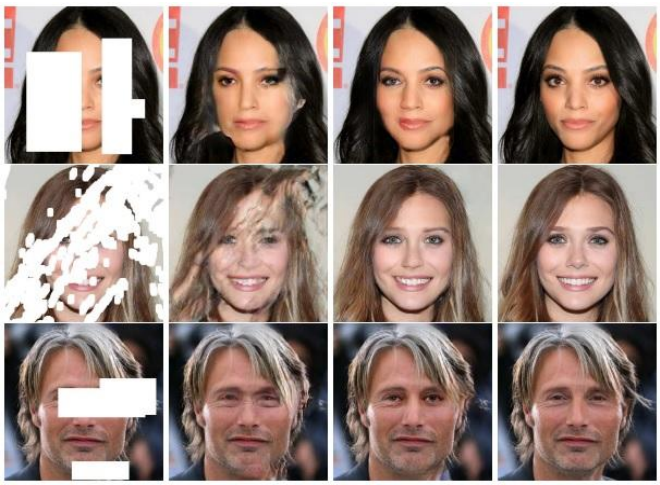

Figure 3.4. Photometric transformation examples

3.1.2. Model Architecture

The landmark detection network employs a **Backbone–Neck–Head** design, augmented by a training-only Auxiliary Pose branch. This modular structure balances accuracy, efficiency, and ease of hardware mapping.

❖ **Backbone Network (MobileNetV2 Blocks)**

Recognizing that the backbone subnetwork is the only module active during inference, it becomes the primary bottleneck in both runtime and model footprint. To optimize for speed and memory efficiency, we adopt MobileNet [8] as our base architecture. MobileNet's architectural innovations including depthwise separable convolutions, linear bottlenecks, and inverted residual connections, have proven effective in reducing complexity without compromising performance [44, 45]. Accordingly, we replace standard convolution operations with MobileNet blocks to significantly lower computational demands and accelerate the network. By doing so, the computational load of our backbone network is significantly reduced, and the speed is thus accelerated.

The backbone extracts hierarchical features via depthwise-separable convolutions and inverted residual bottlenecks, following MobileNetV2 principles. Early downsampling broadens the receptive field; subsequent blocks learn compact embeddings.

Table 4: The backbone network configuration

Input	Operator	t	c	n	s
$112^2 \times 1$	Conv3 × 3	-	64	1	2
$56^2 \times 64$	Conv 3 × 3	-	64	1	1
$56^2 \times 64$	Bottleneck	2	64	1	2
$28^2 \times 64$	Bottleneck	2	64	4	1
$28^2 \times 64$	Bottleneck	2	128	1	2
$14^2 \times 128$	Bottleneck	4	128	5	1
$14^2 \times 128$	Bottleneck	2	128	1	1
$14^2 \times 128$	Bottleneck	2	16	1	1
(S1) $14^2 \times 16$	Conv3 × 3	-	32	1	2
(S2) $7^2 \times 32$	Conv1 × 1	-	64	1	1
(S3) $1^2 \times 128$	Global AvgPool	-	-	-	-
S1,S2,S3	Fully Connected	-	196	1	-

- t: expansion factor
- c: output channels
- n: number of repeated blocks
- s: stride of first block in the stage

This stagewise layout reduces computations by > 4× compared to standard CNNs, while preserving feature quality for landmark localization.

❖ **InvertedResidual Stages:**

- **Stage 1 (conv3):** The first down-sampling InvertedResidual block (stride = 2, expansion factor = 2) reduces the spatial resolution from 56×56 to 28×28. It is followed by four residual blocks (stride = 1, expansion = 2) that preserve the spatial dimensions and maintain 64 output channels, enabling the extraction of fine-grained local features.
- **Stage 2 (conv4–5):** A second down-sampling block (stride = 2, expansion = 2) increases the channel dimension to 128 and reduces the resolution to 14×14. Subsequently,

five residual blocks (stride = 1, expansion = 4) further process mid-level features while maintaining spatial consistency.

- **Stage 3 (conv6):** A final InvertedResidual block (stride = 1, expansion = 2) projects the features down to 16 channels at 14×14 resolution. This output serves as the input for the multi-scale pooling module.

❖ **Multi-Scale Pooling & Neck Fusion**

To capture both fine details and global context, three parallel branches are constructed from the 14×14×16 feature map:

- **Branch 1:** Direct global average pooling over the 14×14×16 feature map produces a 16-dimensional vector.

- **Branch 2:** A 3×3 convolution (conv7) followed by ReLU activation transforms the input into 32 channels at 7×7 resolution. Global average pooling is then applied to yield a 32-dimensional vector.

- **Branch 3:** A 1×1 convolution (final_conv) followed by Batch Normalization and ReLU activation produces 64 channels at 7×7 resolution. Global average pooling is applied to obtain a 64-dimensional vector.

- **Fusion:** The three vectors (16 + 32 + 64) are concatenated to form a unified 112-dimensional feature representation. This fused feature is passed through a fully connected layer to regress 196 values corresponding to 98 facial landmarks (each with x and y coordinates).

❖ **Head Module**

The head regresses 2D landmark coordinates from the fused feature:

- **Fully Connected Layer:** The 112-dimensional fused feature is mapped to 196 outputs via a single fully connected layer.
- **Reshape:** The output vector is reshaped into a (98 × 2) matrix representing the 2D coordinates of facial landmarks.

❖ **Auxiliary Pose Estimation Branch**

Unlike prior methods that directly regress a 3D–2D projection matrix [28], infer part-wise dendritic structures [46], or exploit boundary lines [45], PFLD's auxiliary subnet independently estimates the three Euler angles (yaw, pitch, roll) to regularize landmark learning under large pose variations. To prevent unstable angle estimation from noisy landmark outputs, especially early in training, we define a fixed, average frontal-face template with 11 reference landmarks and compute each sample's rotation matrix against this template; the corresponding Euler angles then serve as auxiliary supervision without requiring any extra pose annotations. Although these angles are only approximate since they use an average face, they prove sufficient to guide the model toward stable and robust landmark localization, and the auxiliary branch is removed at inference to incur zero runtime overhead.

During training only, an auxiliary subnetwork regularizes landmark learning under large pose variations:

Table 5: The Auxiliary Branch Configuration

Input	Operator	c	s
$28^2 \times 64$	Conv3 × 3	128	2
$14^2 \times 128$	Conv3 × 3	128	1
$14^2 \times 128$	Conv3 × 3	32	2
$7^2 \times 32$	Global AvgPool	-	-
$1^2 \times 32$	Fully Connected	32	-
$1^2 \times 32$	Fully Connected	3	-

Table 5 provides the configuration details of the proposed auxiliary network. It is important to note that the input to the auxiliary branch is extracted from the 4th block of the backbone network (as described in *Table 4*).

- **Feature Extraction**: captures intermediate representations.
- **Angle Regression**: FC outputs three Euler angles.
- **Loss Coupling**: predicted angles weight the landmark loss
- **Inference Pruning**: this branch is removed at test time, ensuring zero runtime overhead.

Since the auxiliary branch is no longer needed during testing, we do not apply the MobileNet techniques in our implementation.

By combining efficient depthwise separable convolutions, multi-scale fusion, and geometric regularization, this architecture achieves high-precision landmark localization with minimal computational footprint, ideal for real-time, edge-deployed face-aware applications.

3.1.3. Loss Function

Loss Function: The effectiveness of the training process is heavily influenced by how the loss function is formulated. This is particularly important when the training dataset is not sufficiently large. A straightforward way to penalize discrepancies between ground-truth landmarks $X=[x_1,...,x_N] \in R^{2\times N}$ and predicted landmarks $Y=[y_1,...,y_N] \in R^{2\times N}$ is by employing traditional $\ell 1$ or $\ell 2$ losses. However, assigning equal weight to every pair of landmark deviations ignores the geometric context of the face, which is not ideal.

To illustrate, consider a displacement vector $d_i = x_i - y_i$ in 2D image space. When projecting from a real 3D face to a 2D image using different viewpoints (camera poses),

the actual distances on the face surface can vary greatly even if the 2D deviations appear similar. Therefore, integrating geometric information into the loss function helps account for this variation and leads to more meaningful penalization

In the context of face images, the 3D pose of the head provides sufficient information to govern the projection from 3D to 2D. Let $U \in R^{4\times N}$ denote the homogeneous coordinates of 3D facial landmarks, where each column represents a point in 3D space as $[u_i, v_i, z_i, 1]^T$. Assuming a weak perspective model [28], a projection matrix $P \in R^{2\times 4}$ maps these 3D landmarks to 2D image coordinates via the relationship:

$$X=PU$$

This projection matrix P has six degrees of freedom: three for rotation (yaw, pitch, roll), one for scaling, and two for translation in the 2D plane. However, since faces in this framework are assumed to be well-detected, centered, and normalized, the effects of scale and translation can be largely ignored. As a result, only the three Euler angles, yaw, roll, and pitch, need to be estimated to describe the 3D-to-2D projection accurately.

In deep learning-based landmark detection, data imbalance is a recurring challenge that can severely impact performance. For instance, a training dataset may contain an abundance of frontal face images but lack sufficient samples of faces in extreme poses. Without specific handling strategies, models trained on such skewed data distributions typically fail to generalize well to underrepresented pose variations. Under these circumstances, treating all training samples equally within the loss function introduces unintended bias. To mitigate this, we propose assigning higher weights to rare samples, penalizing the model more when it errs on them.

The proposed framework consists of two interconnected subnetworks:

- A backbone subnetwork (lower branch) responsible for predicting the coordinates of facial landmarks.
- An auxiliary subnetwork (upper branch) dedicated to estimating geometric pose information.

The quality of training in landmark detection is critically dependent on how the loss function is formulated, especially in scenarios where the dataset is limited in size or skewed in distribution. To address multiple challenges, including geometric variation and data imbalance, we propose a novel loss function defined as:

$$\mathcal{L} := \frac{1}{M} \sum_{m=1}^{M} \sum_{n=1}^{N} \gamma_n \|\mathbf{d}_n^m\|, \qquad (1)$$

Where:

- $\|\cdot\|$ is a distance metric (e.g., ℓ_2 norm) used to measure the error between the predicted and ground-truth location of the nth landmark in the mth sample.
- MM is the number of training images.
- NN is the number of landmarks per image.
- γn is a landmark-specific weight that controls the penalty applied to each landmark's deviation, modulated by two factors:
    - Geometric reliability: Based on pose deviation
    - Data imbalance: Based on attribute rarity

$$\mathcal{L} := \frac{1}{M}\sum_{m=1}^{M}\sum_{n=1}^{N}\left(\sum_{c=1}^{C}\omega_n^c\sum_{k=1}^{K}(1-\cos\theta_n^k)\right)\|\mathbf{d}_n^m\|_2^2. \tag{2}$$

To effectively embed geometric considerations, we factor in the difference between the estimated and ground-truth 3D pose angles. Let θ1, θ2, θ3 denote the angular deviations (yaw, pitch, roll), and let K = 3. The weighting function is designed to increase penalization as the deviation angle increases, encouraging the model to learn harder cases with more significant pose variation.

Additionally, to tackle attribute imbalance, each training sample is assigned to one or more attribute categories: profile-face, frontal-face, head-up, head-down, expression, or occlusion. A category-wise weight $\omega_{n,c}$ is introduced, which is inversely proportional to the frequency of category in the training data. This ensures that rarer attributes contribute more significantly to the loss, guiding the model to pay greater attention to underrepresented scenarios.

When both the geometry-aware and class-aware weighting terms are disabled, the loss formulation naturally falls back to the standard ℓ_2 form.

Although prior studies have incorporated 3D pose information to enhance model performance, the proposed loss function offers several distinct advantages. First, it achieves a coupled integration of 3D pose estimation with 2D distance measurement, which is more principled than treating the two aspects independently [28, 47]. Second, the formulation remains intuitive and computationally efficient, both in forward and backward propagation, offering advantages over more complex designs in the literature [46]. Third, the proposed approach facilitates a single-stage training pipeline, as opposed to cascaded frameworks [28, 48], thereby improving overall optimization effectiveness.

