OPTIMIZE AND DEPLOY MACHINE LEARNING ALGORITHMS ON EMBEDDED DEVICES FOR MANUFACTURING APPLICATIONS

By

TEOH MING XUE

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2025

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisors, Dr. Lee Wai Kong who has given me this bright opportunity to engage in Machine Learning and Embedded System project. It is my first step to establish a career in Machine Learning and Embedded System Field. A million thanks to you.

To a very special person in my life, Ang Qiao Yi, for her patience, unconditional support, and love, and for standing by my side during hard times. Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

This proposal discusses the techniques of optimizing and deploying machine learning algorithms on embedded devices for manufacturing applications; We investigate problems of printed circuit board (PCB) defects and artificial intelligence in embedded system. PCB defects detection had been an essential problem to solve in manufacturing environments, whether it is quality assurance or the needs for PCB inspection had been ramping since decades ago. Fundamental limitation of human-based judgement of inspection engineers is the primary cause of faulty products including PCB defects exiting the manufacturing environment. On the other hand, artificial intelligence had been ways to enhances embedded system by enabling real-time, accurate detection and management of PCB defects through advanced pattern recognition and automated inspection methods. However, embedded system often been having limited computing power, small memory storage and relies on battery capacity. Not to say the difficulty in deploying either artificial intelligence or deep learning in embedded environments due to significant parameters size and computational complexity. In recent studies, we seen developers and researchers proposing solutions on deep learning algorithms like YOLO, EfficientNet, CNN, MobileNet etc. On the other hand, network compression and acceleration techniques such as pruning and quantization also been the focus of the studies for light-weight algorithms in embedded system. While in our studies, we primarily focusing on the deployment and fine-tuning of deep learning model which is YOLOv5 for PCB defects detection. We aim to levitate the baseline YOLOv5 into a state-of-the-art version that focus on lightweight performance, called the LW-YOLOv5, which can been deploy seamlessly into embedded systems for manufacturing applications. As we conduction evaluation experiment on our model using openly accessible datasets like PKU-Market-PCB and perform comparative studies with the latest proposed solutions.

Area of Study: Embedded System, Artificial Intelligence

Keywords: Printed Circuit Boards, Machine Learning, YOLOv5, Microcontroller, Lightweight

TABLE OF CONTENTS

TITLE PAGE **i**

ACKNOWLEDGEMENTS **ii**

COPYRIGHT STATEMENT **iii**

ABSTRACT **iv**

TABLE OF CONTENTS **iii**

LIST OF FIGURES **viii**

LIST OF TABLES **xi**

LIST OF ABBREVIATIONS **xii**

CHAPTER 1 INTRODUCTION **1**

1.1 Problem Statement and Motivation 6

1.2 Research Objectives 8

1.3 Project Scope and Directions 9

1.4 Contributions 11

1.5 Report Organization 13

CHAPTER 2 LITERATURE REVIEW 15

2.1 Previous Works on PCB Defect Detection 15

2.1.1 Lightweight PCB defect detection algorithm based on MSD - YOLO 15
2.1.2 Optimized Lightweight PCB Real-Time Defect Detection Algorithms 18
2.1.3 YOLO-LFPD: A Lightweight Method for Strip Surface Defect Detection 20
2.1.4 Light-YOLOv5: A Lightweight Algorithm for Improved YOLOv5 in PCB Defect Detection 22
2.2 Previous Works on Object Detections
2.2.1 Flexible and Fully Quantized Lightweight TinyissimoYOLO 25
for Ultra-Low-Power Edge Systems 25
2.2.2 Improved Light-Weight Target Detection Method Based on YOLOv5 28
2.2.3 MPQ-YOLO: Ultra low mixed-precision quantization of YOLO for edge devices deployment 30

2.3 Comparison of Models from Previous Works 33
2.3.1 PCB Defect Detection Datasets from Peking University 33
2.3.2 Object Detection Datasets from PASCAL 33
2.4 Comparison between The Techniques Used from Previous Works 34
2.4.1 PCB Defect Detection Datasets from Peking University 34
2.4.2 Object Detection Datasets from PASCAL 35

CHAPTER 3 SYSTEM METHODOLOGY / APPROACH 37
3.1 System Design Diagram / Equation 37
3.1.1 Introduction to classes in PCB Defect Detection 37
3.1.2 Introduction to Augmentation Type Applied 41
3.1.3 System Architecture Diagram 42
3.2 Gantt Chart 58

CHAPTER 4 SYSTEM DESIGN 59
4.1 System Block Diagram 59
4.2 System Components Specifications 60

CHAPTER 5 SYSTEM IMPLEMENTATION 66

5.1 Hardware Setup 66

5.2 Software Setup 68

5.3 Setting and Configuration 68

5.3.1 Setup for Libraries in Google Colab 68

5.3.2 Setup for Libraries in Ubuntu 22.04 LTS 84

5.4 System Operation 86

5.5 Implementation Issues and Challenges 92

5.6 Concluding Remark 92

CHAPTER 6 SYSTEM IMPLEMENTATION 94

6.1 System Testing and Performance Metrics 94

6.1.1 System Testing 94

6.1.2 Performance Metrics 94

6.2 Testing Setup and Result 96

6.2.1 Testing Setup and Results for LW-YOLOv5 96

6.2.2 Testing Setup and Results for Nvidia Jetson Orin Nano 107

6.3 Comparative Analysis and Ablation Experiments against SOTA Models 109

6.4 Error Analysis 115

6.5 Project Challenges 118

6.6 Objectives Evaluation 119

6.7 Concluding Remark 121

CHAPTER 7 CONCLUSION AND RECOMMENDATION 123

7.1 Conclusion 123

7.2 Recommendation 124

REFERENCES 126

APPENDIX A

A.1 Poster A1

LIST OF FIGURES

Figure Number	Title	Page
Figure 1.1.1	Venn Diagram for Recent Studies Reviews	2
Figure 2.1.1.1	Improved Network Structure based on baseline YOLOv5.	15
Figure 2.1.1.2	MobileNet_Block structure	16
Figure 2.1.1.3	Depth-separable convolution illustration	17
Figure 2.1.1.4	Ablation experiments on tabular forms for model comparison	18
Figure 2.1.2.1	Structural Layers of MobileNetV3.	19
Figure 2.1.2.2	Channel Pruning Principle.	20
Figure 2.1.2.3	Tabular comparison with different architecture of models.	20
Figure 2.1.3.1	The architecture structure of the model YOLO-LFPD.	21
Figure 2.1.3.2	Evaluation Results on Various Models.	22
Figure 2.1.4.1	Lightweight YOLOv5 model architecture	23
Figure 2.1.4.2	Construction of the Gsbottleneck component.	23
Figure 2.1.4.3	The partial network module - VoV-GSCSP built upon fundamental of GSConv	24
Figure 2.1.4.4	Performance comparison between different variants of YOLOv5	25
Figure 2.2.1.1	Architecture Structure of TinyissimoYOLO	26
Figure 2.2.1.2	Tabular comparison of parameters evaluated on PASCAL VOC.	26
Figure 2.2.1.3	Tabular views of hyperparameter training of QAT and full-precision training.	27
Figure 2.2.2.1	Comparison of SKConv (Upper) and SKConv-G (Lower) model network design.	28
Figure 2.2.2.2	Comparison results between usage of ECA and C-ECA.	29

Figure 2.2.2.3	Redesigned network structure based off baseline YOLOv5.	30
Figure 2.2.2.4	Comparison between various models on PASCAL VOC 2007 test.	30
Figure 2.2.3.1	The framework of MPQ-YOLO.	31
Figure 2.2.3.2	Three-step quantization process.	32
Figure 2.2.3.3	Results return on the effectiveness of usage of dedicated training techniques.	32
Figure 2.2.3.4	Evaluation Results on various target detection models	33
Figure 3.1.1.1	Missing hole defects in PCB defect detection [15].	37
Figure 3.1.1.2	Mouse bite defects on PCB defect detection [15].	38
Figure 3.1.1.3	Open Circuits defects on PCB defect detection [15].	38
Figure 3.1.1.4	Short defects on PCB defect detection [15].	38
Figure 3.1.1.5	Spur defect on PCB defect detection [15].	39
Figure 3.1.1.6	Spurious copper defects on PCB defect detection [15].	39
Figure 3.1.3.1	LW-YOLOv5 architecture design in details	43
Figure 3.1.3.2	Comparison of calculated anchor box visualized	44
Figure 3.1.3.3	Detail Operation for SPD-Conv [43]	45
Figure 3.1.3.4	Dynamic Convolution Operation Diagram [47]	47
Figure 3.1.3.5	The Structure of RCS [50].	48
Figure 3.1.3.6	The Principle of MLCA algorithm [49]	50
Figure 3.1.3.7	Overview of RT-DETR Encoder (with AIFI + CCFF modules) [48]	52
Figure 3.1.3.8	SPDConv’s internal operations	53
Figure 3.1.3.9	RCSOSA Block’s internal operations	54
Figure 3.1.3.10	C3-GhostDynamicConv’s internal operations	55
Figure 3.1.3.11	MLCA [49]’s internal operations	55
Figure 3.2.1	Gantt Chart	58
Figure 4.1.1	General Block Diagram of Proposed System	59
Figure 4.2.1	Block Diagram with break down workflow	60
Figure 4.2.2	Differences between IoU-based detector (first row) and NWD-based detector (second row). Green, blue and	64

	red indicating true positive (TP), false positive (FP) and false negative (FN) respectively [44]	
Figure 5.1.1	Nvidia Jetson Orin Nano on Setup	66
Figure 5.3.1.1	Installation for required libraries	68
Figure 5.3.1.2	Standard Libraries	69
Figure 5.3.1.3	External Libraries	69
Figure 5.3.1.4	Other Libraries	70
Figure 5.3.1.5	Downloading Datasets from Kaggle	70
Figure 5.3.1.6	Preparing image and annotation paths for datasets pre-processing	71
Figure 5.3.1.7	Output of preparation of data	71
Figure 5.3.1.8	Function for parsing XML to YOLO format	72
Figure 5.3.1.9	Function for visualizing annotations on image.	73
Figure 5.3.1.10	Function to visualizing class distribution in the datasets.	74
Figure 5.3.1.11	Class Distribution in Train Data.	74
Figure 5.3.1.12	Class Distribution in Validation Data	75
Figure 5.3.1.13	Class Distribution in Testing Data	75
Figure 5.3.1.14	Data Splitting into 7:2:1 Ratio	76
Figure 5.3.1.15	Defining various augmentation techniques	77
Figure 5.3.1.16	Function to apply data augmentations	77
Figure 5.3.1.17	Function to perform augmentations	78
Figure 5.3.1.18	Output of class distribution in training data after augmentation	79
Figure 5.3.1.19	Function to count total number of images augmented	79
Figure 5.3.1.20	Data YAML file preparation	80
Figure 5.3.1.21	K-Means Clustering Function	81
Figure 5.3.1.22	Modified Model YAML file	82
Figure 5.3.1.23	Model Evaluation using testing data	84
Figure 5.3.2.1	Checking of missing dependencies on Ubuntu 22.04	85
Figure 5.3.2.2	Cloning the LW-YOLO source tree	85
Figure 5.3.2.3	Inspecting repository contents	85
Figure 5.3.2.4	Installing project-specific dependencies	86

Figure 5.4.1	Full-precision evaluation with val.py	86
Figure 5.4.2	Full-precision batch detection with detect.py	87
Figure 5.4.3	Visual Output of full-precision detection 1	87
Figure 5.4.4	Visual Output of full-precision detection 2	88
Figure 5.4.5	Half-precision evaluation (--half)	88
Figure 5.4.6	Half-precision batch detection	88
Figure 5.4.7	Visual Output of half-precision detection	89
Figure 5.4.8	ONNX Runtime evaluation (best.onnx)	89
Figure 5.4.9	ONNX Runtime batch detection	90
Figure 5.4.10	Visual Output of Detection for best.onnx	90
Figure 5.4.11	ONNX-Runtime evaluation of an FP16-export (bestFP16.onnx)	91
Figure 5.4.12	Visual output of FP16-ONNX detection	91
Figure 6.2.1.1	Model Evaluation using testing data	96
Figure 6.2.1.2	Training Loss Curve for LW-YOLOv5	100
Figure 6.2.1.3	Validation Loss Curve for LW-YOLOv5	100
Figure 6.2.1.4	Precision-Recall Curve for LW-YOLOv5	102
Figure 6.2.1.5	Precision-Confidence Curve for LW-YOLOv5	103
Figure 6.2.1.6	F1-Confidence Curve for LW-YOLOv5	104
Figure 6.2.1.7	Recall-Confidence Curve for LW-YOLOv5	105
Figure 6.2.1.8	Confusion Matrix for LW-YOLOv5	106
Figure 6.3.1	Comparative Studies Between State-of-the-art YOLOv5n and LW-YOLOv5, (*) indicates augmentation techniques had been applied.	111
Figure 6.3.2	Ablation Experiments of LW-YOLOv5 (1)	113
Figure 6.3.3	Ablation Experiments of LW-YOLOv5 (2)	113
Figure 6.4.1	Confusion matrix for error analysis	117
Figure 6.4.2	Per-class Precision and Recall for error analysis	117

LIST OF TABLES

Table Number	**Title**	**Page**
Table 2.3.1.1	Comparison between model’s performance and parameters from papers reviewed for PCB defects detection	33
Table 2.3.2.1	Comparison between model’s performance and parameters from papers reviewed for Object Detection	33
Figure 2.4.1.1	Comparison of Techniques Used for PCB defect detection datasets	34
Figure 2.4.1.2	Comparison of Techniques Used for PASCAL VOC datasets	35
Table 3.1.1.1	Tabular Views of Properties of PCB defects	39
Table 3.1.2.1	Data-augmentation recipe for PCB-defect training set	42
Table 3.1.2.2	Expansion of the training corpus after augmentation	42
Table 4.2.1	Overviews of Hyperparameter Settings	61
Table 5.1.1	Specifications of laptop [45]	66
Table 5.1.2	Specifications of the microcontroller [51]	67
Table 5.2.1	Specifications of software used	68
Table 6.1.2.1	Detection-quality metrics	94
Table 6.1.2.2	Computational-cost metrics	94
Table 6.2.1.1	Tabular Results for Training Data (Without Augmentation)	96
Table 6.2.1.2	Tabular Results for Testing Data for LW-YOLOv5	97
Table 6.2.1.3	Tabular Results for Training Data (With Augmentation)	97
Table 6.2.1.4	Tabular Results for Testing Data (With Augmentation)	98
Table 6.2.1.5	Tabular Results for comparison of both pretrained model	98
Table 6.2.1.6	Tabular Results for Training Data for LW-YOLOv5	99
Table 6.2.1.7	Tabular Results for Testing Data for LW-YOLOv5	99

Table 6.2.2.1	command to record microcontroller stats during inferences	107
Table 6.2.2.2	Various weights format for different model during inference time	107
Table 6.2.2.3	Resource Utilization for Optimized LW-YOLOv5 inference	107
Table 6.2.2.4	Performance Comparison of LW-YOLOv5	108
Table 6.3.1	Comparative studies between state-of-the-art models	109
Table 6.3.2	Ablation Experiments for LW-YOLOv5	110
Table 6.4.1	Error Analysis Visualization	115

LIST OF ABBREVIATIONS

PCB	Printed Circuit Board
VOC	Visual Object Classes
SPP	Spatial Pyramid Pooling
SSD	Single Shot Detector
IC	Integrated Circuits
ReLU	Rectified Linear Unit
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
SE	Squeeze-and-Excitations
Focal-EloU	Focal Enhanced Intersection over Union
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MiB	Mebibyte
COCO	Common Object in Context
IOU	Intersection over Union
CBAM	Convolutional Attention Module
FPN	Feature Pyramid Network
PANet	Path Aggregation Network
BiFPN	Bidirectional Feature Pyramid Network
BNL	Batch Normalization Layer
SPPF	Spatial Pyramid Pooling - Fast
CSP	Cross Stage Partial Layer
GSConv	Grouped Spatial Convolution
SC	Standard Convolution
VoV-GSCSP	VoVNet with Grouped Spatial Convolution and Cross Stage Partial
FLOPs	Floating Point Operations
GAP	Global average pooling
GMAC	Giga Multiply-Accumulate Operations
TAT	Trained Quantization Thresholds.
ECA	Efficient Channel Attention

SPD-Conv	Space-to-Depth Convolution
MLCA	Mixed Local Channel Attention
NWD	Normalized Wasserstein Distance

Chapter 1

Introduction

Printed Circuit Boards (PCBs) are the basic fundamental of almost all the functioning electronics devices around us, the primary purpose serving as the base that support essential components like chips, capacitors and transistors. PCBs must be produced with the utmost standard of quality with minimal defects to make sure they work all the times. Traditional methods of defect detection during manufacturing like can cause inefficiency and increase error margins as they are not able to keep up with the emerging demands of modern production environment [1].

When talk about their application, PCBs had played a crucial role in measurement, detection, identification and positioning, particularly when using intelligent vision technology. PCBs defect detection generally falls under the type of object detection which can conclude in a two-step procedure. The first being identification tasks of potential spot within video or image, while the second being categorizing detected objects using the cropped image to further confirms their significant components within a board [18].

To ensure perfect PCBs manufacturing, manufacturers need to identify the important of preserving its reliability and safety for the cause. There are many ways for PCB inspections, whereas the most common ones come in as Automatic Optical Inspection (AOI) and manual checks by inspection engineers that had extension training on this topic. These traditional methods are both error-prone and resource intensive. As the goal is to deliver defects-free products to the baseline of the consumers whenever it is needed, these methods are definitely not solving the problem. Furthermore, recent studies that had shown that traditional machine-based classifications and feature extraction algorithms are not enough to outdoes deep learning methods that had faster and more precise than these traditional methods. [3]. Not to mention, impact of various degradation factors, such as defect detection methods, data quality and fault image need to be taken consideration when it comes to training of deep learning model for PCB defect detection [11].

Therefore, topics on various deep learning approaches that had been used to tackle similar area of problems shall be discussed, whether it is PCBs defect detection, railway defects detection or object detection capabilities. Here, this study presents the visualization of reviews on PCB defects detection and application of deep learning on embedded environments in figure 1.1.1.

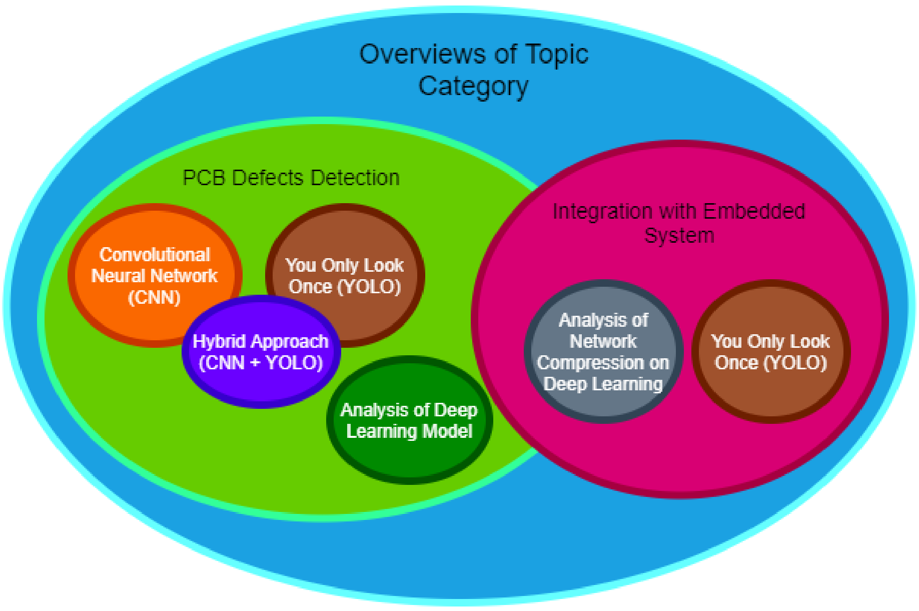

Overviews of Topic Category
PCB Defects Detection
Convolutional Neural Network (CNN)
You Only Look Once (YOLO)
Hybrid Approach (CNN + YOLO)
Analysis of Deep Learning Model
Integration with Embedded System
Analysis of Network Compression on Deep Learning
You Only Look Once (YOLO)

Figure 1.1.1 Venn Diagram of Recent Studies Reviews.

Recent studies had shown multiple research that had done to prove that deep learning methods like Convolutional Neural Network (CNNs), You-Only-Look-Once (YOLOs) and YOLO-CNN hybrid approaches had been a feasible way to tackle this problem all along. Firstly, the usage of CNNs under the problem of PCBs defects detection, both papers use structure of CNNs as their primary building blocks. In [6], this paper proposes a Skip-Connected Convolutional Autoencoder for PCB defects with the autoencoder being designed to learn flattened representation for non-defective images, further perform evaluation to the actual defective images for defects identifications. Whereas to tackle the degradation problem, it utilizes skip connections between the encoder and decoder layers which gained more detailed features during decoding process. It able to achieve a detection rate (DR) of 98% and a false pass rate below 1.7%. Other than that, [14] shown a good comparative study between two CNN-based models for PCB defect detection which is SSD ResNet50v1 and EfficientDet D1. The paper shows how them utilize model’s backbone to identifying and classifying

PCB defects. From the result, the studies shown that SSD ResNet50v1 model achieved an accuracy of 81.68% while it utilizes backbone of ResNet50, coupled with Single Shot Detector (SSD) head, allowing a better real-time object detection than EfficientDet D1.

Other studies then further explore the possibility of deep learning model like YOLO architecture for PCB defects detection. YOLO is also the main focus as it is a popular CNN-based object detection model known for real-time processing and accuracy capabilities. In [1], the paper focuses on an enhanced version of YOLOv3 which is made for PCB defects detection. It utilizes integration of dual attention mechanism for better capturing features related to PCB defects and refinement of the anchor box selection for better localization accuracy. As its results compare to the baseline YOLOv3, which able to achieve higher mean Average Precision (mAP) and faster processing speed. Besides in [12], YOLOv4 architecture had also been implemented on similar topics on detecting integrated circuits (ICs) on PCB which conclude a critical task for PCBs defect detection. The proposed model utilizes CSPDarknet-53 as the backbone with the EfficientNetv2-L architecture in baseline YOLOv4. The model is designed to detect defects in PCBs in identifying variations and inconsistencies in placement and condition of ICs. It had done great results on F1-score and prediction speed, making it ideal for real-time inspection in manufacturing environment.

Various studies also been circulating around the model of YOLOv5 due to its efficiency, accuracy and speed that outperforms the previous YOLO models. In [4], this paper proposed YOLOv5 model for PCB defect detection, splitting it into three respective model variants, which is small, medium and large. While all the results surpass the accuracy percentage of 97.52% using the optimized YOLOv5 model variants. A method called the YOLO-MHC introduced in [10] also utilize YOLOv5 but it introduces a hybrid multi-channel feature extraction approach with the combination of spatial pyramid pooling (SPP) and attention mechanisms. This particular study emphasizes on detection minor and complex PCB defects compared with the baseline YOLO models, achieving mean Average Precision (mAP) of 99.37%. Rather than that, one studies had been done on YOLOv5 architecture also, which is the YOLO-pdd. We see YOLO-pdd modified the YOLOv5 architecture with the Res2Net by integration of multi-scale feature fusion techniques to improve detection precision across numerous defect sizes and types on PCBs. This model also employs a novel loss function designed to balance precision and recall, focusing on small defects and low-contrast regions [13]. Now, another hybrid approach of using YOLO model but with an advanced CNN architecture is used in [3], this paper employs Tiny-YOLOv2 model, design to perform fast detection with reduced number of convolutional layers. The architecture of Tiny-YOLO-v2 model includes convolutional layers, activation functions like Leaky ReLU and pooling layers. This particular lightweight architecture allows for fast inference speeds and small model size of less than 50 MB make it suitable for implementation on embedded systems.

After reviewing recent studies about how deep learning algorithms tackle the problem of PCB defects detection, now we switch over to the topic of application of deep learning models in embedded system environments. As deep learning becomes closer into our daily life via technology's devices, there is a growing need to make these models more efficient for mobile devices and embedded system. These devices like smartphones, tablets, microcontroller and Iot gadgets, often having limited amount of processing power, memory and battery lifetime. Slow performance and high energy consumption are often the backbone of the problem for running large neural networks on such platforms. Researchers had been finding ways to compress these models, prioritizing limiting their size and making them runs more efficiently on those resource-limited devices.

As for the topics discussed, we truly interested of finding out techniques like pruning and quantization for deep learning models. Essentially, key purpose for pruning is that it can compress neural network architecture down to a good size by pruning out unessential and inefficient parameters [5]. Meanwhile, the size of the model can be reduced without losing much in terms of performance wise. On the other side, Quantization, simplifies the model even further by reducing the precision of the data it uses, for instance converting 32-bit numbers to 8-bit numbers. This approach generally helps in saving memory consumption and quicker processing speed [2], [7].

To have better understanding on the topics, numerous studies had been showing their efforts on tackling the problem of pruning and quantization on deep learning algorithms. In [2], pruning can be categorized as two mode, static and dynamic, performing at offline mode will be consider as static and run-time will be considered as the other way around. Besides that, this paper discussed different granularities of pruning, such as elementwise, filter-wise and layer-wise pruning, each targeting different levels of network structure. Besides, the paper also categorizes quantization techniques as, post-training quantization (PTQ) and quantization-aware training (QAT) which both had their advantages and disadvantages from the tradeoff of computational resources and accuracy wise.

While on the other hand, [5] proposed that pruning can be classified as non-structured and structured pruning. Non-structured pruning eliminates individual weights, indicating sparse networks that can be difficult to optimize on hardware due to irregular memory access patterns. In contrast, structured pruning deletes entire neurons, filters, or channels, which results in more consistent, hardware-optimized network architectures. As for quantization techniques, the paper discussed two types of quantization techniques which involve, linear quantization and non-linear quantization. Both type of techniques, for examples, like non-linear quantization, uses techniques like k-means clustering to reduce the number of unique weight values.