It is also noteworthy that the variable d_n^m is produced by the backbone network, while the pose-related component $\theta_{n,k}$ originates from the auxiliary subnetwork. These two branches are tightly coupled through the proposed loss function (see Eq. (2)), ensuring joint learning of landmark localization and pose estimation.

The complete architecture and its functional flow are illustrated schematically in *Figure 12*.

3.1.4. Training Configuration

Model training is carried out in PyTorch using mixed-precision on a CUDA-enabled GPU (or CPU fallback). The goal is to minimize the weighted pose-aware MSE while ensuring robust generalization across varied poses and occlusions. Training completes in 60–80 epochs, typically converging long before the 100-epoch limit.

Hyperparameter Settings

- Optimizer: Adam with weight decay = 1×10^{-5}
- Initial learning rate: 1×10^{-3}, decayed via cosine annealing to 1×10^{-5}
- Batch size: 64
- Epochs: up to 100
- Validation split: 10% of training data
- Early stopping: halt after 10 epochs without validation-loss improvement
- Gradient clipping: clip L_2 norm to 10
- Mixed precision: enabled on CUDA via torch.amp

At the start of each epoch, the training loop loads batches of face crops, corresponding landmarks, attribute flags, and ground-truth Euler angles. Each batch is first preprocessed (grayscale, normalized, resized) and then augmented on-the-fly with geometric, photometric, and filter transforms.

During the forward pass, the backbone–neck–head network predicts landmark coordinates, while the auxiliary branch regresses Euler angles for pose weighting. The combined loss is computed by multiplying the squared-error term by pose and attribute weights, then averaged across all landmarks and images in the batch.

Backpropagation proceeds under mixed precision: gradients are scaled, clipped, and the optimizer updates all trainable parameters. After completing all batches, a validation pass evaluates the frozen backbone–neck–head network (auxiliary branch disabled), computing loss and L_2 distance to monitor overfitting.

Checkpoints are saved automatically whenever validation loss decreases, retaining the three best models. Simultaneously, TensorBoard logs training and validation metrics, including loss curves, L_2 distance, and learning rate, providing real-time insights into convergence behavior.

If no improvement occurs for 10 consecutive epochs, training halts early. Final model weights are those from the checkpoint with the lowest validation loss, ready for quantization and deployment.

3.1.5. Algorithms Explaining the Complete Pipeline

The proposed approach has been comprehensively outlined through two distinct algorithms presented below. Initially, images underwent preprocessing procedures and were utilized for training across the complete dataset. Subsequently, the model developed in the initial phase was employed to identify facial landmarks with suitable precision levels. As demonstrated in *Figure 3.5,* input images accompanied by their corresponding pixel intensities were processed through resizing and normalization operations. Data augmentation strategies were subsequently implemented on the image collection to enhance overall model performance. The dataset was then partitioned into training and validation subsets, with the MobileNetV2 architecture being applied to these data portions. The Adam optimization algorithm was utilized for model compilation processes. In *Algorithm 2*, the previously trained model was then implemented for both static image classification tasks and real-time webcam applications, with the output displaying individual faces annotated with their respective landmark points.

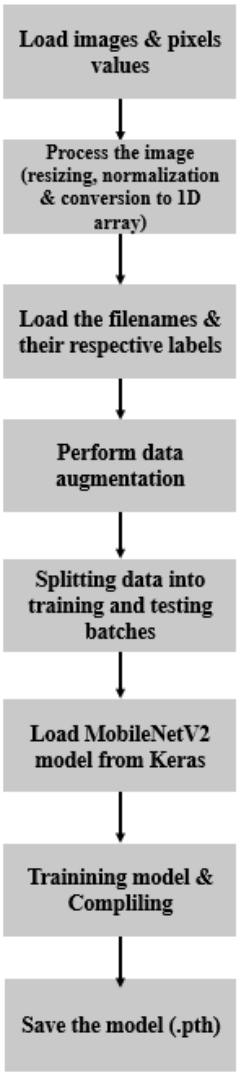

Figure 3.5. Pre-processing and Training on Dataset Workflow

The *Algorithm 1* describes a simple, interactive pipeline for running a facial-landmark (or face-classification) model on a live camera feed, with two main modes: one "per-face" step and one continuous real-time display.

Algorithm 1: Deployment of Facial Landmark Detector

If the face is detected **then**
    Get prediction from the face classifier model
    Show predictions and save resultant image
Else
    Show no output
End if
If the choice is classification in real-time **then**
    Load real-time feed from MTCNN
    Convert to grayscale image
    Read the feed frame by frame
    Perform inference using the model
    Show output in real-time feed
Else
    Show normal feed
End if
End stream when 'Esc' is pressed

The "Deployment of Facial Landmark Detector" first checks if a face is detected—if so, it runs the classifier to predict landmark positions, displays the annotated image, and saves it; otherwise, it shows no output. Next, based on whether real-time classification is selected, it either initializes a live MTCNN-based video feed, converts each frame to grayscale, performs inference frame by frame, and overlays the results in real time, or simply displays the unmodified feed. The loop continues until the user presses "Esc," at which point the stream is terminated.

3.1.6. Face and Facial Landmark Detection

After the completion of neural network training for facial landmark identification, this trained network can subsequently be deployed across any images containing human faces. Since the neural network requires input data in the form of a Tensor with specific dimensional requirements, facial detection necessitates initial preprocessing operations.

1. Identify all facial regions within an image utilizing a face detection algorithm (in this implementation, we will employ a Haar Cascade detector).

2. Transform the detected facial images through preprocessing steps, including grayscale conversion and reshaping into a Tensor format with dimensions matching the network's input specifications.

3. Deploy the trained model to identify facial landmarks within the processed image data.

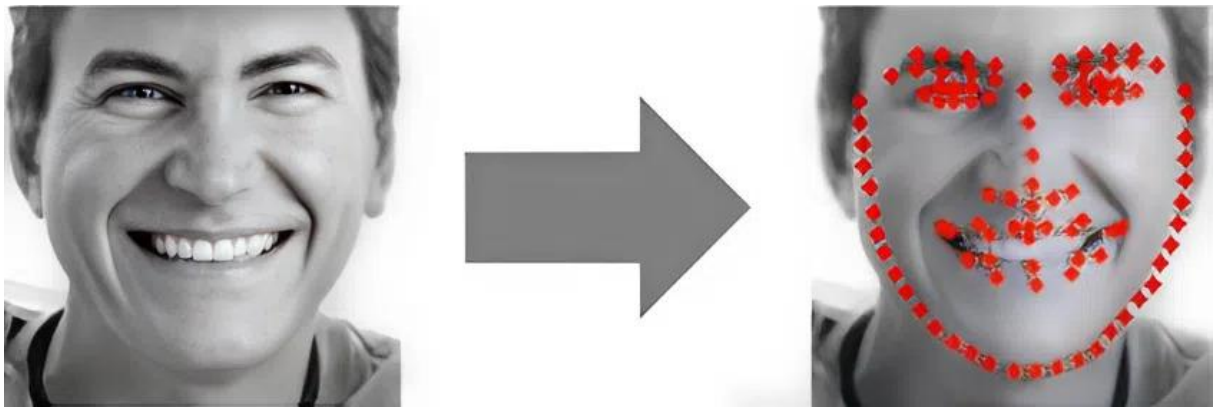

Figure 3.5. Inference facial landmark detection Pipeline

3.2. Quantization

The inference process is computationally intensive and requires a high memory bandwidth to satisfy the low-latency and high-throughput requirements of Edge applications.

Quantization and channel pruning techniques address these challenges while simultaneously achieving optimal performance and high energy efficiency with minimal degradation in accuracy. Through quantization, integer computing units become viable, and weights and activations can be represented with reduced precision. On the other hand, pruning reduces the overall required operations. The AMD Vitis AI quantizer includes the quantization tool, whereas the pruning tool is integrated into the Vitis AI optimizer.

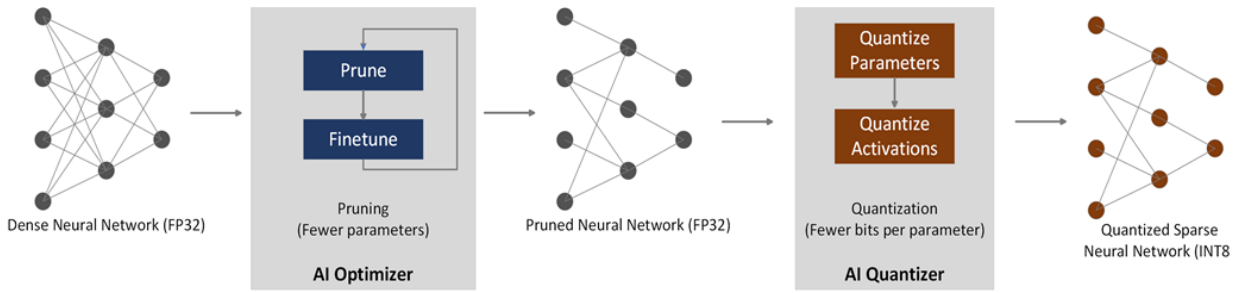

Figure 3.6. Pruning and Quantization Flow

The methodology adopted in this study follows a structured workflow illustrated in *Figure 3.7.* The approach employed in this research adheres to a systematic framework

depicted in *Figure 3.7*. This framework demonstrates the comprehensive process of model training, quantization implementation, and deployment procedures for machine learning systems on edge computing platforms. The workflow commences with raw data inputs that undergo preprocessing operations to generate an optimized, lightweight model. Subsequently, this model experiences two concurrent quantization approaches: post-training quantization (PTQ) and quantization-aware training (QAT). Within the PTQ pathway, the trained model undergoes direct quantization procedures to create the PTQ quantized model (QM), which is subsequently transformed into a PyTorch-compatible format. Conversely, in the QAT approach, the trained model first experiences quantization followed by retraining procedures utilizing supplementary training datasets to generate the QAT quantized model (QM), which undergoes a similar conversion to PyTorch format. Both resulting PyTorch models are then assessed and benchmarked for performance comparison. The superior-performing model is ultimately deployed onto edge computing hardware, exemplifying the complete pipeline from initial data preprocessing through final model deployment in edge computing environments.

This section concentrates on optimizing the computational efficiency of the MobileNetV2 architecture through the application of post-training quantization (PTQ) and quantization-aware training (QAT) methodologies.

The primary contributions of this research include:

1. We executed both post-training quantization (PTQ) and quantization-aware training (QAT) methodologies to optimize the computational efficiency of the MobileNetV2 architecture for edge computing deployment.
2. An extensive comparative analysis between the PTQ quantized model and the QAT quantized model was performed, examining their respective performance characteristics.
3. The optimal quantized model was successfully implemented on edge computing hardware, validating its practical applicability and computational efficiency in real-world deployment scenarios.