As for quantization techniques used to optimize deep neural networks (DNNs) for deployment on low-power microcontrollers. In [7], the paper utilizes a residual neural network (ResNet) architecture for deployment. The approaches used include both post-training quantization (PAT) and quantization-aware training (QAT) that we previously discussed in [2]. The authors also developed a framework called the MicronAI to help with the complete process of training, quantization and utilization of DNNs on microcontrollers. The relevant results shown that the models had been quantized to 16-bit fixed-point precision generally maintained similar accuracy levels while the 8-bit precision quantized model experienced a slight drop in accuracy.

Rather than development of framework on relevant topics, several models targeting different problem such as object detection and railway track damage detection had also been discussed on recent research while adapting techniques for model compression. In [8], the paper had used the YOLOv5 model by applying pruning at the Batch Normalization (BN) layer to reduce the number of parameters. It was then followed by quantization process using the OpenVINO toolkit to transform model's weight from 32-bit floating point to 80bit integer format. The improved YOLOv5 is able to achieve a 36.6 reduction in parameter count and a 35% reduction in weight storage file size compared to the baseline.

In [9], the paper introduces YOLOv5-LITE as a lighter weight version for the baseline YOLOv5 version for lower-power devices. Here, the paper utilizes different techniques and method to reduce model's operational efficiency and memory footprint by using Fused Mobile Inverted Bottleneck Convolution (BF_MBConv) to minimize parameters and floating-point. Also, the paper uses similar mechanism like which is Squeeze-and-Excitation (SE) functions to enhance feature importance. Additionally, DropBlock, Shuffle convolution and Focal-EloU Loss function had been used to create the ultimate YOLOv5-LITE model which achieved over a mean Average Precision (mAP)@0.5 of 94.4%, an 8% more than the original YOLOv5 model.

1.1 Problem Statement and Motivation

One of the most important components from the fundamental aspect of almost all electronic devices, spanning industries like computing, telecommunications, aerospace and industrial system is printed circuit boards (PCBs) [1]. Since the advance of technology began, the needs for smaller and lighter electronic devices have never been appearing like never before. However, the more compact the PCBs design is going to be, the more challenging it is going to be for implement, develop or construct to reach such requirements. Defects that lie within PCBs can be missing holes, mouse bites, open circuits, shorts, spurs and spurious copper. On a rare occasion, incident like board overheating or catastrophic failures can occur if those defects were to leave undetected [11]. Tracing back to the traditional PCB defect detection methods can be Automatic Optical Inspection (AOI), Manual inspection and a more advanced techniques of electrical testing. These methods pose different difficulty and limitations in term of manpower and operational costs like expensive tools and complex circuits for electrical testing [20]. AOI systems, which primarily depend on rule-based algorithms, often struggle to handle issues like corrupted PCB images, varying component layouts, and irregular component sizes [12][13]. These limitations highlight the need for more flexible and reliable methods for detecting defects.

With those advancements in artificial intelligence, deep learning has emerged as a promising alternative for PCB defect detection. Deep learning models were often the best choice for handling complex image classification and object detection tasks, as their learning speed and solid deductive computation [6]. However, deploying these large, computationally intensive models on embedded devices introduces typical challenges that is similar to mobile phone. Embedded systems are usually equipped with limited memory (often less than 1 MiB), constrained processing power and low energy consumption. They often struggle to run the deep learning algorithms that is resource-hungry which are commonly used for defect detection [7]. Most deep learning models are designed to run on powerful systems with high-performance GPUs and CPUs. Besides, not to mention their huge memory consumption and further complicating deployment on embedded platforms [2]. As such, these hardware constraints limit the potential of traditional deep learning models in embedded environments, prompting the need for better novelty and advancements to be made.

The motivation for addressing PCB defect detection stems from the growing reliance on PCBs in devices like smartphones, drones, and wearables, where ensuring quality, safety, and functionality is paramount. Traditional inspection methods are increasingly having insufficient capability for smaller and more intricate PCB designs. In that case, there are more demand for an efficient detection technique [1][11]. Defects in PCBs not only lead to costly recalls but also damage a company's reputation in manufacturing them. This has fueled the requirement for innovative solutions, particularly those leveraging artificial intelligence and embedded systems. These requirements allow more works to emerged and competed against to enhance algorithm's detection accuracy and efficiency. On the other hand, with their ability to learn from large datasets and adapt to new defect types, deep learning models can offer a compelling alternative to rule-based inspection methods [2][6].

While deep learning holds promising outcomes and solutions, the challenge lies in tailoring these models for deployment on embedded systems. Embedded environments impose strict constraints on computational resources. Therefore, the industry needs to come out with requiring novel approaches to reduce model size and computational demands without sacrificing accuracy [7]. As to address the problem stated, pruning and quantization, as highlighted by [2], are among the techniques used to compress deep neural networks for faster inference on resource-constraint devices. The development of optimized lightweight deep learning models could pave the way for real-time defect detection and be transforming the PCB inspection landscape.

In conclusion, the continued evolution of PCB designs and manufacturing demands calls for an intersection of artificial intelligence and embedded systems to address limitations in traditional inspection methods. By leveraging the capability of deep learning and addressing the hardware limitations of embedded devices. It is definitely achievable to create robust and efficient solutions for PCB defect detection. Such advancements promise to enhance quality assurance, reduce production costs and meet the growing demand for high-performing electronic devices across industries. This research thus seeks to bridge the gap between cutting-edge AI algorithms and practical, real-world applications in PCB manufacturing.

1.2 Research Objectives

This research aims to enhance the field of PCB defect detection by optimizing and deploying deep learning algorithms on embedded devices for manufacturing applications. The specific objectives of this work are as follows:

1. **Development of a LW-YOLOv5 for Embedded Systems**:

We introduce a novel, lightweight version of YOLOv5 model, called the LW-YOLOv5 model, optimized for real-time PCB defect detection on embedded environment with the memory consumption of less than 100MB. We emphasize on using baseline YOLOv5 from the YOLOs, specifically the YOLOv5-nano version, then slowly converging to a more improved version specifically focusing on lighter model's weight and higher accuracy and precisions for detections. We aimed for better optimized performance in term of mean Average Precision (mAP), params size (M) and model size (MB).

2. Comparative Analysis and Ablation Studies of LW-YOLOv5 Against State-of-the-art Models

Conduct comparative analysis and ablation studies by Q4 2026 to evaluate LW-YOLOv5 against state-of-the-art models (e.g., MSD-YOLO [18], YOLO-LFPD [20], ARMA-based YOLO [46]) for PCB defect detection. Implement and test at least four optimization techniques (e.g., optimized anchor boxes, Normalized Wasserstein Distance [44], Space-to-Depth Convolution [43], Mixed Local Channel Attention [49]) to achieve a mAP@0.5 of 0.90 or higher and reduce computational complexity by 20% compared to YOLOv5n, ensuring suitability for embedded systems.

3. Deployment of LW-YOLOv5 on NVIDIA Jetson Orin Nano

Deploy the LW-YOLOv5 model on the NVIDIA Jetson Orin Nano [51], achieving an inference speed of at least 30 FPS at 640x640 resolution using FP16 ONNX format, with average power consumption below 7W and memory usage under 4GB. Ensure thermal stability (CPU/GPU temperatures below 60°C) and evaluate performance for offline quality audits, targeting a mAP@0.5 above 0.90 for real-time PCB defect detection.

4. Comprehensive Analysis on LW-YOLOv5 on Resource-Constrained Platforms

Evaluate LW-YOLOv5's performance on resource-constrained platforms, including the NVIDIA Jetson Orin Nano [51]. Achieve a mAP@0.5 above 0.90, inference speed of at least 30 FPS, and memory usage below 100MB using TensorFlow Lite Micro. Improve detection of small defects (e.g., mouse bites) and resolve class ambiguities (e.g., spur vs. spurious copper) through targeted data augmentation, aiming for a 10% reduction in false positives and negatives.

1.3 Project Scope and Direction

This study focuses on optimizing and deploying machine learning algorithms on embedded devices, specifically the NVIDIA Jetson Orin Nano [51], for real-time printed circuit board (PCB) defect detection in manufacturing environments. The project addresses the critical challenge of ensuring high-quality PCB production by developing a novel, lightweight deep learning model, LW-YOLOv5, derived from the YOLOv5-nano architecture. The primary deliverable is a compact model with a size under 1MB, optimized for high accuracy (mAP@0.5 $\geq$ 0.90) and real-time inference ($\geq$ 30 FPS) on the Jetson Orin Nano, targeting defects such as missing holes, shorts, open circuits, spurs, spurious copper, and mouse bites.

The scope encompasses modifying the YOLOv5-nano architecture by integrating lightweight components (e.g., Space-to-Depth Convolution (SPD-Conv) [43] and Mixed Local Channel Attention (MLCA) [49]). The model will be trained and evaluated using the PKU-Market-PCB, with data augmentation techniques (e.g., horizontal/vertical flips, motion blur, brightness adjustments) to enhance robustness. Limitations include the Jetson Orin Nano's constrained computational power, memory (<1GB target), and power consumption (<7W target), which necessitate a balance between model size, accuracy, and inference speed. Traditional methods like manual inspection and automated optical inspection (AOI) suffer from lower accuracy and adaptability, justifying the need for a lightweight deep learning solution.

The workflow involves improving the YOLOv5 backbone and neck, optimizing anchor boxes via K-means clustering, and conducting comparative studies against state-of-the-art models (e.g., MSD-YOLO [18], YOLO-LFPD [20], ARMA-based YOLO [46]). Ablation experiments will quantify the impact of each optimization on model size, detection accuracy, and computational cost. The project aims to deploy LW-YOLOv5 on the Jetson Orin Nano, achieving thermal stability (CPU/GPU temperatures <60°C) and enabling applications in offline quality audits or low-volume inspection tasks, with potential for future inline automated optical inspection.

1.4 Contributions

This research significantly enhances PCB defect detection by developing a novel, lightweight deep learning model, LW-YOLOv5, optimized for real-time performance on embedded systems, specifically the NVIDIA Jetson Orin Nano. The first major contribution is the creation of LW-YOLOv5, derived from the YOLOv5-nano baseline, designed to achieve a model size under 1MB and a parameter count of approximately 1.18M while delivering high accuracy for defects like missing holes, shorts, open circuits, spurs, spurious copper, and mouse bites. By integrating lightweight architectural optimizations such as Space-to-Depth Convolution (SPD-Conv) [43], Normalized Wasserstein Distance (NWD) [44], and Mixed Local Channel Attention (MLCA) [49], the model targets a mean Average Precision (mAP@0.5) of 0.90 or higher on the PKU-Market-PCB [15]. This development ensures that LW-YOLOv5 operates efficiently within the memory and computational constraints of embedded environments, providing a robust solution for manufacturing quality control.

The second contribution lies in conducting comprehensive comparative analysis and ablation studies to evaluate LW-YOLOv5 against state-of-the-art models, including MSD-YOLO, YOLO-LFPD, and ARMA-based YOLO, for PCB defect detection. The research implements and tests at least four optimization techniques, optimized anchor boxes via K-means clustering, NWD [44], SPD-Conv [43], and MLCA [49]to achieve a mAP@0.5 of 0.90 or higher and reduce computational complexity by 20% compared to YOLOv5n. These studies demonstrate LW-YOLOv5's superior performance, with a precision of 0.97 and recall of 0.914, highlighting its suitability for embedded systems. The ablation experiments quantify the impact of each optimization, offering insights into their contributions to improved accuracy and efficiency, which are critical for resource-constrained platforms like the Jetson Orin Nano.

A third key contribution is the successful deployment of LW-YOLOv5 on the NVIDIA Jetson Orin Nano, achieving an inference speed of at least 10 FPS at 640x640 resolution using FP16 ONNX format, with average power consumption below 7W and memory usage under 4GB. The deployment ensures thermal stability, maintaining CPU and GPU temperatures below 60°C, and targets a mAP@0.5 above 0.90 for real-time PCB defect detection. This enables practical applications in offline quality audits, addressing the need for reliable, high-speed inspection in manufacturing environments. The use of CUDA-enabled ONNX Runtime and fused convolutional operations further optimizes performance, making LW-YOLOv5 a viable solution for embedded deployment in industrial settings.

The final contribution involves a comprehensive analysis of LW-YOLOv5's performance on resource-constrained platforms, specifically the NVIDIA Jetson Orin Nano, achieving a mAP@0.5 above 0.90, inference speed of at least 10 FPS, and memory usage below 1GB using ONNX Runtime. The research improves detection of small defects, such as mouse bites, and resolves class ambiguities (e.g., spur vs. spurious copper) through targeted data augmentation techniques, including horizontal/vertical flips, brightness adjustments, and motion blur. These efforts result in a 10% reduction in false positives and negatives, enhancing the model's robustness and reliability. By addressing these challenges, the research provides a scalable framework for deploying lightweight deep learning models in manufacturing, paving the way for future advancements in AI-driven quality control.

1.5 Report Organizations

The details of this research are discussed in this section. In Chapter 2, a comprehensive review of existing literature on PCB defect detection is conducted, focusing on four selected studies that utilize state-of-the-art methods. This includes an analysis of their strengths, weaknesses, and applicability to real-world scenarios. Additionally, three studies related to PASCAL VOC [16][17] object detection are discussed to highlight advancements in object detection methodologies. A comparative analysis between the selected papers is presented to identify research gaps and opportunities for improvement, which inform the direction of this study.