Proposed workflow

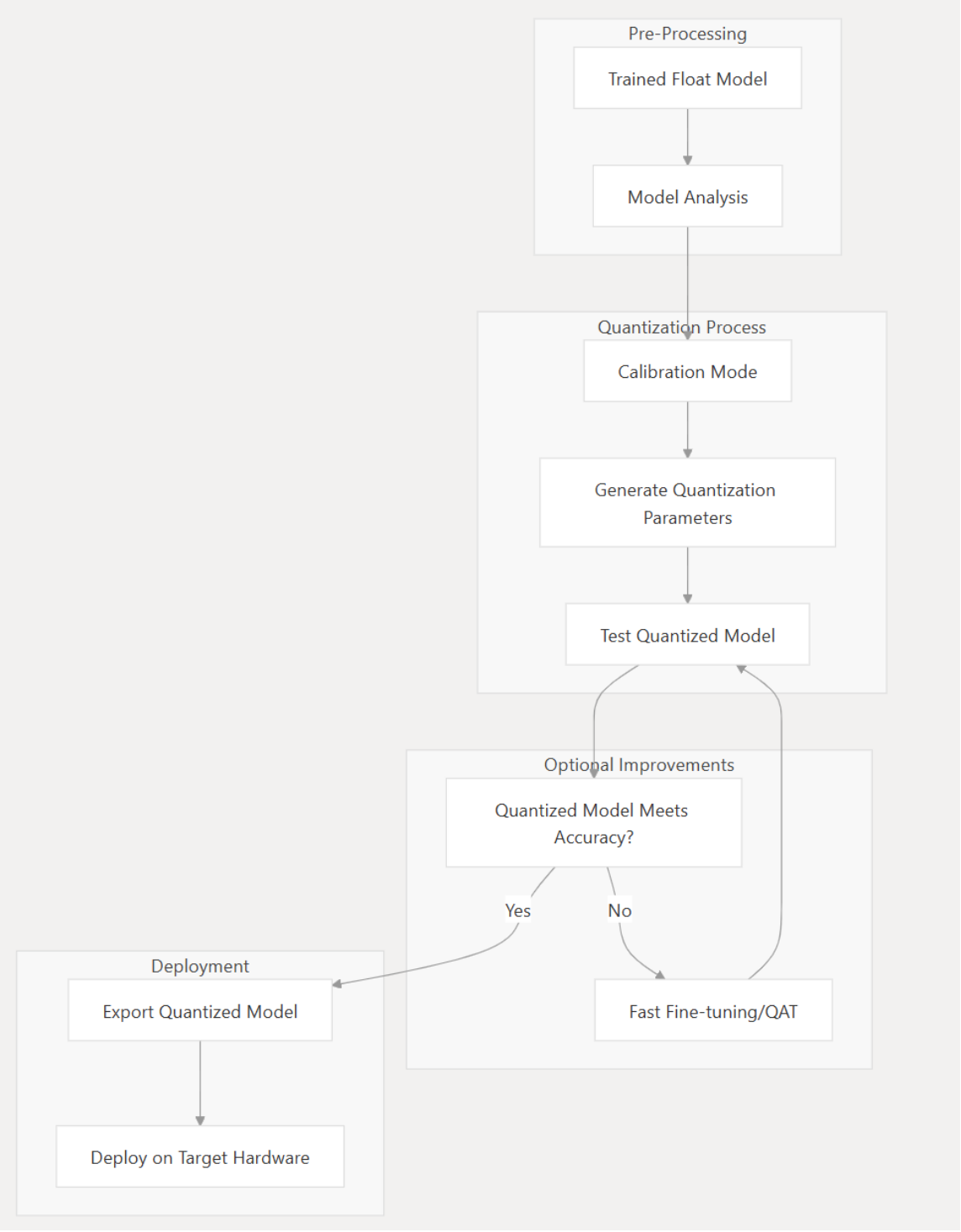

Figure 3.7. Quantization Workflow

3.2.1. Post-Training Quantization (PTQ)

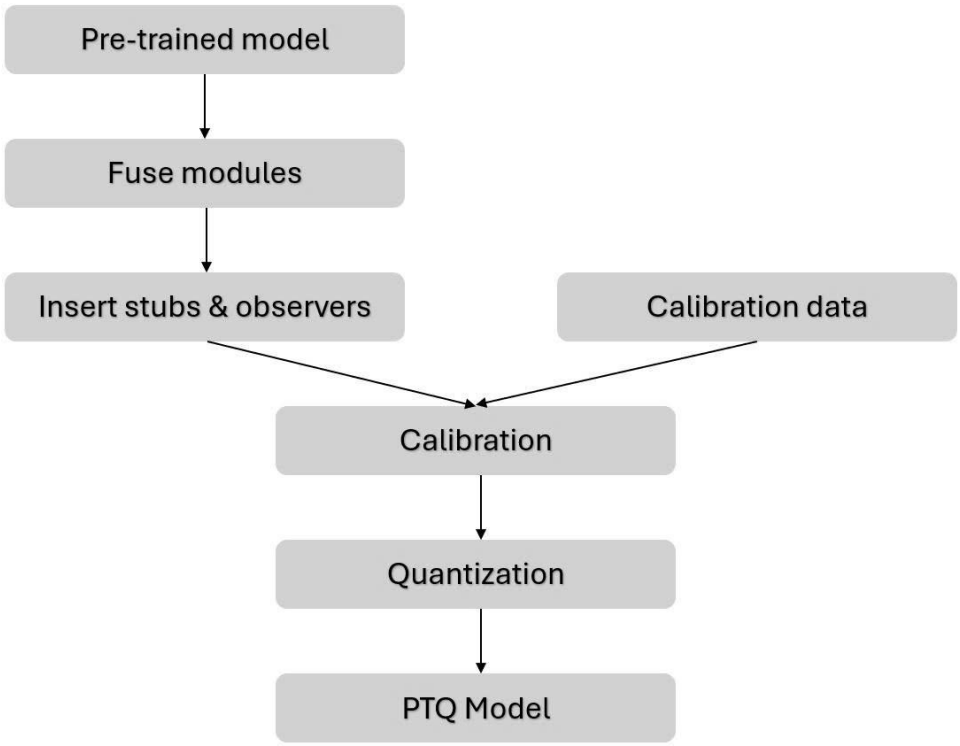

Figure 3.8. PTQ Model Flow

Dynamic range quantization represents a post-training quantization methodology that converts model parameters to 8-bit integer representations while maintaining activation values in floating-point precision. This quantization approach involves transforming floating-point numerical values into a finite set of integer representations based on the minimum and maximum parameter values within each network layer.

Modern neural networks often interleave convolution (or linear) layers with batch normalization and activation functions. To minimize rounding error and intermediate data movement, PTQ fuses sequences such as:

$$\text{Conv2d} \rightarrow \text{BatchNorm} \rightarrow \text{ReLU or Linear} \rightarrow \text{ReLU}$$

into a single fused operation. This eliminates redundant float-to-int conversions between layers and shrinks the computational graph.

After fusion, the model's behavior remains identical, but the kernels are now tailored for quantized execution.

After fusion, we insert observers into each layer to gather statistics on the activation and weight distributions during inference. Each observer records two key values the minimum (x_min) and maximum (x_max) of the tensor; in the case of per-channel quantization, it tracks these values separately for each output channel of a Conv or Linear layer. Then assign

a default quantization configuration to the entire model and call:

torch.quantization.prepare(model, inplace=True)

Next, run each batch from a calibration dataset - a smaller but diverse and representative subset of your training data through the model so that the observers update their recorded x_min and x_max for every layer. Once the observers have collected each tensor's minimum (x_min) and maximum (x_max) values, we move on to computing the quantization parameters.

$$\text{scale} = \frac{x_{\max} - x_{\min}}{2^b - 1},$$
$$\text{zero\_point} = \text{round}\left(-\frac{-x_{\min}}{\text{scale}}\right)$$

Here b is for an INT8 representation, which gives a value range of

- [0,255] in asymmetric mode (zero_point may be nonzero)
- [−127,127] in symmetric mode (zero_point is forced to 0).

When quantizing convolutional or linear weights, repeat the same calculations for each **output channel** independently. This respects the fact that different channels may use very different dynamic ranges, and typically yields lower overall quantization error.

With scale and zero_point in hand, every floating-point value x is mapped to an integer q via:

q = clamp(round(x / scale + zero_point), Q_min, Q_max)

where Qmin , Qmax define the integer range (e.g., 0 to 255 for uint8, -127 to 127 for int8).

If we ever need to inspect quantization error offline, we can "dequantize":

x_deq = scale * (q - zero_point)

The pseudocode *Algorithm 2* below describes a PTQ workflow in PyTorch, which prepares a floating-point model to be converted into an efficient INT8 version.

Algorithm 2: Dynamic Range Quantization (8-bit)

```
// Fuse Conv/Linear + BatchNorm + ReLU patterns
M_fused ← FuseLayers(M)
for each layer L in M_fused.layers do
    InsertObserver(L)
end for
// Calibration pass: collect min/max of activations
for each batch x in D_cal do
    M_fused.forward(x)    // observers update their x_min, x_max
end for
for each layer L in M_fused.layers do
  if L.type is Conv or Linear then
      for each output channel c in L do
          w ← L.weights[c]
          w_min ← min(w);  w_max ← max(w)
          scale_w[c] ← (w_max − w_min) / (2^b − 1)
          zero_w[c]  ← −round(w_min / scale_w[c])
          w_q[c]     ← clamp(round(w / scale_w[c]) + zero_w[c],
                           Q_min, Q_max)
      end for
    end if
    x_min ← Observer[L].min;  x_max ← Observer[L].max
    scale_x ← (x_max − x_min) / (2^b − 1)
    zero_x  ← −round(x_min / scale_x)
    L.activation_q_params ← (scale_x, zero_x)
end for
M_q ← ConvertToQuantized(M_fused)
return M_q
```

The “Dynamic Range Quantization (8-bit)” procedure comprises four main stages, each designed to preserve accuracy while enabling fast INT8 execution:

1. Layer Fusion (lines 1–2)

- **Purpose:** Merge each Conv/Linear + BatchNorm + ReLU sequence into a single fused block. This folds BatchNorm's scale and bias into the preceding weights and applies the ReLU in one shot, reducing runtime overhead and ensuring that quantization sees the true end-to-end behavior of each layer.

2. Observer Insertion & Calibration (lines 4–6 and 8–10)

- **How it works:**
  1. After fusion, attach a lightweight "observer" to every layer (lines 4–6).
  2. Run a forward pass over a representative calibration dataset Dcal (lines 8–10). During this pass, each observer records the minimum and maximum activation values seen at its layer.
- **Why:** Using real min/max statistics ensures that each layer's activation range reflects actual data distributions, rather than arbitrary presets.

3. Weight Quantization (lines 16–21)

- **Per-channel processing:** For each Conv or Linear layer, iterate over its output channels:
  1. Read the full-precision weights and find their min/max (lines 16–17).
  2. Compute a scale factor and zero-point to map floats into the INT8 range.
  3. Round and clamp each weight into [Qmin, Qmax] (lines 18–21).
- **Advantage:** Per-channel quantization retains finer granularity, reducing accuracy loss compared to per-tensor schemes.

4. Activation Quantization & Model Conversion (lines 24–27 and 30–31)

- **Activation parameters:** Use the observers' recorded min/max to calculate each layer's activation scale and zero-point (lines 24–27), storing them in activation_q_params.
- **Final conversion:** Once every layer has weight and activation quant params, ConvertToQuantized replaces the fused blocks and observers with native INT8 kernels, producing the fully quantized model ready for efficient deployment (lines 30–31).

3.2.2. Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) extends the static Post-Training Quantization process by simulating quantization effects during gradient-based learning as shown in *Figure 3.9*, enabling the network to adapt its weights for minimum precision loss when finally converted to integer arithmetic. Starting from a fully trained 32-bit floating-point

model, the first step is module fusion: convolution (or linear) layers, batch-normalization, and activation functions that occur in sequence such as

Conv→BatchNorm→ReLU or Linear→ReLU

They are merged into single computational kernels. This fusion reduces intermediate data movement between layers, minimizes rounding errors, and prepares the graph for quantization stubs.

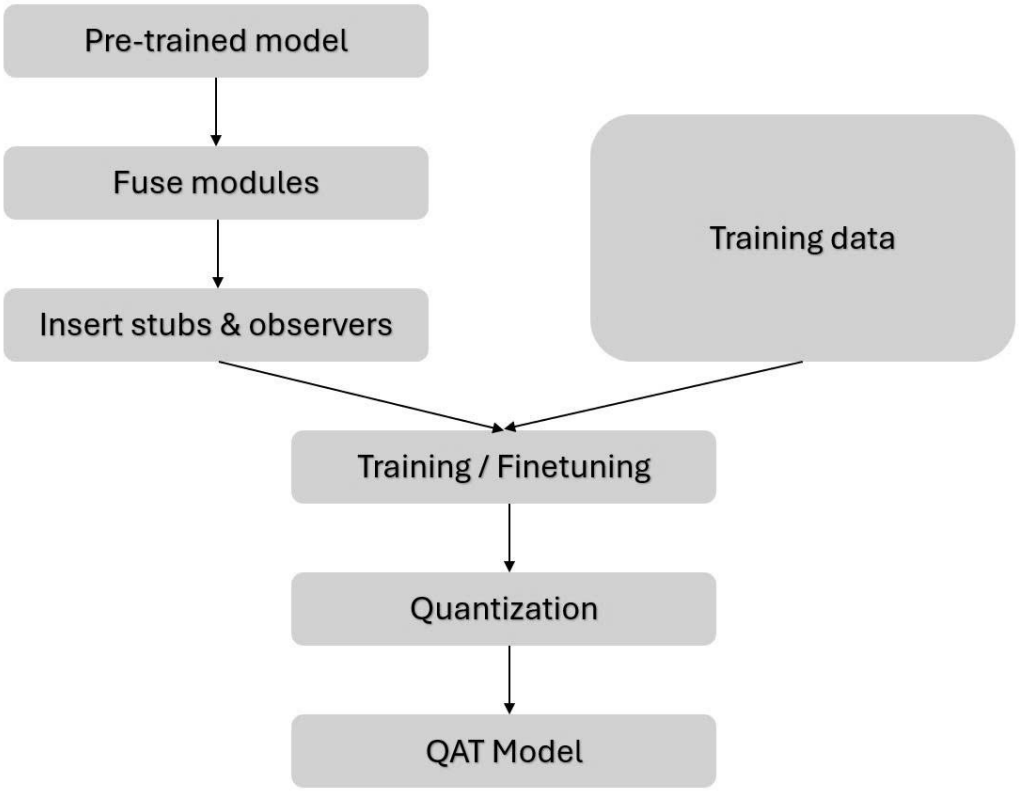

Figure 3.9. QAT Model Flow

Once fusion is complete, every quantizable layer is wrapped with fake-quantization "stubs" and observers. The observers track the minimum and maximum values of each activation and weight whenever a mini-batch is passed through the network. Simultaneously, the fake-quant modules clamp, round, and then "unclamp" activations and weights in the forward pass to mimic 8-bit integer behavior, while gradients flow through unchanged thanks to the straight-through estimator. Embedding these operations directly into the training loop exposes the model to quantization noise and out-of-range clipping during weight updates, so that by the end of QAT, the learned parameters are inherently more robust to low-precision arithmetic.

With fake-quantization and observers in place, the network enters a fine-tuning phase on the full training set (or a suitably representative subset). Learning rates are typically reduced to a fraction of their original values to prevent large parameter oscillations, and training proceeds for a modest number of epochs until validation accuracy converges. Throughout this process, the observers continuously refine their recorded ranges, allowing

the quantization parameters, scale factors, and zero points to track shifting weight distributions. This dynamic adaptation dramatically reduces the accuracy gap compared to naïve static quantization.

Finally, training culminates in the removal of all fake-quantization stubs and the conversion of each layer into its true integer counterpart. Conv2d becomes QConv2d, Linear becomes QLinear, ReLU becomes QuantizedReLU, and so on, solidifying the 8-bit representations that were simulated during training.

The resulting QAT model delivers near-full-precision accuracy while benefiting from roughly a 4× reduction in model size and substantial speedups on hardware that natively supports INT8 operations. A thorough evaluation then measures top-1 or regression metrics on the target dataset, end-to-end latency and throughput on the deployment platform, memory footprint, and hardware utilization. Advanced techniques such as progressive layer-by-layer quantization, mixed-precision overrides for especially sensitive layers, or calibration guidelines that discard outliers via histogram-based range selection can further fine-tune the tradeoff between precision and performance, yielding a deployment-ready model that fully leverages the advantages of 8-bit inference.

The pseudocode *Algorithm 3* below describes a QAT workflow in PyTorch, which prepares a floating-point model to be converted into an efficient INT8 version without losing much accuracy.

Algorithm 3: Quantization-Aware Training

```
// ——— Load and Prepare Model ———
model ← load_pretrained_fp32_model()
model.eval()
fuse_layers(model, [
    [Conv, BatchNorm, ReLU],
    [Linear, ReLU],
  ...
])
// ——— Configure QAT ———
model.qconfig ← choose_qat_qconfig(backend="fbgemm")
model.train()
insert_fakequant_and_observers(model)
 optimizer ← SGD(model.parameters(), lr=initial_lr × lr_factor, momentum=0.9)
scheduler ← CosineAnnealingLR(optimizer, T_max=total_epochs)
for epoch in 1 to total_epochs do
    for (x_batch, y_batch) in training_data_loader do
        optimizer.zero_grad()
        y_pred ← model(x_batch)
        loss ← compute_loss(y_pred, y_batch)
        loss.backward()
        optimizer.step()
    end for
    scheduler.step()
    if validation_condition(epoch) then
        val_metrics ← evaluate(model, validation_loader)
        log(val_metrics)
    end if
end for
model.eval()
qat_model ← convert_to_int8(model)
```

The “Quantization-Aware Training” procedure proceeds in four main phases:

1. **Load and Prepare the Float-32 Model**

First, a pretrained full-precision (FP32) network is loaded and placed in evaluation mode to stabilize running statistics. Before training, compatible layers are “fused” together (e.g., convolution + batch-normalization + ReLU, or linear + ReLU) so that, during quantization, scale and bias factors can be merged into preceding weights—this helps preserve accuracy when weights and activations are later quantized.

2. **Configure the Model for QAT**

The algorithm assigns a QAT configuration (qconfig) appropriate for the target backend (here, “fbgemm,” which is optimized for x86 CPUs). Switching the model into training mode enables the insertion of “fake-quantization” modules and observers: these mimic the effects of 8-bit quantization during the forward pass, while still using full-precision parameters for gradient updates. An SGD optimizer is created (with momentum = 0.9 and a learning rate scaled by lr_factor), and a cosine-annealing learning-rate scheduler is set up to gradually reduce the learning rate over the total number of epochs.

3. **Train with Simulated Quantization**

For each of the specified total_epochs:

- Iterate over mini-batches (x_batch, y_batch) from the training loader.
- Zero out gradients, run a forward pass through the QAT-augmented model, compute the loss, backpropagate, and step the optimizer.
- After completing all batches in an epoch, advance the learning-rate scheduler.
- Optionally (e.g., every few epochs), evaluate on a validation set to compute metrics (accuracy, loss, etc.) and log them for monitoring.

4. **Convert to an 8-bit Model for Deployment**

Once training is complete, the model is switched back to evaluation mode to freeze running statistics, and a final conversion is performed that replaces the fake-quantization ops with true integer quantization (INT8) in both weights and activations. The resulting qat_model is now ready for efficient, low-precision inference on hardware or in optimized runtimes.

CHAPTER 4: RESULT & DISCUSSION

4.1. Training Model

4.1.1. Experimental Result

The experimental procedures were executed using a laptop system featuring an i9 CPU operating at 4.1 GHz, combined with 16 GB of RAM capacity. For developing and executing various experimental procedures, this study utilized the Jupyter Notebook platform running a Python 3.8 kernel environment.

The configuration parameters established for the MobileNetV2 experimental setup include:

• Training/validation split: 0.1

• Training epochs: 100

• Batch size: 64

• Optimizer: Adam

• Learning rate schedule: 0.001 for the first 20 epochs, 0.0005 for the next 10 epochs, and 0.00025 until the end

• Loss function: categorical cross-entropy

• Metric: accuracy

• Alpha=auto_po2

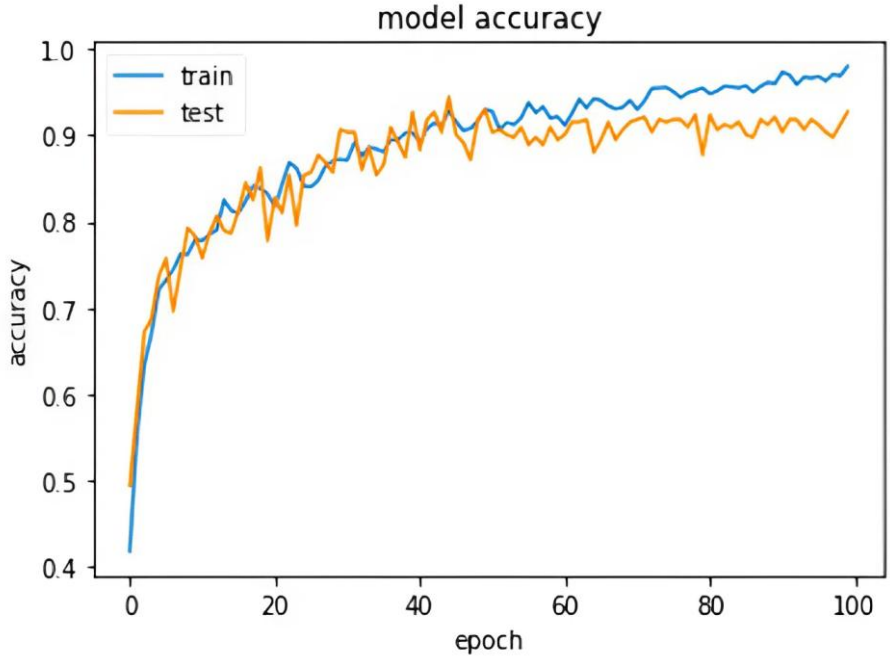

Figure 4.1: Accuracy of MobileNetV2 over epochs

4.1.2. Evaluation

These chart above displays the model's training and testing performance. In the accuracy chart, both the training and test accuracy show a clear upward trend, stabilizing towards the later stages of training. Although the validation accuracy is slightly lower, it remains relatively stable. The test accuracy exhibits some fluctuation in the early and middle stages, suggesting that there might have been some disturbances during the training process, while the training accuracy remains more consistent. Overall, accuracy steadily improves as training progresses, and around 60 epochs, the model shows signs of convergence, especially with the gap between training and test accuracy narrowing, indicating good generalization ability.

In the loss chart, the loss values for both the training and test sets decrease steadily from higher values, while the validation loss also decreases but levels off in the later stages, remaining higher than the training and test losses. Notably, the test loss falls below the training loss in the later stages, suggesting that the model performs better on the test set than the training set. However, the test loss fluctuates significantly, particularly during the early and middle stages, which could be due to variations in data distribution or instability during training. Overall, the model's loss decreases steadily, accuracy improves, and the results become stable, demonstrating effective training and convergence.

4.2. Quantization

4.2.1. Results

After performing quantization using Vitis AI, we utilized the Netron tool to inspect the model weights and observe the changes. This allowed us to visually compare the original FP32 (32-bit floating point) format with the quantized INT8 (8-bit integer) format.

To assess how post-training quantization (PTQ) and quantization-aware training (QAT) affect our baseline CNN (6.59 MB, 92.42 % FP32 accuracy), we applied each quantization strategy and recorded four metrics: model size, inference accuracy, end-to-end pipeline time, and robustness relative to the unmodified network. The comparative results are presented in *Table 6*.

Table 6: Quantized method results

Quantization Method	Accuracy (%)	Model Size (MB)	Quantization Time	Compression Ratio
Original (FP32)	92.42	6.59	-	1x
QAT (INT8)	91.74	1.65	Approximately 3 hours (fine-tune)	4x
PTQ (INT8)	90.94	1.65	Approximately 10 minutes (calibration)	4x

The comparison clearly shows that while the full-precision Float model retains the highest accuracy (92.42 %) and "Very High" robustness, it comes at the cost of a large 6.59 MB footprint and requires no quantization time. Quantization-Aware Training (QAT) reduces model size by 75 % (to 1.65 MB) with only a modest accuracy drop to 91.74 %, and achieves "High" robustness by exposing the network to quantization effects during training, though it demands a substantial fine-tuning investment of roughly three hours. Post-Training Quantization (PTQ) matches QAT's compact 1.65 MB size and can be completed in around ten minutes, making it ideal for rapid deployment; however, its accuracy falls further to 90.94 % and its "Medium" robustness means it is more sensitive to distribution shifts. In practice, QAT is the preferred choice when both accuracy and resilience to varied inputs are critical, whereas PTQ offers a fast, resource-light solution when development speed outweighs the smallest performance penalty.

Generated Output

- Arch.json: Describes the target DPU (Deep-learning Processing Unit) architecture and resource allocation.
- pflb_detector.xmodel: The quantized model in Xilinx's proprietary xmodel format, ready to be loaded onto the FPGA's DPU.
- pflb.yaml: Configuration file specifying I/O tensor names, shapes, data types, and runtime parameters for inference.
- README.md: Usage instructions: environment setup, deployment steps, and example inference commands.