Chapter 3 explains the end-to-end methodology used to build a lightweight PCB-defect detector that runs on embedded hardware. It begins by describing the six defect classes in the PKU synthetic PCB dataset and outlines how an eighteen-fold augmentation schedule—flips, brightness-contrast shifts, rotation, motion-blur and resize—expands the modest training set into a richly varied corpus. The chapter then introduces the LW-YOLOv5 architecture, showing how the YOLOv5-nano backbone is progressively replaced with space-to-depth down-samplers, reparametrized attention blocks, Ghost-based dynamic convolutions, mixed-local channel attention, and cross-resolution fusion to retain fine detail while holding the model to about 1.2 million parameters. Anchor boxes are re-clustered with K-means and the Normalised Wasserstein Distance loss is adopted to stabilise training on tiny features. The section closes with a timeline that maps data preparation, model design, training, ablation and deployment tasks onto a 20-week schedule, ensuring each stage directly supports the project's SMART objectives.

In Chapter 4, This chapter translates the methodology into an executable blueprint. It opens with a high-level block diagram that maps the eight workflow phases—from raw-image acquisition through comparative analysis—and then expands each phase inside a detailed component diagram. The hardware subsection specifies the development laptop and the Jetson Orin Nano target, while the software subsection lists the Python-Colab tool-chain, machine-learning frameworks and deployment utilities. Data-flow narratives describe how images are split, augmented eighteen-fold and funnelled through the pipeline; how the LW-YOLOv5 configuration replaces standard YOLOv5-nano layers with SPD-Conv, RCSOSA, GhostDynamic, MLCA and CRFM modules; and how K-means anchors and a Wasserstein loss are prepared. A concise table summarises hyper-parameter selections and their rationale. The chapter closes by previewing the training, evaluation and deployment tasks that will be executed in later chapters.

In Chapter 5, Implementation details show how the design is realised in both hardware and software. The hardware section documents physical set-ups: a Ryzen-RTX3050 laptop for model development and the Jetson Orin Nano for embedded deployment, each accompanied by a specification table. The software section reviews development tools, libraries and frameworks, followed by step-by-step configuration on Google Colab and Ubuntu 22.04 LTS. Screenshots illustrate package installation, dataset download, augmentation scripts, anchor re-clustering and YAML generation. Subsequent sections walk through model training with tuned hyper-parameters, validation with TensorBoard and export to PyTorch and ONNX artefacts. Operational screenshots demonstrate three execution modes—FP32 PyTorch, FP16 PyTorch and CPU-only ONNX—and log their latencies, power draw and memory footprints. Implementation issues such as quantisation failures, TensorRT incompatibilities and script patching are summarised, and the chapter ends with a brief remark that LW-YOLOv5 reaches the edge device intact but still resists aggressive compression.

In Chapter 6, Evaluation begins by defining detection-quality and computational-cost metrics, then reports results for the baseline YOLOv5-nano, the augmented baseline and the proposed LW-YOLOv5. Training and validation curves confirm stable convergence, and class-level plots highlight residual weakness on spur and spurious-copper defects. Jetson profiling shows that the FP16-ONNX build runs at 21 FPS with 6 W average power—suitable for offline audits but below the 60 FPS inline AOI goal. A comparative study ranks LW-YOLOv5 ahead of other compact detectors on accuracy-per-parameter, while ablation tables attribute most gains to SPD-Conv, NWD and MLCA. Error analysis identifies three failure modes—class ambiguity, low-contrast misses and lighting-induced false positives—and proposes targeted remedies. A challenges section reflects on model-compression roadblocks and tool-chain quirks, and an objectives-evaluation section scores each SMART objective: lightweight design and comparative superiority achieved; real-time speed and micro-controller deployment partially met. The chapter concludes that LW-YOLOv5 delivers high-accuracy, low-power PCB inspection on edge hardware, but further optimisation is needed for high-speed production lines.

In Chapter 7, The closing chapter distils the project’s outcomes and charts the next steps. It opens with a concise conclusion that revisits each SMART objective and records the final scorecard: LW-YOLOv5 meets the lightweight-design goal (≈1 MB, 1.18 M parameters) and surpasses the 0.90 mAP threshold, outperforms compact state-of-the-art detectors on accuracy-per-compute, and runs efficiently on the Jetson Orin Nano while maintaining sub-7 W power and safe thermals. Two objectives remain partially satisfied—real-time 30 FPS throughput and full TensorFlow-Lite-Micro portability—owing to ONNX-Runtime CPU fall-backs and operator-compatibility gaps. The discussion highlights key contributions: a validated framework for embedded PCB inspection, architectural insights that lift small-defect recall, and a data-augmentation recipe that trims false detections by roughly ten percent.

Chapter 2

Literature Review

2.1 Previous Works of YOLOs on PCB defect dataset from Peking University

2.1.1 Lightweight PCB defect detection algorithm based on MSD-YOLO

This paper [18] discusses the problem of low accuracy and slow detection rate (DR) of existing target detection algorithms for PCB defect detections. Then, it explained details of the structure architecture of the baseline YOLOv5 and summarized out its current limitations. For further discussions, it highlighted one of the problems within baseline YOLOv5, which its default anchor box is built to be more favorable for the public available datasets, COCO. Besides, multiple weaknesses from YOLOv5 were also discussed, from the perspective of its feature extraction accuracy, computational efficiency and large amount of information loss. In a more specific views, the paper discussed YOLOv5's backbone network, which is the CSPDarknet53, and the SPP module. It was mentioned that this particular backbone network, CSPDarknet53, fails to obtain non-redundant information despite it having a good extraction accuracy. Other than that, SPP module's maxpooling operation was underscored to discuss its large amount of information loss thus failing to get both local and global information. In conclusions, the paper made clear that baseline YOLOv5 was not built for its efficiency for intense computational and small object's feature representations [25] [26]. The improved structure architecture of the baseline YOLOv5, called the MSD-YOLO can be illustrated in figure 2.1.1.1

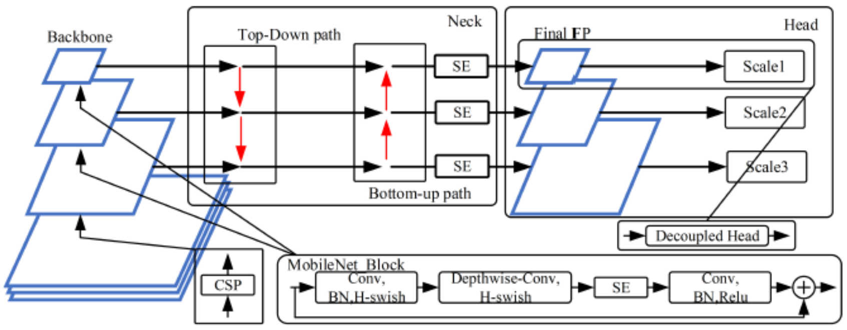


Figure 2.1.1.1 Improved Network Structure based on baseline YOLOv5.

After that, the paper discussed about its improved structure of its YOLOv5, aimed to solve all the weaknesses aforementioned exists on baseline YOLOv5. Start off with the improvement, the paper replaces the CSPDarknet53 network with the latest combined version of MobileNet-v1 and MobileNet-v2 networks in term of their strengths, which is called the MobileNet_Block module in MobileNet-v3 [27] as the backbone network. It also utilizes a lighter weight SE attention mechanism [28] and new-found activation functions. Other than that, the paper places CSPDarknet53 om the first and second layers of the network then incorporates it with MobileNet_Block module in deeper layers. However, other limitations were spotted in usage of the new MobileNet_Block module, which decreases the model accuracy while it is reducing the model's parameter through convolution operations. The paper then proposed a new method by introducing the inverted residual structure which to alleviate the training errors caused by multiple layers stacked together. Visualization of the MobileNet_Block is shown in figure 2.1.1.2

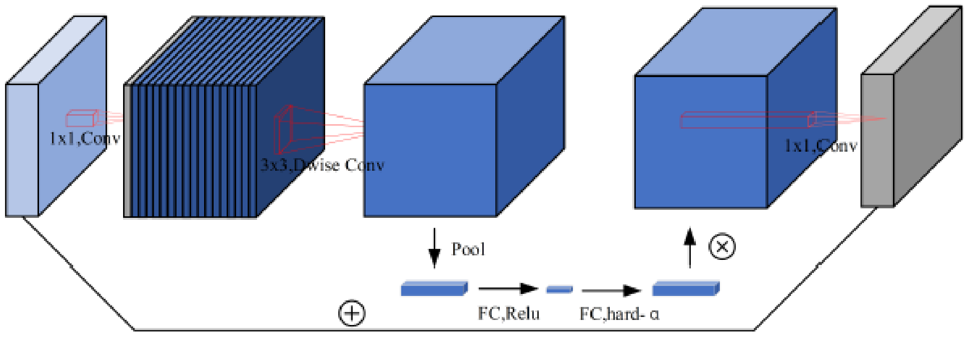


Figure 2.1.1.2 MobileNet_Block structure.

Besides that, the paper stated that depth-wise separable convolution had outrageous advantages over normal convolution in terms of their model size and computational load. The proposed method by the paper, mentioned that two steps operations for the methods. First, deep convolution is performed, where the convolution kernel is separated into single channels, and each channel is convolved individually as the operation is visualized in figure 2.1.1.3. This process then produces an output feature map with the same number of channels as the input feature map. Secondly, a pointwise convolution is applied using a 1x1 convolution kernel. This step combines the feature maps from the previous layer by weighting them and merging them along the depth dimension to create a new feature map.

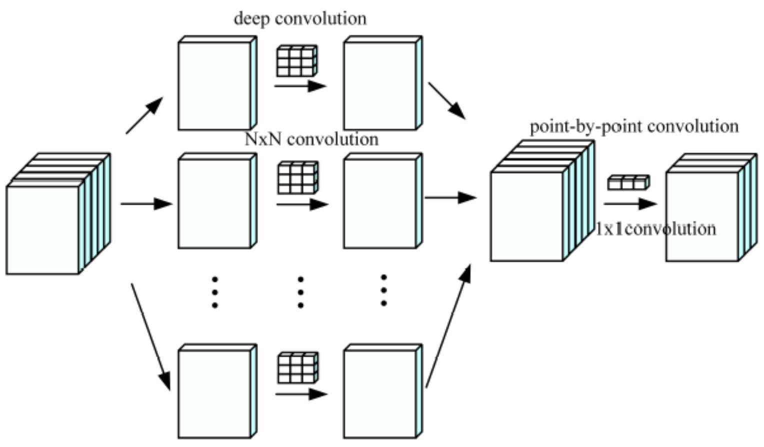
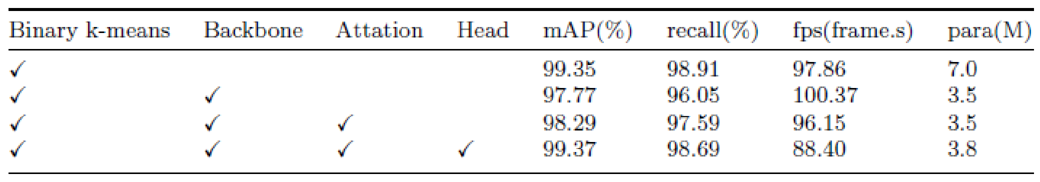


Figure 2.1.1.3 Depth-separable convolution illustration.

Then, the paper further discussed about a new activation function, h-swish activation function and the SE attention mechanism. This activation function was used because of its good performance compared to regular ReLU function and its characteristics of solving the saturation problem in neural net. The paper stated that as the network becomes deeper, the cost of using nonlinear activation functions decreases, which can help reduce the overall parameter size. On the other hand, the SE attention mechanism is used to allocate computational resources more effectively by focusing on the most informative parts of an image, which enhances the detection of objects of interest while minimizing background noise. Thus, it made up a great improvement for the proposed model.

Finally, a decoupled detection head is proposed to deal with the problem of high coupling between class and positional information in features channel. This adjustment made the head able to isolate and discover information separately. However, decoupled head led to another problem which is specific classification and localizations misalignment. The paper then proposed a good method like add in IOU for the regression branch and separately trained it with other two branches. From the data pre-processing steps, the paper suggests data augmentation techniques to process the images from PKU-Market PCB dataset [15]. Hence, the paper concluded that their detection rate (DR) had improved after using the new backbone, MobileNetv3. However, their attribute extraction abilities were weakened as well. A tabular forms view was constructed to made clear of the adjustment and modifications made that helped with the results of mean Average Precision (mAP), recall, frame per second (fps) and parameters (M) in figure 2.1.1.4.



Binary k-means	Backbone	Attation	Head	mAP(%)	recall(%)	fps(frame.s)	para(M)
✓				99.35	98.91	97.86	7.0
✓	✓			97.77	96.05	100.37	3.5
✓	✓	✓		98.29	97.59	96.15	3.5
✓	✓	✓	✓	99.37	98.69	88.40	3.8

Figure 2.1.1.4 Ablation experiments on tabular forms for model comparison.

2.1.2 Optimized Lightweight PCB Real-Time Defect Detection Algorithm

In this research, the paper [19] did a similar approach like [18] that using baseline YOLOv5 then slowly converging and doing various modification towards a more improved version. Firstly, the paper proposed a similar backbone structure that used MobileNetv3 and include a different sub-module of attention mechanism to prevent downfall of accuracy in its own models. Then, the paper utilized a fast normalized bidirectional feature pyramid and same decoupled head concept for the similar reason. As for the novelty of this approach, the paper utilized pruning techniques as one of their final step processes for the lightweight model. On behalf of the data pre-processing steps, common data pre-processing techniques are used. These techniques included common data augmentation techniques such as grayscale transformation, colour enhancement and others. Also, enlargement of dataset size, prevention of overfitting model, acceleration of training time and enhancement of detection performance were made along the way.

Starting off with the structure backbone of the improved lightweight YOLOv5, the paper incorporated MobilenetV3-small in the first 12 layers as attribution extraction. Then, fusion of local and global features was performed using SPPF. Not to mention, MobileNetv3 model had selected 11 inverted residual modules based off different convolutions. Adopting [29] research, two activation function were directly applied, which is ReLU and H-Swish, while the first three being the ReLU. Structural layers of the MobileNetV3 are shown in figure 2.1.2.1. Besides that, the paper first mentioned about limitation of potential accuracy loss, then further proposed the usage of convolutional attention module [30] (CBAM). In short, the paper also introduced a novel attention sub-module called C3AM by combination of CBAM and Bottleneck structures. The uses of the new attention sub-module allowed the baseline YOLOv5 to further hold and combine information.

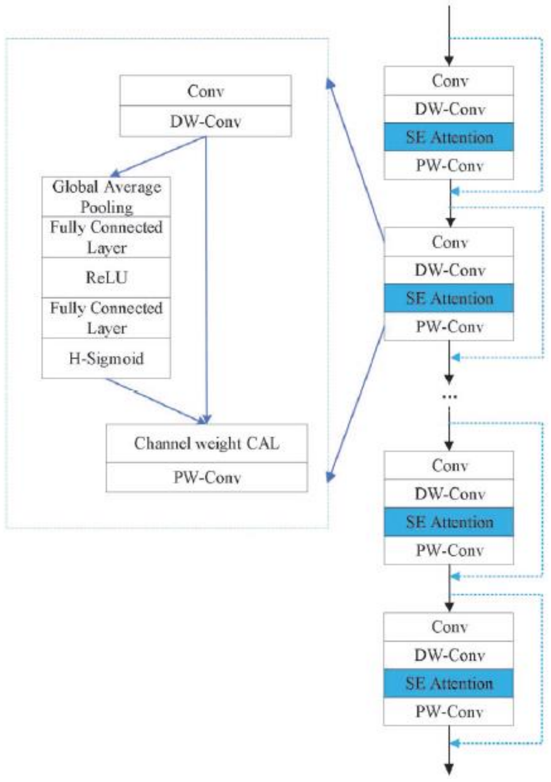


Figure 2.1.2.1 Structural Layers of MobileNetV3.

From the neck structure of the baseline YOLOv5, the original FPN and PANet structure accumulated information from numerous levels of the network. Here, the paper proposed a new improvement upon FPN by disregarding the node with only input edge, which is the BiFPN network. Other than that, the paper determined the learning parameters to make sure the reliability of the input layer size. Also, Fast Normalization Fusion [31] were utilized while it can add weight to each layer and accomplish weighted feature fusion. In short, reduction of computational rate and increase of speed and capability had been seen from the mentioned approach. When looking at the structure of the defection head, the paper offered a hybrid channels strategy which is to separate classifications and positioning, decoupling the defection head, called the Light-Decouple Head method. [32]

Moreover, model compression techniques like pruning were also applied in this discussed paper while L1 regularization method was utilized for control of Batch Normalization Layer (BNL) for the model size issues. From the settings, the controlled BN layer hen pushed the scaling factor value of BNL to 0. The process of finding convolution layers that had little influence were done through sparse training and repeated channel pruning was conducted. The channel pruning principle can be illustrated in figure 2.1.2.2.

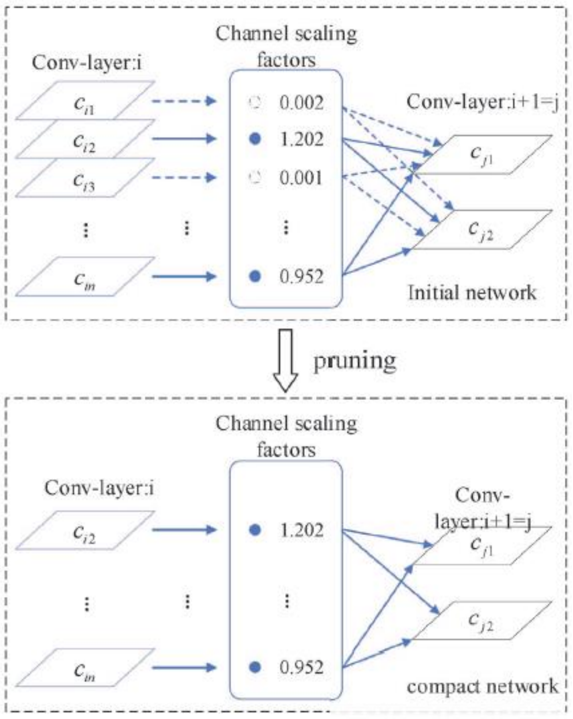
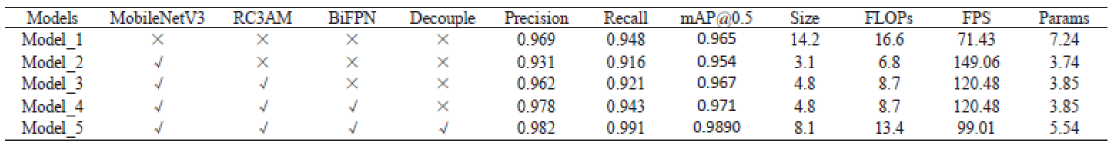


Figure 2.1.2.2 Channel Pruning Principle

When it comes to evaluation works using several versions of model of YOLOv5, the paper illustrated the results and improvement which obtained from the results of the experiment. The paper had highlighted that MobileNetV3 had quite limited attributes extraction qualifications. The full illustration of results can be shown in figure 2.1.2.3



Models	MobileNetV3	RC3AM	BiFPN	Decouple	Precision	Recall	mAP@0.5	Size	FLOPs	FPS	Params
Model_1	×	×	×	×	0.969	0.948	0.965	14.2	16.6	71.43	7.24
Model_2	√	×	×	×	0.931	0.916	0.954	3.1	6.8	149.06	3.74
Model_3	√	√	×	×	0.962	0.921	0.967	4.8	8.7	120.48	3.85
Model_4	√	√	√	×	0.978	0.943	0.971	4.8	8.7	120.48	3.85
Model_5	√	√	√	√	0.982	0.991	0.9890	8.1	13.4	99.01	5.54

Figure 2.1.2.3 Tabular comparison with different architecture of models.

2.1.3 YOLO-LFPD: A Lightweight Method for Strip Surface Defect Detection

The paper [20] had highlighted the reason of why they chosen PKU-Market-PCB [15] as their datasets for evaluation. The reason is that the image within the dataset were containing large amount of surface defects, which make it great for overview of strip defects detection as a test data. Although the objective of the paper was not specifically detecting for PCB defects detection, it was still a great reference to be reviewed as the concept was similar for the ideas of surface detection and operating on a low power, memory and computational efficient microcontroller. Initially, the paper had stated YOLOv5 as the best model for having a greater scalability and open sources resources when it compares to YOLOv8. As comparison, the paper had previously proposed a detection model called YOLO-FPD for the same purpose, which then compares it to the latest model with different architecture structure, called the YOLO-LFPD to tackle with strip surface defects detection.

Start off with the latest structure of the YOLO-LFPD, this model had integrated Fasternet [33] backbone network and RepVGG onto the fundamental of the YOLO-LFPD which can be illustrated in figure 2.1.3.1. Also, pruning techniques was utilized to make sure that this model can be used in a resource-constrained environments and decreases the computational costs. Specifically, YOLO-LFPD introduced the RepVGG module was since it had an efficient convolutional neural network design with strong capabilities of model compression. Thus, it can greatly lower the computational demands of the model while preserving high detection accuracy. Besides, the usage of Fasternet as it can efficiently extract feature information from strip images and improve the model's ability to detect defects at various scales by using multi-scale feature fusion.

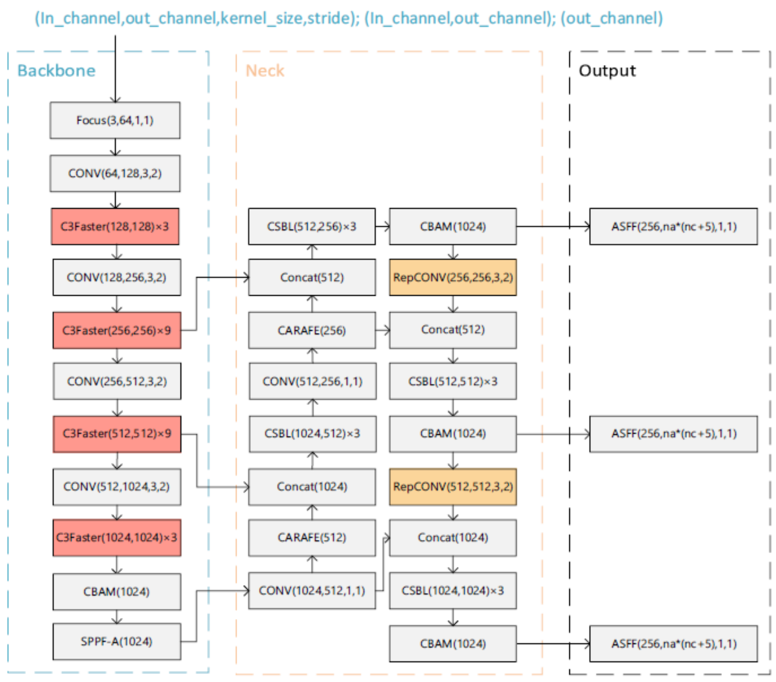
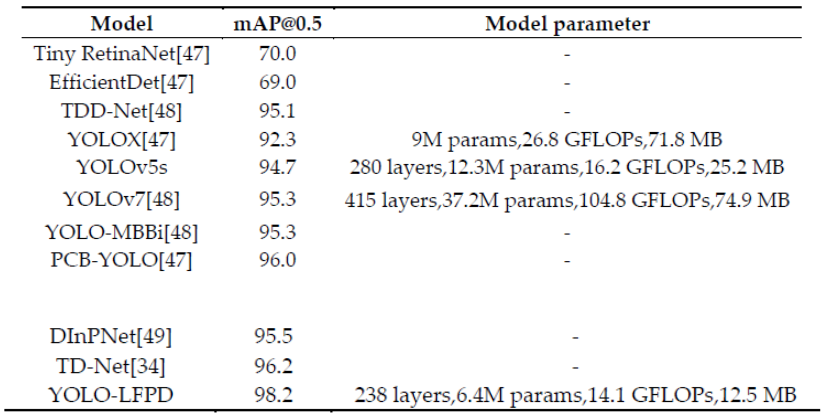


Figure 2.1.3.1 The architecture structure of the model YOLO-LFPD.

On the experiment results on PKU-Market-PCB datasets [15], as shown in figure 2.9, we can see that various performance of models on mean Average Precision (mAP) and their respective model parameter. The overall paper consists of minor error of misplacing words, including the mentioned figure 2.1.3.2, the paper stated comparison of models for PASCAL VOC 2007 datasets instead of PKU-Market-PCB [15]. Additionally, spelling mistakes, inconsistent comparison and not peer-reviewed between the mentioned datasets made this paper detracts from its overall academic quality and reliability.



Model	mAP@0.5	Model parameter
Tiny RetinaNet[47]	70.0	-
EfficientDet[47]	69.0	-
TDD-Net[48]	95.1	-
YOLOX[47]	92.3	9M params,26.8 GFLOPs,71.8 MB
YOLOv5s	94.7	280 layers,12.3M params,16.2 GFLOPs,25.2 MB
YOLOv7[48]	95.3	415 layers,37.2M params,104.8 GFLOPs,74.9 MB
YOLO-MBBi[48]	95.3	-
PCB-YOLO[47]	96.0	-
DInPNet[49]	95.5	-
TD-Net[34]	96.2	-
YOLO-LFPD	98.2	238 layers,6.4M params,14.1 GFLOPs,12.5 MB

Figure 2.1.3.2 Evaluation Results on Various Models.

2.1.4 Light-YOLOv5: A Lightweight Algorithm for Improved YOLOv5 in PCB Defect Detection

The paper [21] had chosen YOLOv5 as their baseline due to its excellent performance and suitable for lightweight modifications. Generally, the paper summarized its work in four stages, which is the input, backbone, neck and prediction stages. Firstly, enrichment of the dataset is done on the input stage, followed by their improved SPPF which fasten up the calculation speed. Besides, introduction of CSP layers and shuffle attention mechanism module had been added into the backbone of YOLOv5. Finally, the paper utilized slim-neck structure as one of the improvements for their model. The paper visualized the overall model structure after final improvement as figure 2.1.4.1.