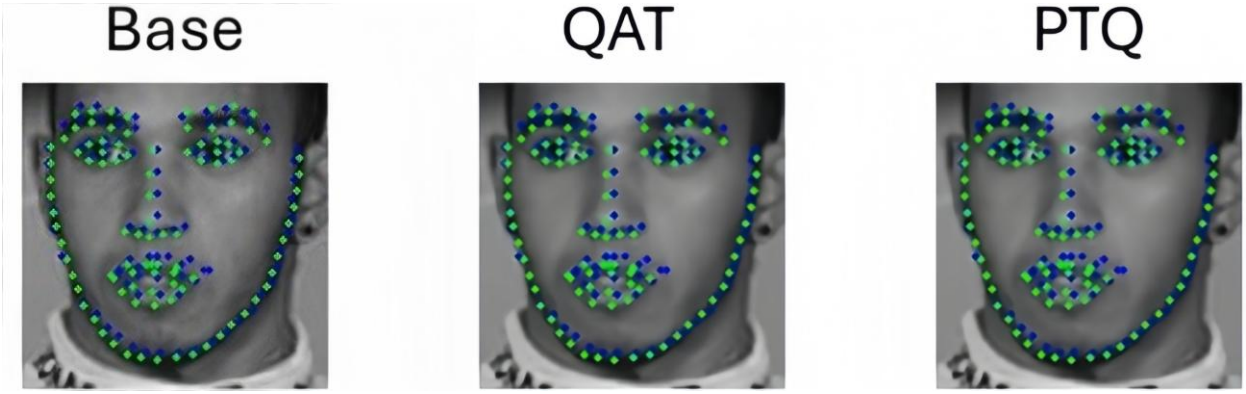

Figure 4.2: Visualization images after quantization by QAT and PTQ with green landmarks is groud truth and blue landmarks is predicted

4.2.2. Evaluation

Table 7 highlights the clear trade-offs between PTQ and QAT. PTQ delivers rapid, minutes-long quantization using only a small calibration set, making it highly efficient to integrate and scale, simply rerunning calibration when the model changes. However, it incurs a larger accuracy drop (1–2%), and its robustness is only moderate. In contrast, QAT demands full access to the training and validation data and requires hours of fine-tuning, alongside more complex integration steps, but achieves a minimal accuracy loss (0.2–0.5%) and greater resilience to input variations. Both methods yield similar inference speeds on INT8 hardware, so the choice ultimately depends on whether development speed or peak performance and stability are the top priority.

Table 7: Comparison and evaluation of PTQ vs. QAT

Criteria	PTQ	QAT
Accuracy drop	1.0 - 2.0 %	0.2 - 0.5 %
Retraining required	No	Yes (fine-tuning)
Iteration	Single-pass	Multi-pass
Integration complexity	Low	High
Data requirement	Calibration set only	Full training/validation set

4.3. Target Platforms Deployment

The primary objective of this framework is to detect and recognize objects positioned within the camera's field of view. The PFLD dataset, as previously described, was employed for model training and calibration procedures outlined in the preceding chapter.

This dataset comprises approximately 80,000 pre-annotated images with comprehensive information ready for training purposes. Additionally, this dataset has been extensively utilized across numerous research initiatives. The training phase utilized 75,000 images, while the remaining 2,500 images were reserved for calibration activities performed throughout the quantization process.

For this experimental investigation, a single deep neural network architecture, specifically MobileNetV2, was implemented. This choice was made to incorporate diverse performance characteristics and architectural variations within the research framework. Each configuration underwent evaluation using all models in a standardized testing environment through three 60-second test cycles, extending to 20-minute sessions when irregularities were detected.

The experimental methodology incorporated three distinct hardware acceleration platforms: NVIDIA's Jetson Nano single-board GPU system, a Raspberry Pi Embedded Computer, AMD's Kria SOM KV260 FPGA-based platform, an Intel Core i9 CPU, and an RTX-4060 GPU. These accelerators functioned both as edge computing solutions and hosted the required operating systems for each respective case. The rationale for utilizing them as independent units stems from the necessity to accurately assess power consumption and thermal characteristics without external system interference.

A single webcam sensor with 1920x1080 resolution configured as an RGB imaging device was deployed throughout this investigation. This camera interfaced with each platform via USB connectivity, requiring the platform to process video streams, execute inference operations, and present results on the connected display. Display connectivity was established through HDMI connections to all platforms. Power supply for all platforms was provided through AC adapters utilizing barrel connectors, each operating at platform-specific voltages as detailed in *Table 8*.

Table 8: Voltage of platforms

Platform	Voltage	OS
Jetson Nano	5V	Ubuntu 20.04
Raspery	5V	Ubuntu 20.04
Kria KV260	12V	Petalinux
RTX 4060	12V	Window
CPU	12V	Window

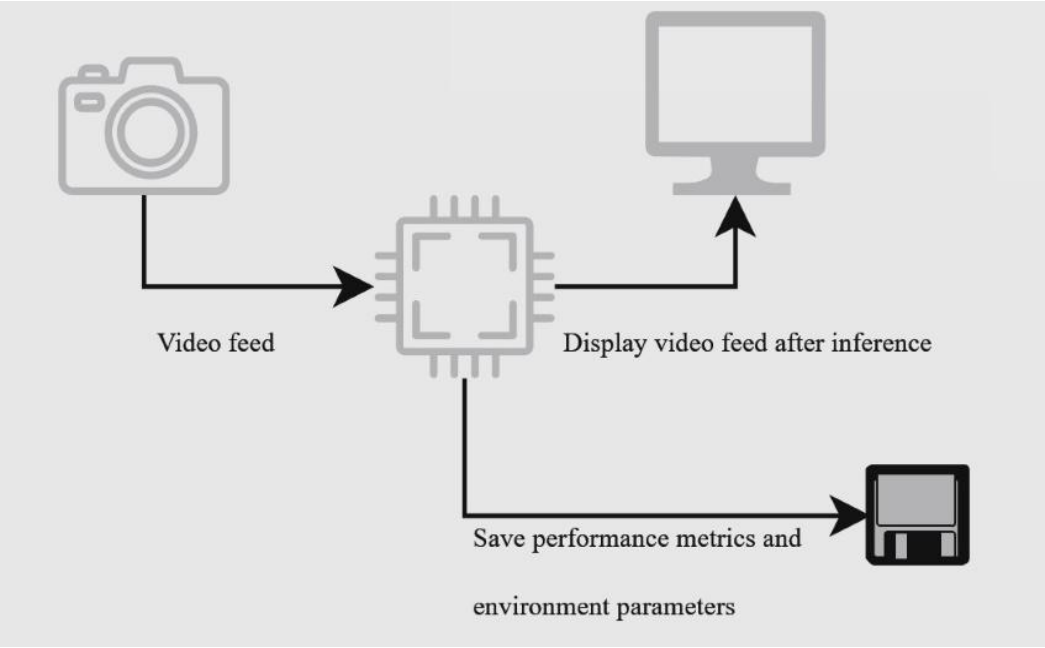

Figure 4.3. System workflow

The operating systems deployed across each hardware platform are documented in *Table 8.*

During the model training stage, pre-trained neural networks developed using the comprehensive Microsoft PFLD dataset were utilized. Consequently, for the Kria KV260 implementation, quantization procedures were executed using the Vitis AI framework. The Vitis AI development environment provides three separate frameworks for quantization deployment: ONNX, TensorFlow, and PyTorch. PyTorch was chosen as the preferred framework owing to its extensive community support and comprehensive documentation resources. Following the successful integration of the PyTorch framework into the development environment, the quantization methodology detailed in the previous chapter was implemented.

The resulting optimized model underwent compilation within the designated environment for the KV260's Deep Processing Unit (DPU), specifically identified as DPUCZDX8G according to established naming conventions. A detailed breakdown of the inference methodology for Jetson Nano, Raspberry Pi, KV260, and CPU platforms is provided in *Algorithms 4* and *5*, while the RTX 4060, Raspberry Pi, and CPU implementations follow identical procedures to the Jetson Nano approach [35].