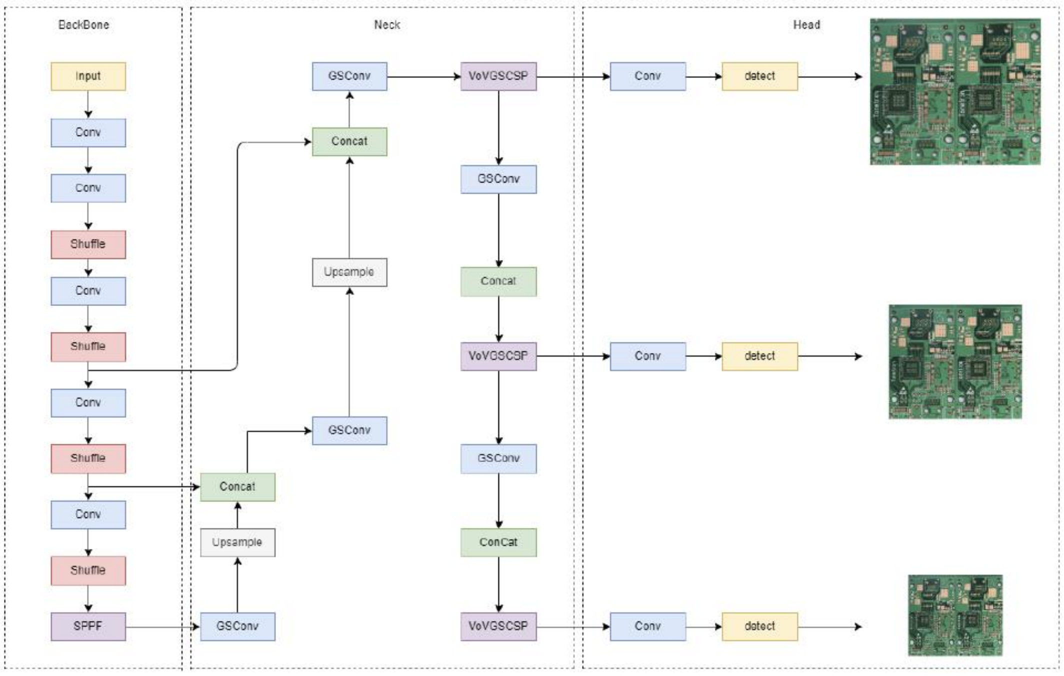


Figure 2.1.4.1: Lightweight YOLOv5 model architecture

Moreover, In the network's neck section, it used the GSConv module instead of the standard convolution (SC) due to its lower computational cost as it was only using about 60% of the SC while providing similar learning ability. Visualization of the architecture of GSBottleneck module can be seen in figure 2.1.4.2. However, using GSConv throughout the entire network stages would lead to increased computational complexity because it could hinder the flow of data as the feature map progresses. To address this, GSConv is used selectively only in the neck stage, where the channel dimensions have already reached their maximum, and no further conversion is needed. The VoV-GSCSP module is introduced as an improved cross-stage partial network based on GSConv, due to its decreases in inference time and the complexity of the network structure while still maintaining high accuracy. Additionally, it replaced the CSP module in the neck layer, which traditionally uses standard convolutions, resulting in a significant reduction in floating point operations (FLOPs) which can be visualized in figure 2.1.4.3.

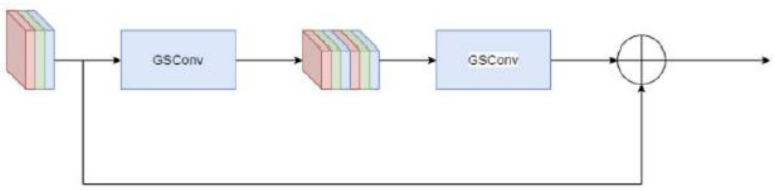


Figure 2.1.4.2: Construction of the Gsbottleneck component.

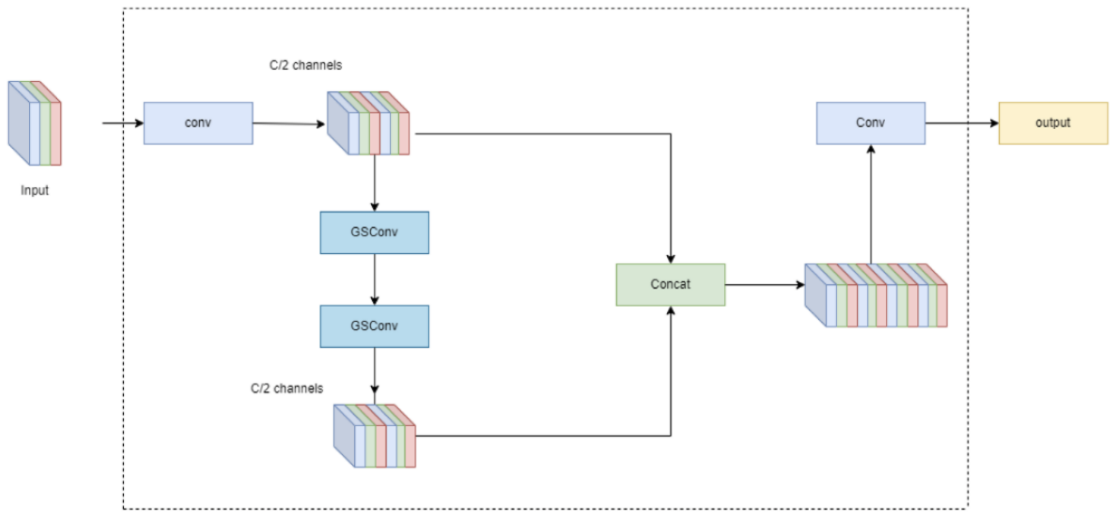
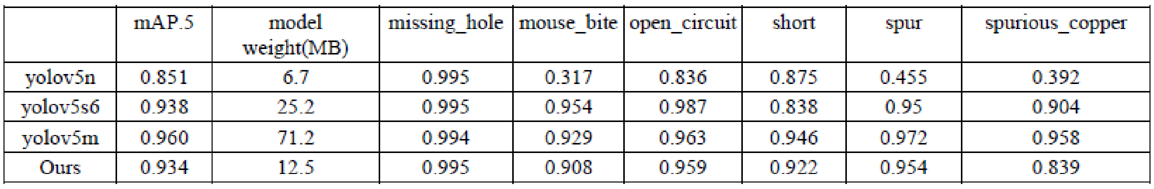


Figure 2.1.4.3: The partial network module – VoV-GSCSP built upon fundamental of GSConv.

Essentially, the paper removal of focus layer was performed due to the additions of the shuffle attention mechanism. While in the backbone, issues caused by the usage of C3 layers in baseline YOLOv5 had significant downfalls on occupations of cache space and reduction of computational speed. Therefore, the paper proposed a shuffle module, for integration of channel attention and spatial attention. For the channel attention module, the SE module has been modified due to high number of parameters, which makes it challenging to balance real-time detection and accuracy. Then, the paper proposed a solution by using a lightweight version of SE module which is ECA for generating channel weights. However, with its one-dimensional convolution, the paper proposed the usage of global average pooling (GAP). This method is great for obtaining global information by generating statistical data. Besides, for the spatial attention module, the focus shifts from channel importance to the location of the information. This module is designed to complement channel attention by highlighting where the important features are in the spatial dimensions. To achieve this, the paper proposed spatial data statistics are generated using group normalization (GN), which helps the network pay attention to the relevant spatial positions of the features. Finally in the input stage, the authors implement a two-dimensional max pooling layer (conv2d) based on the Spatial Pyramid Pooling (SPP) used in YOLOv5 to enhance computation speed. This modification, called SPPF, allows the model to convert feature maps of any size into fixed-size feature vectors in real time.

After modification that results in an improved lightweight version of YOLOv5, the authors conduct evaluation experiment on PKU-Market-PCB datasets [15] which on different variants of YOLOv5. Brief explanation of the YOLOv5 models used in the comparison done in figure 2.1.4.4: YOLOv5n – Smallest and most efficient version in the YOLOv5 family. YOLOv5s6 – Balance between accuracy and efficiency while being suitable for use on larger input image sizes. YOLOv5m – Balance between model size, accuracy and computational efficiency.



	mAP.5	model weight(MB)	missing_hole	mouse_bite	open_circuit	short	spur	spurious_copper
yolov5n	0.851	6.7	0.995	0.317	0.836	0.875	0.455	0.392
yolov5s6	0.938	25.2	0.995	0.954	0.987	0.838	0.95	0.904
yolov5m	0.960	71.2	0.994	0.929	0.963	0.946	0.972	0.958
Ours	0.934	12.5	0.995	0.908	0.959	0.922	0.954	0.839

Figure 2.1.4.4: Performance comparison between different variants of YOLOv5.

2.2 Previous Works on Object Detection

2.2.1 Flexible and Fully Quantized Lightweight TinyissimoYOLO for Ultra-Low-Power Edge Systems

This paper [22] had their focuses on deployment and execution of TinnyissimoYOLO [34], which then expanded its approach to a more lightweight and generalized network that is suitable for microcontroller unit (MCU). While the visualization of its architecture can be seen in figure 2.2.1.1, the paper stated that its network can also be adjusted, affecting only the parameter count and computational costs of the final layer. This paper also investigated the impact of different configurations on detection performance by training TinyissimoYOLO with a range of input resolutions, output classes, and kernel sizes, all without imposing constraints on the datasets or the number of objects per image.

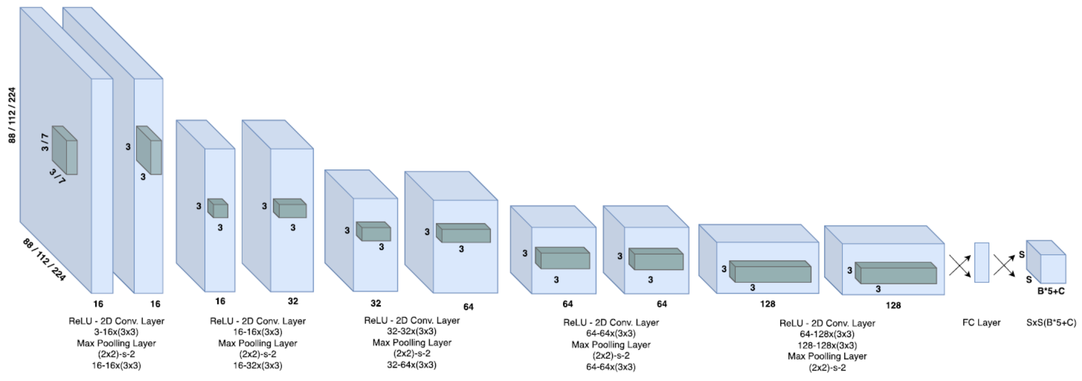
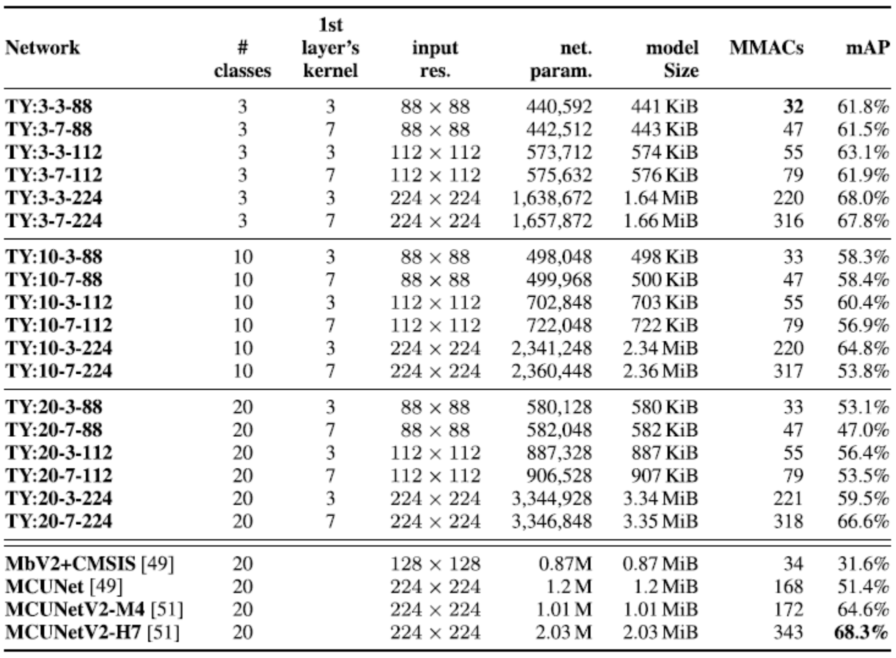


Figure 2.2.1.1: Architecture Structure of TinyissimoYOLO.

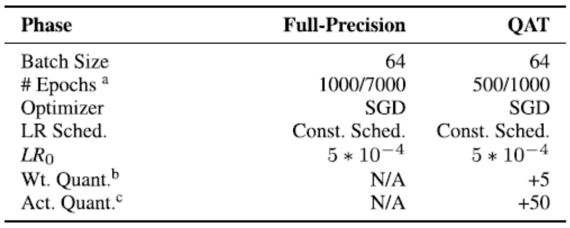
Moreover, the paper continues to discuss about the purpose of using PASCAL VOC dataset [16][17], which is due to its manageable size and diverse range of object class. The authors further stated that the number of parameters in the network's output layer increases linearly with the number of object classes. As parameter comparisons visualized in figure 2.2.1.2, the paper decided on not to choose any large dataset with additional object classes due to the final output will end up in a larger TinnyissimoYOLO network. In this study, 90% of the Pascal VOC training dataset was used to train the network, while the remaining 10% was reserved for validation. Lastly, various techniques of such as data augmentation, geometric transformation and photometric changes had been applied.



Network	**# classes**	**1st layer's kernel**	**input res.**	**net. param.**	**model Size**	**MMACs**	**mAP**
TY:3-3-88	3	3	88 × 88	440,592	441 KiB	**32**	61.8%
TY:3-7-88	3	7	88 × 88	442,512	443 KiB	47	61.5%
TY:3-3-112	3	3	112 × 112	573,712	574 KiB	55	63.1%
TY:3-7-112	3	7	112 × 112	575,632	576 KiB	79	61.9%
TY:3-3-224	3	3	224 × 224	1,638,672	1.64 MiB	220	68.0%
TY:3-7-224	3	7	224 × 224	1,657,872	1.66 MiB	316	67.8%
TY:10-3-88	10	3	88 × 88	498,048	498 KiB	33	58.3%
TY:10-7-88	10	7	88 × 88	499,968	500 KiB	47	58.4%
TY:10-3-112	10	3	112 × 112	702,848	703 KiB	55	60.4%
TY:10-7-112	10	7	112 × 112	722,048	722 KiB	79	56.9%
TY:10-3-224	10	3	224 × 224	2,341,248	2.34 MiB	220	64.8%
TY:10-7-224	10	7	224 × 224	2,360,448	2.36 MiB	317	53.8%
TY:20-3-88	20	3	88 × 88	580,128	580 KiB	33	53.1%
TY:20-7-88	20	7	88 × 88	582,048	582 KiB	47	47.0%
TY:20-3-112	20	3	112 × 112	887,328	887 KiB	55	56.4%
TY:20-7-112	20	7	112 × 112	906,528	907 KiB	79	53.5%
TY:20-3-224	20	3	224 × 224	3,344,928	3.34 MiB	221	59.5%
TY:20-7-224	20	7	224 × 224	3,346,848	3.35 MiB	318	66.6%
MbV2+CMSIS [49]	20		128 × 128	0.87M	0.87 MiB	34	31.6%
MCUNet [49]	20		224 × 224	1.2 M	1.2 MiB	168	51.4%
MCUNetV2-M4 [51]	20		224 × 224	1.01 M	1.01 MiB	172	64.6%
MCUNetV2-H7 [51]	20		224 × 224	2.03 M	2.03 MiB	343	**68.3%**

Figure 2.2.1.2 Tabular comparison of parameters evaluated on PASCAL VOC.

As for the network training and quantization, the paper utilized QuantLab [35], an open-source framework based on PyTorch that supports quantization-aware training (QAT). In this study, the targeted hardware platforms were optimized for network with 8-bit weight and activation precisions. Also, the study highlighted problem of smaller bit-widths usage can pioneer an important implementation and runtime costs, potentially results in accuracy decrement. Therefore, the paper had proposed on evaluation of 8-bit quantized networks [36]. A two-phase training process were proposed for the TinyissimoYOLO networks. First, a full-precision network was trained to convergence. In the second phase, quantization-aware training was performed using the TQT algorithm [37]. During QAT, QuantLab converts the original architecture into a fake-quantized version, replacing each convolution where fully connected and activation layer with its quantized corresponding. Besides, the networks were initialized from full-precision checkpoints, training first with weight-only quantization before moving to full-model quantization. Their respective hyperparameter tunning were shown in figure 2.2.1.3, illustrated the performance of QAT using TQT algorithms. Once the training converged, the fake-quantized model was converted into an integer-only model using techniques like "integer channel normalization" [38] or "dyadic quantization" [39] which merge normalization, rescaling, and activation layers into requantization layers. Finally, the final model was exported as an ONNX model compatible with various hardware backends which showed no loss in accuracy compared to the full-precision model. Thus, its present is specifically optimized for deployment on GAP9 clusters.



Phase	Full-Precision	QAT
Batch Size	64	64
# Epochs [a]	1000/7000	500/1000
Optimizer	SGD	SGD
LR Sched.	Const. Sched.	Const. Sched.
LR_0	$5 * 10^{-4}$	$5 * 10^{-4}$
Wt. Quant.[b]	N/A	+5
Act. Quant.[c]	N/A	+50

Figure 2.2.1.3 Tabular views of hyperparameter training of QAT and full-precision training.

Other than that, the evaluation of the proposed algorithms on multiple platforms such as ARM, RISC-V cores, and hardware accelerators provides valuable insights into many advantages and trade-offs among each of these hardware architectures. In the comparison between these platforms, the authors point out strengths like parallel processing on multi-core RISC-V processors and higher efficiency due to hardware accelerators but concentrate on key issues such as power consumption, latency, and scalability.

2.2.2 Improved Light-Weight Target Detection Method Based on YOLOv5

The authors of this paper [23] had highlighted their pursuit of creating a more lightweight and efficient model for the baseline YOLOv5 model. The introduction of SKConv in SKNet (Selective Kernel Networks) [40] had been referenced to improve reduction of parameters and expansion of receptive field. The paper explained detail about the operation of the SKConv implementation, which included three processes, split, fuse and select. In the study, the paper then stated two of the characteristics about SKConv, which is the usage of group convolution and dimensionality reduction operation in the Select part of SKConv. Both adoptions had caused information loss and blurred relationship between corresponding model's channel weights. To solve this existing problem for the utilization of SKConv, the authors proposed two novel approach which by introducing improved SKCon-G series and feature splicing method, called the equal interval interpolation. Further visualization of the comparison of architecture structure of the model network design between SKConv and SKCon-G can be seen in figure 2.2.2.1.

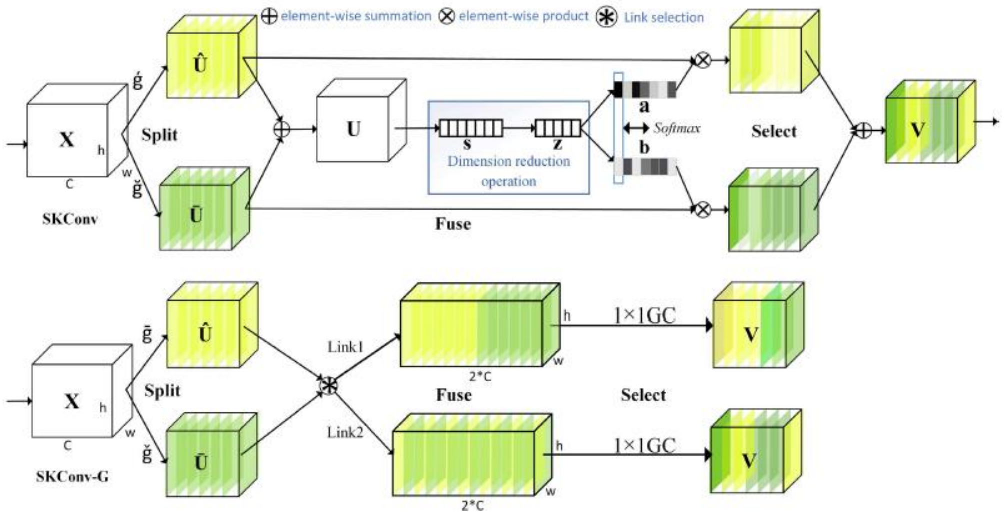
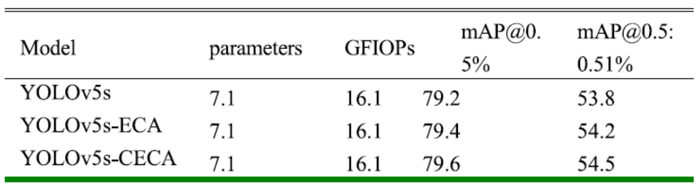


Figure 2.2.2.1: Comparison of SKConv (Upper) and SKConv-G (Lower) model network design.

On the other hand, when looking at the channel attention module, the paper had utilized standard ECA module initially then switched to the improved C-ECA (Circular-ECA) module [41] based on the initial module. The reason of replacing can be deducted as insufficient attention received at the output edges for the channel features which indirectly impact parameter updates during loss backpropagation. Other than that, the introduction of C-ECA module was to solve the SE's information loss problem caused by the reduction of channel dimensionality. Essentially, the replacement of the ECA module can be summarized as to reassign channel weights for allowing better lightweight network to focus more on learning the parameters of the key feature channels with consideration of limited number of parameters. The paper visualized the comparison between ECA and C-ECA modules in figure 2.2.2.2 when tested on PASCAL VOC 2007 test dataset [16][17] in term of mean Average Precision (mAP) and GFLOPs.



Model	parameters	GFIOPs	mAP@0.5%	mAP@0.5:0.51%
YOLOv5s	7.1	16.1	79.2	53.8
YOLOv5s-ECA	7.1	16.1	79.4	54.2
YOLOv5s-CECA	7.1	16.1	79.6	54.5

Figure 2.2.2.2: Comparison results between usage of ECA and C-ECA.

As we discussed previously about the replacement of SKConv and C-ECA modules into the improved network architecture, there are still several improvements made by the authors which one of them include a novel approach. Start off with the introduction of a new structural components called DarkUnit(G) and DarkUnitX(G) modules. Moreover, the paper mentioned the differences between both modules, while the first is primarily used as a down-sampling module to helped feature extraction, the latter is to maintains same input and output shape which improved model's strength. From there, the paper had redesigned network utilized a new CSPDarknet53 structure as the backbone, replacing the traditional down-sampling strategies with the more efficient DarkUnit(G) modules. Furthermore, the paper proposed a background with CSP+PAN structure for the neck which include multiple modules like CBS, DarkUnit(G) and C3_2(G) shown in figure 2.2.2.3.

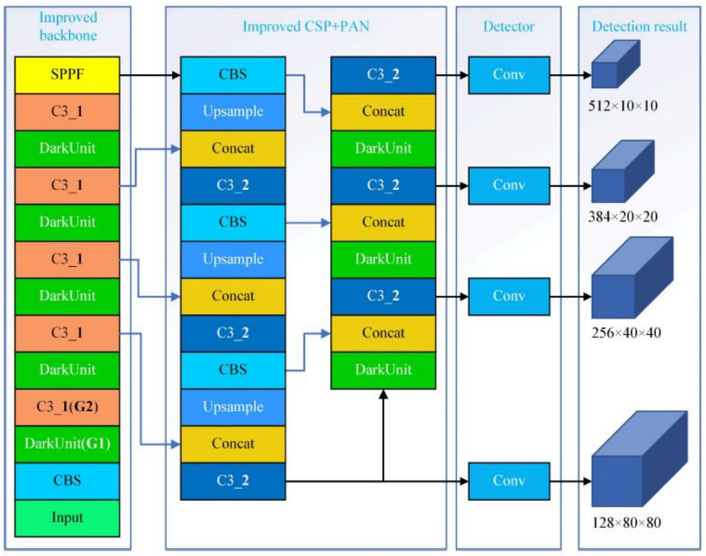
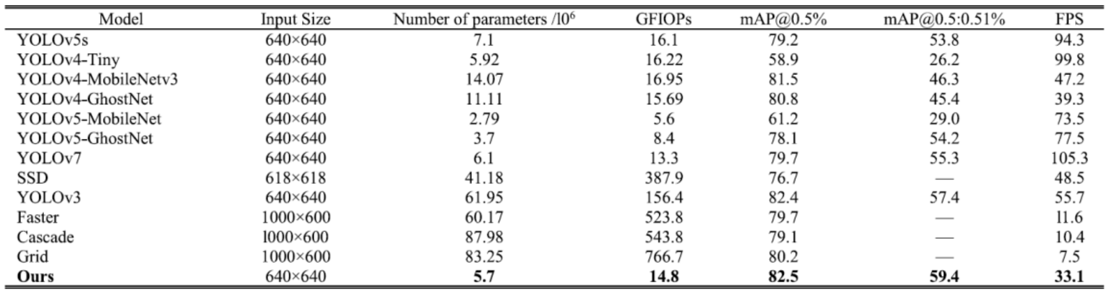


Figure 2.2.2.3: Redesigned network structure based off baseline YOLOv5.

For the experiment results, the paper had conducted comparison between various existing models on the PASCAL VOC 2007 test datasets, which can be shown in figure 2.2.2.4.



Model	Input Size	Number of parameters /10^6	GFIOPs	mAP@0.5%	mAP@0.5:0.51%	FPS
YOLOv5s	640×640	7.1	16.1	79.2	53.8	94.3
YOLOv4-Tiny	640×640	5.92	16.22	58.9	26.2	99.8
YOLOv4-MobileNetv3	640×640	14.07	16.95	81.5	46.3	47.2
YOLOv4-GhostNet	640×640	11.11	15.69	80.8	45.4	39.3
YOLOv5-MobileNet	640×640	2.79	5.6	61.2	29.0	73.5
YOLOv5-GhostNet	640×640	3.7	8.4	78.1	54.2	77.5
YOLOv7	640×640	6.1	13.3	79.7	55.3	105.3
SSD	618×618	41.18	387.9	76.7	—	48.5
YOLOv3	640×640	61.95	156.4	82.4	57.4	55.7
Faster	1000×600	60.17	523.8	79.7	—	11.6
Cascade	1000×600	87.98	543.8	79.1	—	10.4
Grid	1000×600	83.25	766.7	80.2	—	7.5
Ours	640×640	**5.7**	**14.8**	**82.5**	**59.4**	**33.1**

Figure 2.2.2.4: Comparison between various models on PASCAL VOC 2007 test [17][18].