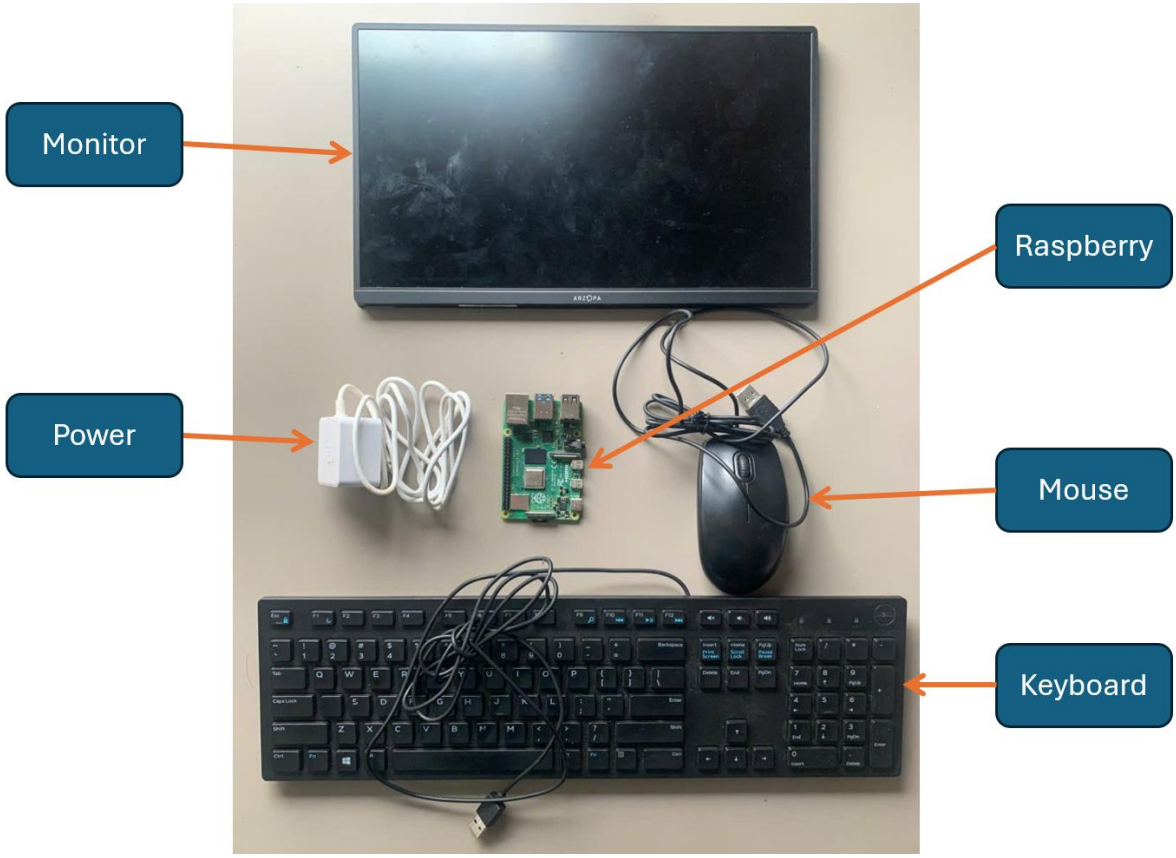

Figure 4.4. Raspberry Setup Environment

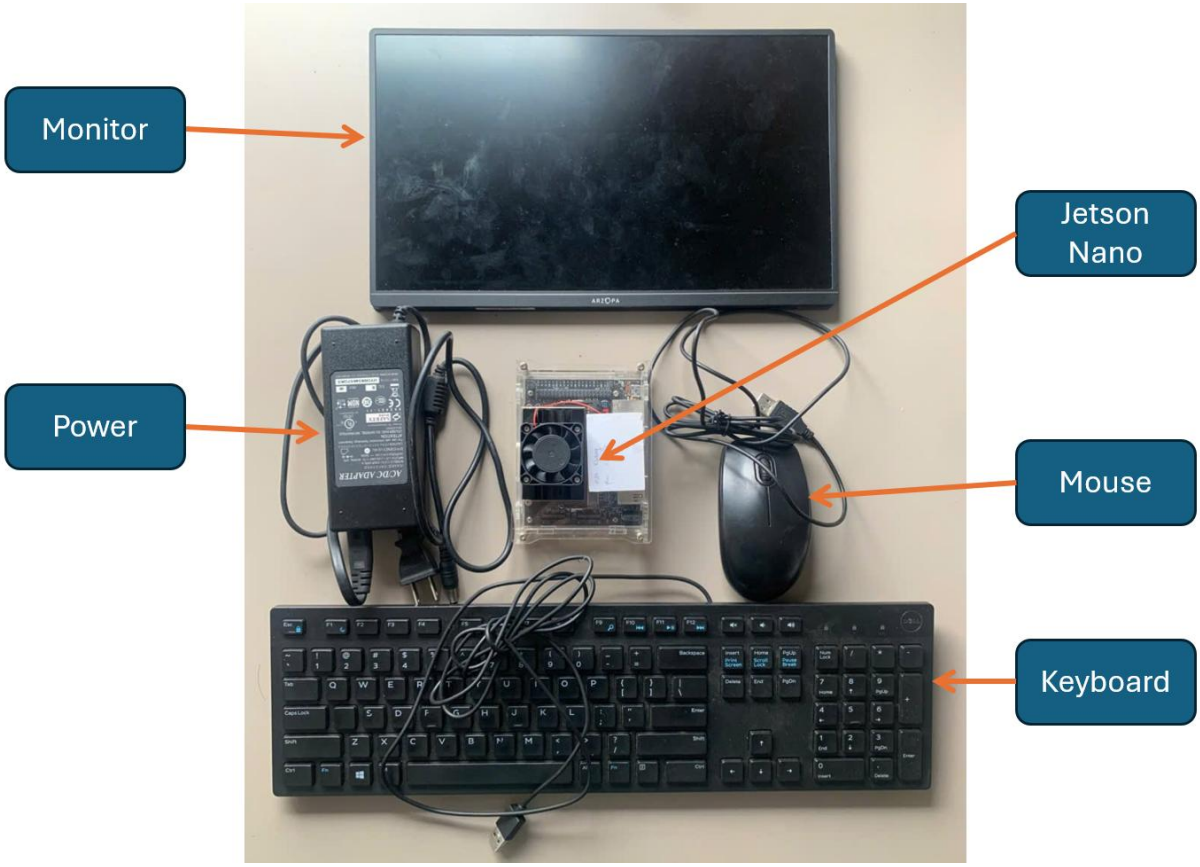

Figure 4.5. Jetson Nano Setup Environment

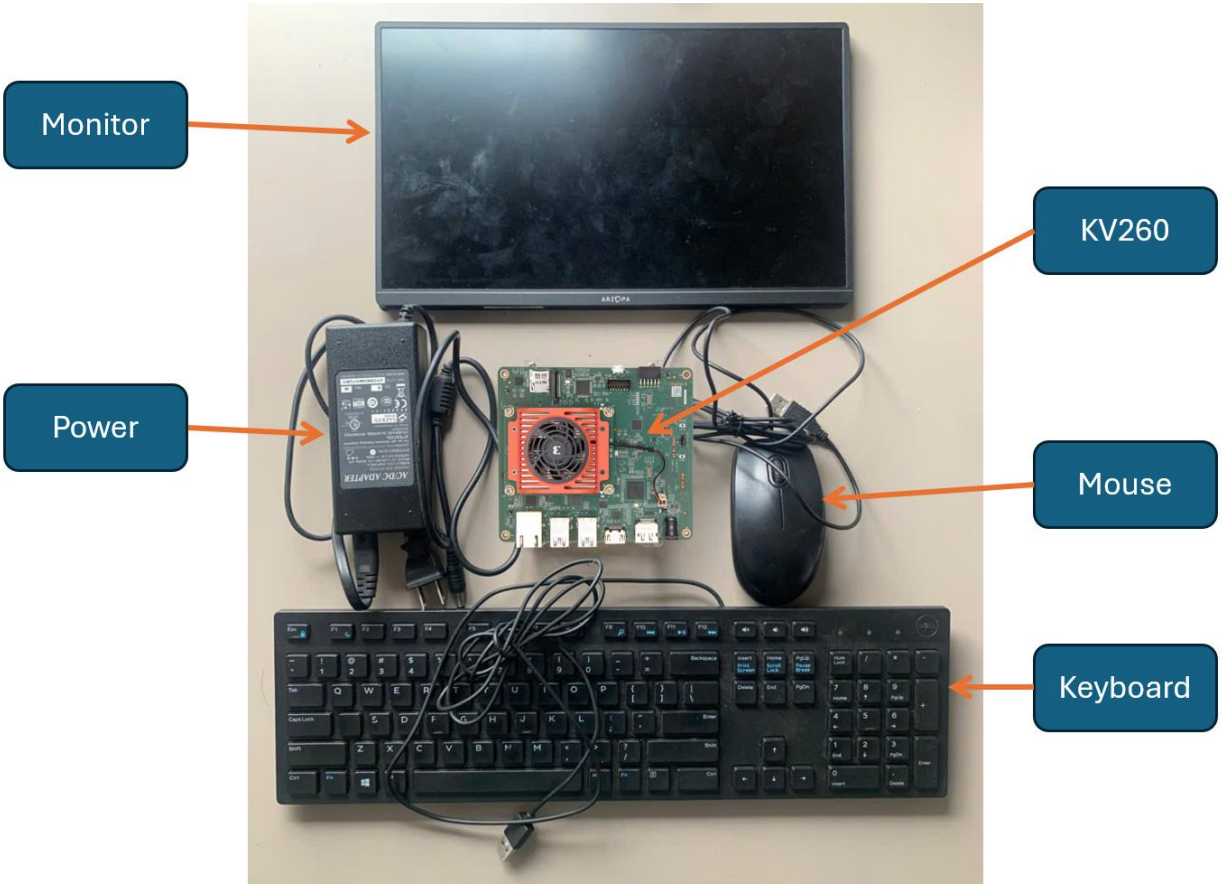

Figure 4.6. KV260 Setup Environment

The image is a piece of pseudocode titled “Algorithm 4 Jetson Nano inference” that outlines, step by step, how to run a live, camera-based inference loop on a Jetson Nano using CUDA-accelerated PyTorch.

ALGORITHM 4: Jetson Nano inference

1: model ← trained_model
2: device ← cuda
3: Load the trained model to the device.
4: Initialize the pipeline.
5: Get camera input as a source.
6: Connect to the device and start the pipeline.
7: **while** True **do**
8: Convert the Grayscale frame to a PyTorch tensor.
9: Load the tensor to the GPU.
10: Perform inference using the model.
11: Calculate FPS.

12: if model == MTCNN then

13: Identify the face and draw a box.

14: else

15: if model == MobileNetV2 then

16: Facial landmark detection

17: **end if**

18: **end if**

19: **end while**

The “Jetson Nano inference” pseudocode begins by assigning the variable model to the pre-trained network and setting device to cuda so it will run on the Nano’s GPU. Next, the code loads the model onto the GPU and initializes the processing pipeline to handle camera input. Once the pipeline is connected and started, the program enters an infinite loop: for each incoming frame, it first converts the image to grayscale and wraps it in a PyTorch tensor, then transfers that tensor to the GPU. The model then performs inference on the tensor, and the code computes the current frames-per-second (FPS) to monitor performance. Finally, depending on which network is in use, if the model is MTCNN, it detects faces and draws bounding boxes; if the model is MobileNetV2, it runs facial-landmark detection the loop repeats for the next frame.

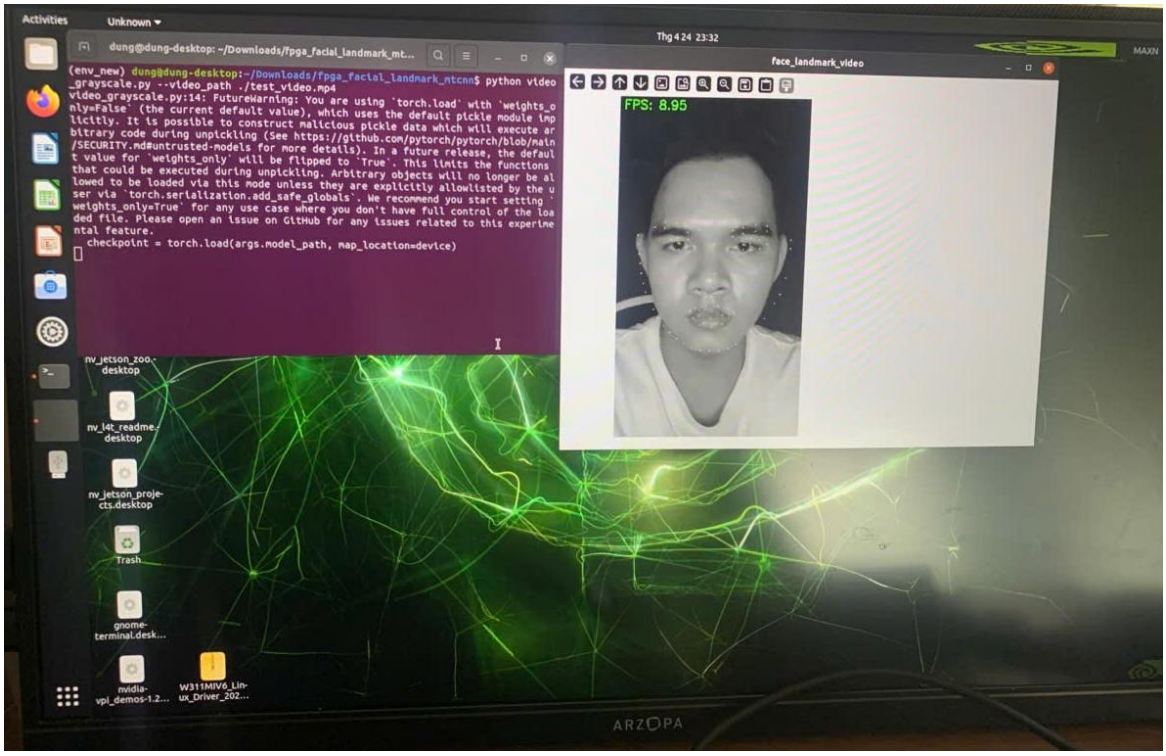

Figure 4.7. Jetson Nano Inference

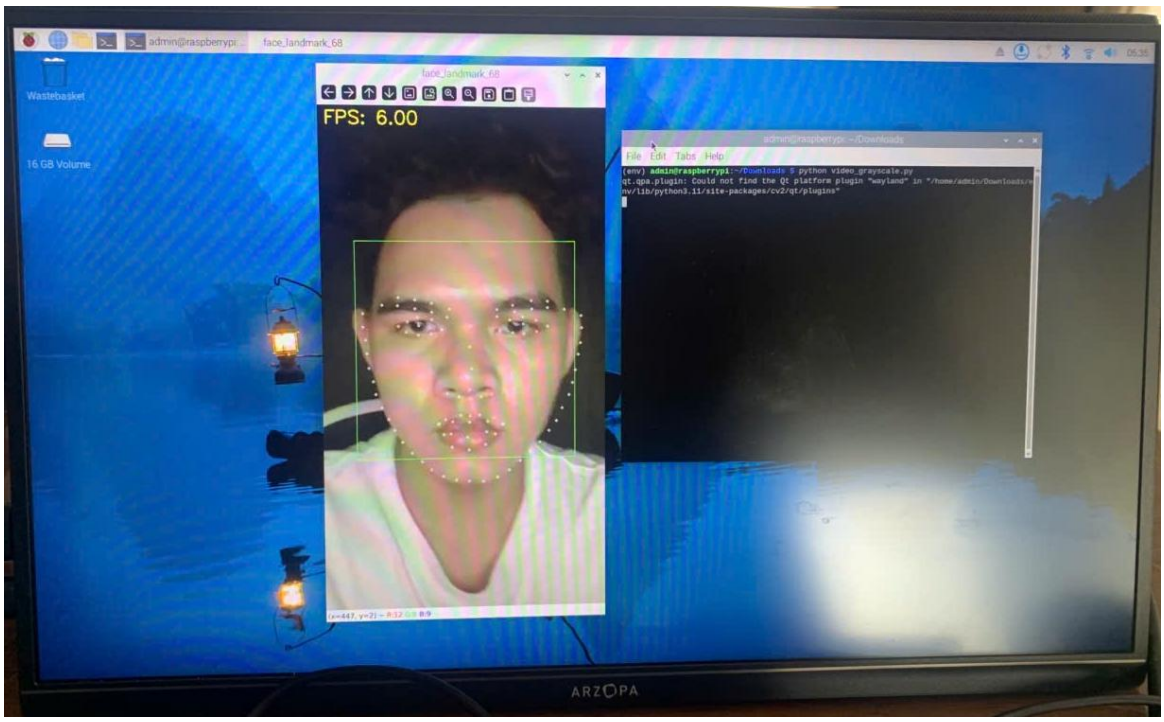

Figure 4.8. Raspberry Inference

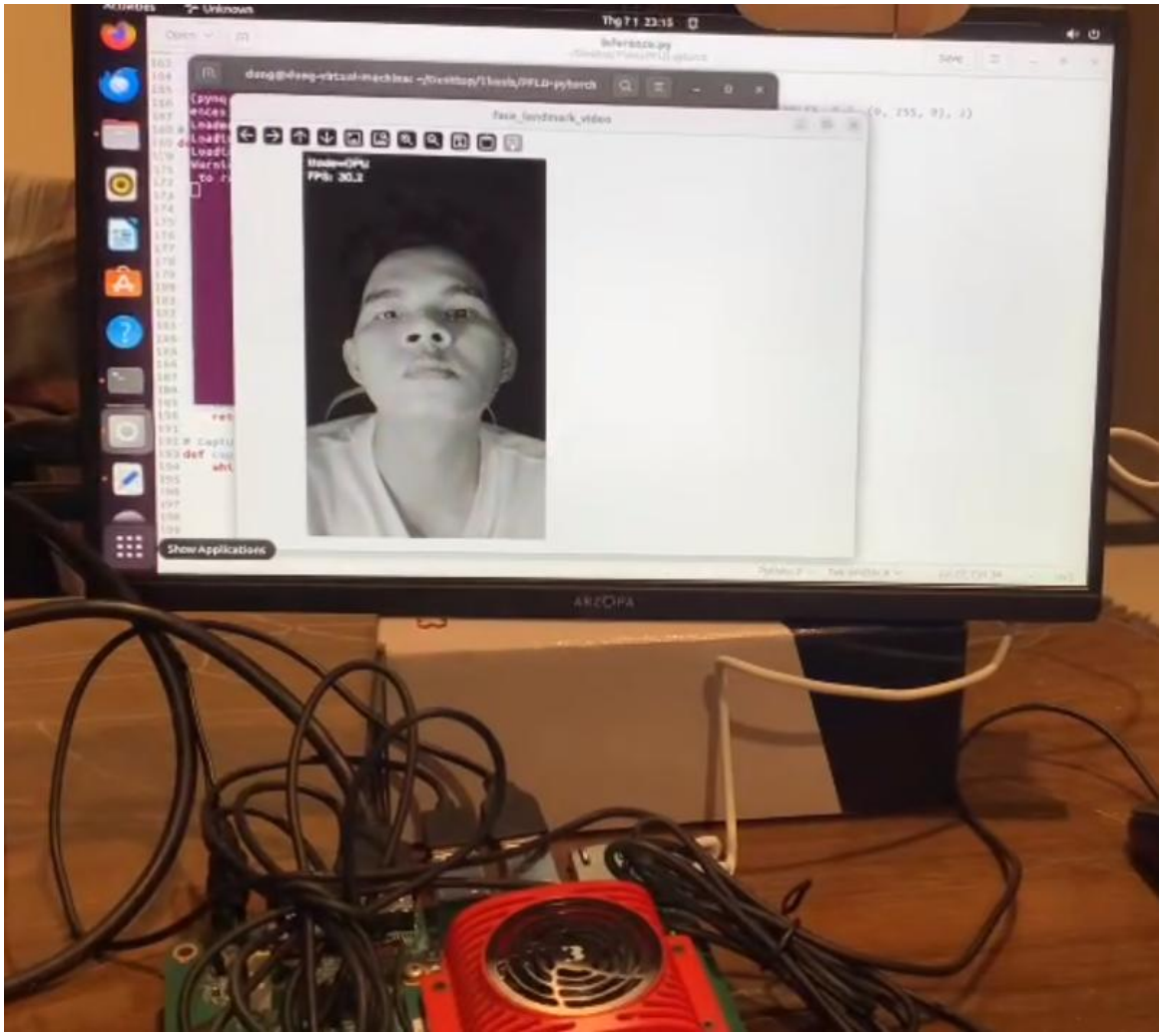

Figure 4.9. KV260 Inference

ALGORITHM 5: Kria KV260 inference

```
Load Xmodel.
Get camera input as a batch of images.
while True do
    Perform inference using the model.
    Calculate FPS.
    if model == MTCNN then
        Identify the object and draw a box.
    else
        if model == MobileNetV2 then
            Identify the object.
        end if
    end if
end while
```

The “Kria KV260 inference” pseudocode starts by loading the compiled Xmodel onto the DPU accelerator. It then captures a batch of images from the camera as input data. Once the data stream begins, the algorithm enters an endless loop: in each iteration, it runs the DPU inference on the current batch, then computes the frames-per-second (FPS) to track throughput. After inference, it checks which network is in use. If the model is MTCNN, it performs object (face) detection and overlays bounding boxes on the output; if the model is MobileNetV2, it carries out a simpler object-identification step before immediately proceeding to process the next batch.

For collecting experimental metrics, including thermal readings, power consumption, and memory utilization, platform-specific monitoring utilities were utilized. Both the Jetson Nano and Kria KV260 incorporate dedicated integrated circuits designed for measuring these parameters and offer application programming interfaces that enable users to retrieve data through built-in monitoring applications. Conversely, for the RTX 4060 evaluation, NVIDIA's containerized environment was implemented, which delivers comprehensive GPU performance statistics and operational data.

4.3.1. Experimental Result

Performance assessment of the trained PFLD model was conducted across four designated hardware platforms, with evaluation metrics encompassing model dimensions, execution memory requirements, energy consumption, landmark detection precision, mean

inference duration, and operational frame rate with the support of psutil for calculating Power consumption and Memory usage; Pytorch framework for calculating loss, NME and accuracy value of the CNN model. All measurements represent averaged values obtained from multiple experimental iterations and are presented in *Table 9* below:

The results presented in this section were obtained by executing inference operations using the two deep neural network architectures across all hardware platforms through the system configuration previously outlined. MobileNetV2 was selected as the model architecture for this experimental evaluation, with all video inputs processed at 1920x1080 resolution. The performance evaluation framework was fundamentally established on four key assessment criteria: processing throughput, memory utilization, power differential, and thermal variation.

Table 9: Platform Experiment Results

Platform		**FPS**	**Power (W)**	**FPS/W**	**Accuracy (%)**	**Use-Case**
Software	**RTX 4060 (1.83 – 2.46 GHz)**	430	110	3.91	92.42	High-performance AI training and complex processing
	CPU i9 (3.20 – 5.50 GHz)	108	80	1.35	92.42	General development and flexible AI applications
	Jetson Nano (0.475 GHz)	9	10	0.90	92.42	Low-cost edge AI prototyping and robotics
	Raspberry (0.600 GHz)	6	7	0.79	92.42	Educational projects and battery-powered sensors
Hardware	**Kria KV260 (0.300 – 0.400 GHz)**	30	3	10	91.74	High-performance AI training and complex processing

4.3.2. Evaluation

As shown in *Table 9*, it is followed by an evaluation of trade-offs and recommendations.

Power Consumption

The desktop RTX 4060 GPU draws the most power under inference load, over 7× more than the i9 CPU, reflecting its higher thermal headroom and raw computational capability. The i9, while much lower than the GPU, still consumes tens of watts, unsuitable for battery-powered edge scenarios. The embedded boards (Jetson and Pi) operate under 10 W, making them viable for mobile/edge use. The FPGA consumes under 1.5 W, an almost order-of-magnitude reduction compared to the Pi, making it the clear winner for ultra-low-power deployments.

Landmark Accuracy

All platforms except the FPGA deliver 92.42 % accuracy (fraction of landmarks within the 0.1 threshold). The FPGA drops slightly to 91.74 %, indicating that its aggressive 8-bit quantization and fixed-point pipeline introduce a modest loss in landmark precision. For applications demanding maximal accuracy, the GPU/CPU/embedded GPU remains preferable.

FPS (Frames Per Second)

Throughput follows the inverse ordering of inference time. The RTX 4060 can process over four hundred frames per second, far exceeding typical video rates. The i9 and FPGA can handle around 100 FPS and 30 FPS, respectively, sufficient for video streams. Embedded boards cap out under 10 FPS, best suited to applications where infrequent updates suffice.

FPS/W

Based on FPS/W as shown in *Figure 36*, the FPGA KV260 SOM achieves around 10 FPS/W the highest of all platforms delivering roughly 2.5× the energy efficiency of the RTX 4060 (4 FPS/W), about 7× that of the Intel Core i9-13900K (1.4 FPS/W) and nearly 11–12× that of edge boards like the Jetson Nano 2 GB (0.9 FPS/W) or Raspberry Pi 4 B (0.8 FPS/W). Therefore, if your goal is to maximize performance per watt, especially in power- and thermally-constrained edge systems, the KV260 is the top choice. While GPUs may offer higher raw frame rates, their energy cost per frame is much greater; FPGAs with optimized bitstreams and quantization strike a better balance between throughput and power consumption.

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1. Conclusion

This research has successfully demonstrated that lightweight PFLD facial landmark detection can be effectively deployed across diverse hardware platforms, from cloud-class GPUs and desktop CPUs to embedded ARM boards and ultra-efficient FPGAs. Through systematic evaluation and optimization, we have established comprehensive performance benchmarks that guide platform selection for edge AI applications. The comparative analysis reveals significant variations in energy efficiency, computational capability, and deployment suitability across different hardware architectures, as shown in *Table 9*.

Based on our comprehensive evaluation, the FPGA Kria KV260 emerges as the optimal Edge AI platform, delivering exceptional energy efficiency of 10 FPS/W, 2.5× more efficient than GPU and 7× more efficient than CPU implementations. The system achieves real-time performance at 30 FPS while consuming only 3W, maintaining competitive accuracy (91.74%) with a compact footprint ideal for safety-critical automotive applications where driver drowsiness detection requires continuous, low-power operation. When examining the relationship between transistor count and energy performance, the KV260 demonstrates superior architectural optimization:

- **RTX 4060 (18.9 billion transistors, 5nm):** Despite having the most advanced technology and highest transistor density, it achieves only 3.91 FPS/W due to generalized GPU architecture.
- **CPU I9 (15.4 billion transistors, 10nm):** With substantial transistor count but low energy efficiency (1.35 FPS/W) due to general-purpose design.
- **KV260 (1.15 billion transistors, 16nm):** Achieves the highest energy efficiency (10 FPS/W) with significantly fewer transistors thanks to specialized FPGA architecture optimization.
- **Jetson Nano (2 billion transistors, 20nm):** Moderate performance (0.9 FPS/W) suitable for prototyping.
- **Raspberry Pi (1.5 billion transistors, 28nm):** Lowest performance (0.79 FPS/W) but suitable for educational purposes.

This demonstrates that energy efficiency depends not solely on transistor count or technology node, but more critically on architectural optimization for specific tasks. The FPGA KV260 with its reconfigurable architecture maximizes utilization of each transistor for drowsiness detection, achieving 8.7× higher performance per transistor compared to the RTX 4060.

The demonstrated FPGA implementation addresses the critical need for energy-efficient driver drowsiness detection, where fatigue-related accidents account for 3-30% of global incidents. Beyond automotive safety, this work establishes a versatile foundation for edge-vision applications including human-computer interaction, telemedicine

monitoring, AR/VR calibration, and biometric authentication systems. Moreover, using a MobileNetV2 backbone combined with a PTQ + QAT strategy yields an ultra-light CNN without sacrificing accuracy:

- MobileNetV2 leverages depthwise-separable blocks to drastically cut parameters and computation versus traditional CNNs.
- QAT tightly optimizes to 8-bit, producing a final model just 6.59 MB in size with almost no accuracy loss (92.42% → 91.74%).

In summary, the FPGA KV260 SOM paired with MobileN.etV2 (PTQ + QAT) offers an exceptional balance of size, performance, accuracy, and power efficiency, perfectly meeting the demands of lightweight, always-on, high-performance Edge AI systems. This architecture sets a precedent for edge AI deployment with optimal performance per transistor ratios, paving the way for a new generation of energy-efficient AI solutions.

5.2. Future Work

❖ Model Quality Enhancement and Advanced Optimization Techniques

Future research should focus on improving model quality through enhanced dataset diversity, incorporating challenging scenarios such as varying lighting conditions, different ethnicities, age groups, and occlusion patterns to increase robustness. Advanced quantization techniques beyond current 8-bit implementations should be explored, including 4-bit and mixed-precision quantization, adaptive bit-width allocation, and learnable quantization parameters. Investigation of knowledge distillation methods could transfer knowledge from larger teacher models to maintain accuracy while achieving extreme compression. Additionally, exploring structured pruning combined with channel-wise quantization may yield superior accuracy-efficiency trade-offs.

❖ Custom RTL Accelerator Design for FINN Independence

A critical future direction involves developing custom RTL-based CNN accelerators independent of FINN toolchain limitations. This includes designing specialized processing elements optimized for depthwise-separable convolutions, custom memory hierarchies with optimized data movement patterns, and configurable dataflow architectures supporting various CNN topologies. Hand-optimized RTL implementation would enable fine-grained control over resource utilization, pipeline depth, and memory bandwidth optimization. Custom instruction set architectures (ISA) specifically designed for facial landmark detection operations could further improve performance and reduce power consumption beyond current FINN-generated implementations.

❖ ASIC Implementation and Silicon Optimization:

Long-term research should investigate Application-Specific Integrated Circuit (ASIC) implementations for mass production scenarios where ultimate energy efficiency and cost optimization are paramount. ASIC design would enable aggressive power optimization

through voltage scaling, custom analog components for specific operations, and elimination of FPGA reconfiguration overhead. Research into specialized AI accelerator architectures such as systolic arrays, dataflow processors, and near-memory computing could achieve orders of magnitude improvement in energy efficiency. Additionally, exploring advanced semiconductor technologies (3nm, 2nm processes) and novel computing paradigms like neuromorphic chips could revolutionize edge AI deployment.

❖ **Extended Research Directions:**

- **Multi-Task Learning Architectures:** Developing unified models capable of simultaneous facial landmark detection, emotion recognition, gaze tracking, and identity verification to maximize hardware utilization and reduce overall system complexity.
- **Federated Edge Learning:** Implementing privacy-preserving collaborative learning across distributed vehicle networks, enabling continuous model improvement while maintaining data security and reducing individual system training requirements.
- **Adaptive Inference Systems:** Creating dynamic neural networks that can adjust computational complexity based on real-time performance requirements, environmental conditions, and available power budget through early exit mechanisms and conditional computation.
- **Cross-Modal Sensor Fusion:** Integrating facial landmark detection with LiDAR, radar, and physiological sensors for comprehensive driver state monitoring, requiring novel fusion architectures and multi-modal quantization techniques.
- **Reliability and Safety Assurance:** Developing fault-tolerant edge AI systems with built-in redundancy, error detection mechanisms, and graceful degradation capabilities essential for automotive safety applications meeting ISO 26262 standards.
- **Green AI and Sustainable Computing:** Investigating energy harvesting techniques, ultra-low-power sleep modes, and environmentally sustainable hardware design principles for long-term deployment in electric and autonomous vehicles.
- **Real-Time Model Compression:** Exploring online model adaptation and compression techniques that can optimize neural networks during deployment based on observed data patterns and computational constraints.

These future research directions would advance the field toward more robust, efficient, and practically deployable edge AI systems while addressing current limitations and expanding application domains beyond driver monitoring to comprehensive intelligent vehicle systems.

REFERENCES

[1]: F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” Proceedings of the MCC Workshop on Mobile Cloud Computing, 2012.

[2]: M. Satyanarayanan, “The Emergence of Edge Computing,” IEEE Computer, vol. 50, no. 1, pp. 30–39,2017.

[3]: F. Bonomi, “Cloudlets: Bringing the Cloud to the Mobile User,” Proceedings of the 3rd ACM Workshop on Mobile Cloud Computing and Services, 2012.

[4]: National Highway Traffic Safety Administration (NHTSA), “Traffic Safety Facts: Drowsy Driving,” U.S. Department of Transportation, Report No. DOT HS 812 333, 2017.

[5]: R. Krishnamoorthi, “Quantizing Deep Convolutional Networks for Efficient Inference: A White Paper,” Facebook AI Research, 2018.

[6]: W. Wu, C. Qian, S. Yang, Q. Wang, Y. Cai, and Q. Zhou, “Look at boundary: A boundary-aware”, 2018.

[7]: A. Tragoudaras, P. Stoikos, K. Fanaras, A. Tziouvaras, G. Floros, G. Dimitriou, K. Kolomvatsos, and G. Stamoulis, “Design Space Exploration of a Sparse MobileNetV2 Using High-Level Synthesis and Sparse Matrix Techniques on FPGAs,” *Sensors*, vol. 22, no. 12, p. 4318, Jun. 2022. https://doi.org/10.3390/s22124318

[8]: M. Sandle, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and lin ear bottlenecks. CoRR, abs/1801.04381, 2018.

[9]: Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; Bengio, Y. “Quantized neural networks: Training neural networks with low precision weights and activations”. J. Mach. Learn. Res. 2017, 18, 6869–6898.

[10]: Cai, Y, Yao, Z, Dong, Z, Gholami, A, Mahoney, M. W, and Keutzer, K. (2020). “Zeroq: A novel zero shot quantization framework”.

[11]: O. Nagel, M. Eichner, and P. Schiele, “Data-Free Quantization through Weight Equalization and Bias Correction,” Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops, 2019.

[12]: Y. Choukroun, M. Katz, and A. Weller, “Outlier Channel Splitting for Post-Training Quantization of Deep Neural Networks,” International Conference on Learning Representations (ICLR), 2019.

[13]: Maarten Grootendorst, “A Visual Guide to Quantization”, A Visual Guide to Quantization - by Maarten Grootendorst, 2024.

[14]: Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint arXiv:1806.08342”, 2018.

[15]: Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, Dmitry Kalenichenko, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”, 2017.

[16]: Raghuraman Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A white paper”, 2018.

[17]: Moran Shkolnik, Brian Chmiel, Ron Banner, Gil Shomron, YuryNahshan, AlexBronstein, UriWeiser, “Robust Quantization: One Model to Rule Them All”, Department of Electrical Engineering- Technion, Haifa, Israel (2019).

[18]: Yefei He, Jing Liu, Weijia Wu, Hong Zhou1, Bohan Zhuang, “EfficientDM: Efficient Quantization-Aware Fine-Tuning of Low-Bit Diffusion Models”, Zhejiang University, China ZIP Lab, Monash University, Australia, 2024.

[19]: Y. Choukroun, M. Katz, and A. Weller, “Outlier Channel Splitting for Post-Training Quantization of Deep Neural Networks,” International Conference on Learning Representations (ICLR), 2019.

[20]: AMD, Vitis AI User guide, UG1414 (v3.5) September 28, 202, https://docs.amd.com/r/en-US/ug1414-vitis-ai .

[21] Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., & Ahmed, A. (2019). “Edge computing: A survey. Future Generation Computer Systems”, 97, 219-235.

[22]: D. M. L. Barbato and O. Kinouchi, “Optimal pruning in neural networks”, Physical Review E, vol. 62, no. 6, pp. 8387–8394, Dec. 2000, Publisher: American Physical Society. DOI: 10.1103/PhysRevE.62.8387. (visited on 05/01/2024).

[23]: Vitis ai optimizer, github.io documentation, [visited:02.04.2024]. [Online].

Available: Developing a Model — Vitis™ AI 3.0 documentation

[24]: “Vitis ai optimizer • vitis ai user guide (ug1414) • reader • amd technical information portal”. [visited:22.04.2024]. (), [Online].

Available: https://docs. amd.com/r/en-US/ug1414-vitis-ai/Vitis-AI-Optimizer.

[25]: R. Gray and D. Neuhoff, “Quantization”, IEEE Transactions on Information Theory, vol. 44, no. 6, pp. 2325–2383, 1998. DOI: 10.1109/18.720541.

[26]: M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, “A white paper on neural network quantization”, arXiv preprint arXiv:2106.08295, 2021.

[27]: Satyanarayanan, M. (2017). “The emergence of edge computing. Computer”, 50(1), 30-39.

[28]: A. Jourabloo and X. Liu. “Pose-invariant 3d face alignment. In ICCV”, 2015.

[29]: Yu, W., Liang, F., He, X., Hatcher, W. G., Lu, C., Lin, J., & Yang, X. (2018). “A survey on the edge computing for the Internet of Things”. IEEE Access, 6, 6900-6919.

[30]: Porambage, P., Okwuibe, J., Liyanage, M., Ylianttila, M., & Taleb, T. (2018). “Survey on multi-access edge computing for internet of things realization. IEEE Communications Surveys & Tutorials”, 20(4), 2961-2991.

[31]: Roman, R., Lopez, J., & Mambo, M. (2018). “Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. Future Generation Computer Systems”, 78, 680-698.

[32]: Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., & Ahmed, A. (2019). “Edge computing: A survey. Future Generation Computer Systems”, 97, 219-235.

[33]: Mao, Y., You, C., Zhang, J., Huang, K., & Letaief, K. B. (2017). “A survey on mobile edge computing: The communication perspective”. IEEE Communications Surveys & Tutorials, 19(4), 2322-2358.

[34]: Kuon, I., Tessier, R., & Rose, J. (2008). “FPGA architecture: Survey and challenges. Foundations and Trends® in Electronic Design Automation”, 2(2), 135-253.

[35]: Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R, Ong Gee Hock, J, ... & Deborah, M. (2017). “Can FPGAs beat GPUs in accelerating next-generation deep learning?” In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 5-14).

[36]: Betz, V., Rose, J., & Marquardt, A. (1999). “Architecture and CAD for deep-submicron FPGAs. Springer Science & Business Media”.

[37] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). “Optimizing FPGA-based accelerator design for deep convolutional neural networks”. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 161-170).

[38]: Lemieux, G. G., & Lewis, D. (2004). “Design of interconnection networks for programmable logic. Springer Science & Business Media”.

[39]: Farabet, C., Martini, B., Akselrod, P., Talay, S., LeCun, Y., & Culurciello, E. (2010). “Hardware accelerated convolutional neural networks for synthetic vision systems. In Proceedings of 2010 IEEE International Symposium on Circuits and Systems (pp. 257-260)”.

[40]: U. Farooq et al, “Tree-Based Heterogeneous FPGA Architectures , FPGA Architectures: An Overview”, 2012

[41]: Ketkar, N., & Moolayil, J. “Deep learning with Python: Learn best practices of deep learning models with PyTorch. Apress”, 2021.

[42]: Guo, X., Li, S., Yu, J., Zhang, J., Ma, J., Ma, L., Liu, W., & Ling, H. (2019). “PFLD: A Practical Facial Landmark Detector [Preprint]. arXiv:1902.10859 v2”.

[43]: Xiang Wang, Kai Wang, Shiguo Lian, “A Survey on Face Data Augmentation”, CloudMinds Technologies Inc, 2019.

[44]: A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. “Mobilenets: Efficient convolutional neural net works for mobile vision applications. CoRR, abs/1704.04861”, 2017.

[45]: W. Wu, C. Qian, S. Yang, Q. Wang, Y. Cai, and Q. Zhou, “Look at boundary: A boundary-aware face alignment algorithm. In CVPR”, 2018.

[46]: A. Kumar and R. Chellappa, “Disentangling 3d pose in a dendritic cnn for unconstrained 2d face alignment. In CVPR”, 2018

[47]: A. Jourabloo, X. Liu, M. Ye, and L. Ren, “Pose invariant face alignment with a single cnn. In ICCV”, 2017.

[48]: H. Yang, W. Mou, Y. Zhang, I. Patras, H. Gunes, and P. Robinson, “Face alignment assisted by head pose estimation. In BMVC”, 2015.