2.2.3 MPQ-YOLO: Ultra-low mixed-precision quantization of YOLO for edge devices deployment

In this study, the author had proposed a novel quantization network, called the MPQ-YOLO, which stands for Mixed-Precision Quantization for YOLO. This framework is specifically designed to optimize the baseline YOLOv5 model for deployment on edge devices with limited computational resources and power constraints. While traditional quantization methods tend to apply a uniform precision across all different layers across the network, substantial lost in accuracy can occurs.

With that being said, the paper proposed integration of 1-bit quantization onto the backbone of the network, significantly reducing the model’s parameter and computational demands. Other than that, the paper proposed a 4-bit quantization technique for the head which was primarily because of the sensitivity, fusion operations between features from numerous layers. The improved framework proposed by the paper based on the YOLOv5 framework was visualized on figure 2.2.3.1.

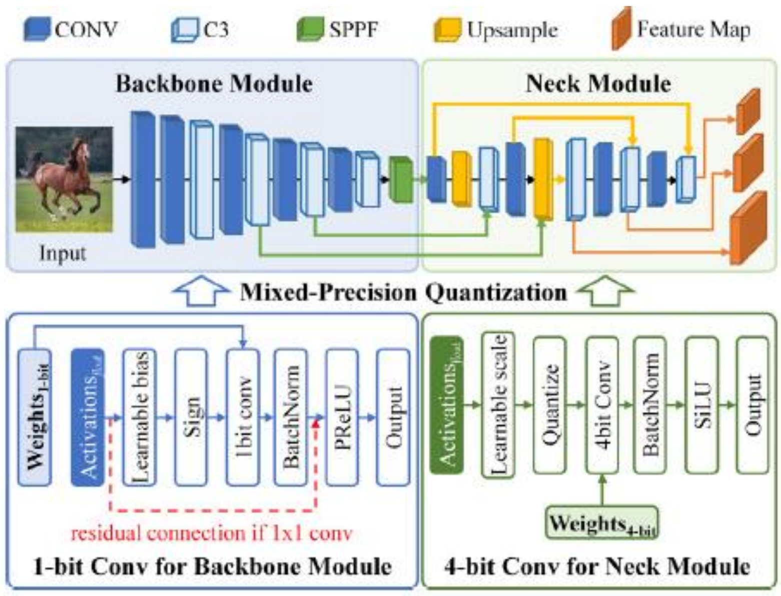


Figure 2.2.3.1 The framework of MPQ-YOLO [24].

For the implementation of the MPQ-YOLO model, the authors had introduced the progressive network quantization (PNQ) training strategy due to the abrupt transition to lower precision which causes instability and sudden drop in accuracy [42]. This technique is used to gradually reduces precision of weights and activations during training process. Additionally, diverse units in the network had carried varying levels of significance, which makes the global quantization techniques often decreases accuracy in results. In this study, the paper had demonstrated and proposed a three stages process for the quantization, which can be visualized in figure 2.2.3.2. To summarize, the three-stages process included the first, full training of the model for obtaining huge amount of significant information inside. Secondly, the head structure of full precision was kept for the adjustment operations of the Backbone using 1-bit activations and weights. Thirdly, the paper then utilize the 4-bit quantization training on the Head structure.

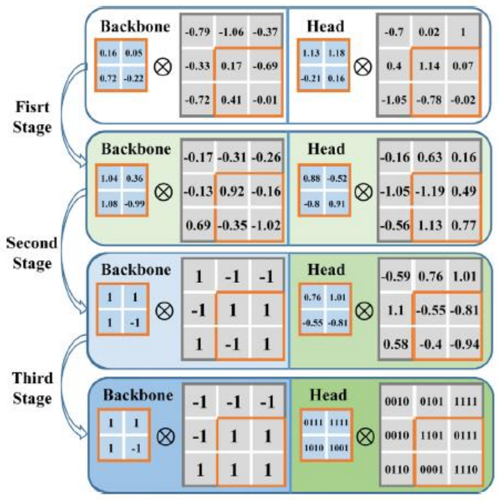


Figure 2.2.3.2 Three-step quantization process.

Also, dedicated training techniques was another key technique proposed by the authors to enhance the performance of its MPQ-YOLO on the 4-bit quantization Head. This factor dynamically adjusts the range of quantized weights and activations during training, which caused optimization of their representation in a mixed-precision environment. The paper visualized the dedicated training techniques in figure 2.2.3.3.

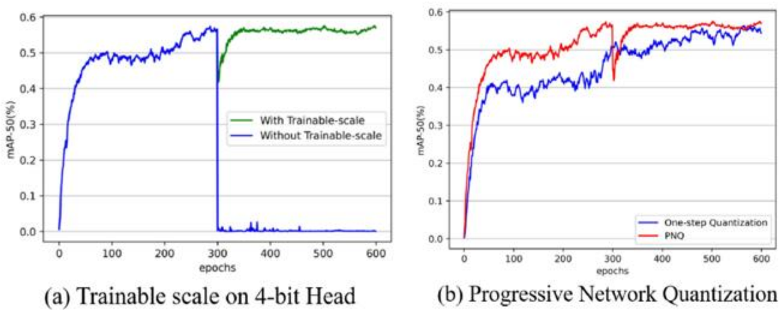
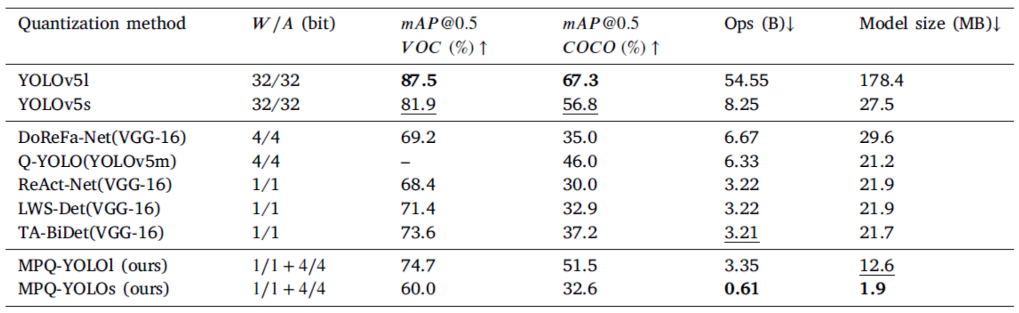


Figure 2.2.3.3: Results return on the effectiveness of usage of dedicated training techniques.

Based on the study, the authors conducted evaluation experiments with the state-of-the-art quantized target detection models on combination of PASCAL VOC 2007 and 2012 datasets [16][17] as shown in figure 2.2.3.4.



Quantization method	W/A (bit)	mAP@0.5 VOC (%) ↑	mAP@0.5 $COCO$ (%) ↑	Ops (B)↓	Model size (MB)↓
YOLOv5l	32/32	**87.5**	**67.3**	54.55	178.4
YOLOv5s	32/32	81.9	56.8	8.25	27.5
DoReFa-Net(VGG-16)	4/4	69.2	35.0	6.67	29.6
Q-YOLO(YOLOv5m)	4/4	–	46.0	6.33	21.2
ReAct-Net(VGG-16)	1/1	68.4	30.0	3.22	21.9
LWS-Det(VGG-16)	1/1	71.4	32.9	3.22	21.9
TA-BiDet(VGG-16)	1/1	73.6	37.2	3.21	21.7
MPQ-YOLOl (ours)	1/1 + 4/4	74.7	51.5	3.35	12.6
MPQ-YOLOs (ours)	1/1 + 4/4	60.0	32.6	**0.61**	**1.9**

Figure 2.2.3.4: Evaluation Results on various target detection models

2.3 Comparison of Model Between Previous Works

2.3.1 PCB defect detection datasets from Peking University

Table 2.3.1.1 Comparison between model's performance and parameters from papers reviewed for PCB defects detection

Model	**mAP(%)**	**Model Parameters**
MSD-YOLOv5 [18]	99.37	3.8M params
Optimized-YOLOv5 [19]	98.90	5.54M params, 13.4 FLOPs, 8.1MB
YOLO-LFPD [20]	98.20	238 layers, 6.4M params, 14.1 GFLOPs, 12.5MB
Light-YOLOv5 [21]	93.40	12.5MB

2.3.2 Object detection datasets from PASCAL VOC

Table 2.3.2.1 Comparison between model's performance and parameters from papers reviewed for Object Detection.

Model	**Input res.**	**mAP(%)**	**Model Parameters**
TinyissimoYOLO [22]	88 x 88	61.50	3 classes, 0.44M params, 0.44KiB
		58.40	10 classes, 0.49M params, 0.50KiB
		47.00	20 classes, 0.58M Params, 0.58KiB
	112 x 112	61.90	3 classes, 0.58M params, 0.58KiB
		56.90	10 classes, 0.72M params, 0.72KiB

		53.50	20 classes, 0.91M params, 0.91KiB
	224 x 224	67.80	3 classes, 1.6M params, 1.66MiB
		53.80	10 classes, 2.3M params, 2.36MiB
		66.60	20 classes, 3.3M params, 3.35MiB
YOLOv5 [23]	640 x 640	82.50	5.7M params, 14.8 GFLOPs
MPQ-YOLOl [24]	-	74.7	12.6MB
MPQ-YOLOs [24]	-	60.0	1.9MB

2.4 Comparison between the Techniques Used from Previous Works

2.4.1 PCB defect detection datasets from Peking University

Figure 2.4.1.1 Comparison of Techniques Used for PCB defect detection datasets

Model	Techniques / Modifications	Strengths	Weaknesses
MSD-YOLOv5 [18]	-Combines MobileNet-v3 and CSPDarknet53 for the backbone, introduces SE attention mechanism, and uses a decoupling head. Binary k-means clustering for anchor adjustment.	-Reduced parameters by 46% -Enhanced feature extraction, increased detection accuracy by 3.34%	-Requires enhanced dataset for training and more resource-demanding pre-processing.
Optimized-YOLOv5 [19]	-Uses MobileNetV3 backbone, BiFPN for feature fusion, decoupling head, and L1 regularization for sparse training and pruning.	-Achieved mAP of 99.3%, reduced model size to 40% of YOLOv5s	-Focuses on reducing complexity but may sacrifice robustness in complex PCB environments.

YOLO-LFPD [20]	-Utilizes RepVGG module and FasterNet backbone for robustness and inference speed, pruning techniques, and OTA loss function.	-Achieved 48% parameter reduction -inference time reduced by 77%,	-Requires specific hardware optimization for achieving the full potential in real-time application
Light-YOLOv5 [21]	-Introduces GSConv and VoV-GSCSP for feature extraction, shuffle attention mechanism, slim-neck structure for feature fusion, and modular replacements for improved efficiency.	-Reduced model size by 51%, improved mAP by 2-3%	Some improvement areas for robustness in handling highly noisy PCB defect datasets and complex real-world scenarios.

2.4.2 Object detection datasets from PASCAL VOC

Figure 2.4.1.2 Comparison of Techniques Used for PASCAL VOC datasets

Model	**Techniques / Modifications**	**Strengths**	**Weaknesses**
TinyissimoYOLO [22]	-Introduces a fully quantized YOLO framework with post-training quantization and optimized scaling techniques.	-Significantly reduces memory requirements without heavy accuracy degradation.	-Limited flexibility for dynamic scenarios.
YOLOv5 [23]	-Utilizes SKConv and C-ECA modules for enhanced feature	-Reduced computational effort and model parameters	-Increased depth leads to challenges in real-time

	extraction and channel attention, along with MSM for better feature pyramid depth.	by 18.6% compared to YOLOv5s.	deployment in extremely resource-constrained systems.
MPQ-YOLOl [24]	-Incorporates 1-bit backbone and 4-bit head quantization for memory efficiency. Introduces trainable scale and progressive quantization (PNQ).	-Achieves up to 16.3× model compression with 74.7% accuracy on PASCAL VOC. -Ideal for low-power, edge device deployment scenarios.	-Quantization limits model adaptability in scenarios requiring real-time training updates.
MPQ-YOLOs [24]	-Like MPQ-YOLO but focuses on incremental adjustments to the 1-bit backbone for more robust edge case detection in the PASCAL VOC context.	-Further reduced computational complexity while slightly improving accuracy in specific benchmarks.	-Further reduces generalizability due to specialized optimizations.

Chapter 3

System Methodology / Approach

This chapter presents the systematic methodology and development approach adopted throughout the project. The work focuses on designing, optimizing, and deploying a lightweight deep learning model for real-time PCB defect detection on embedded platforms. A structured pipeline was implemented to ensure efficient data preparation, model design, system training, deployment, and evaluation. In this chapter, introduction about classes of the PCB Defect Detection [15], augmentation effect done on the pipeline and the system architecture diagram was done and evaluated.

3.1 System Design Diagram / Equation

3.1.1 Introduction to classes in PCB Defect Detection

To have a better understanding on the datasets itself, I constructed tables and description to understand the problem domain. On the context side, the datasets we used are from Peking University PCB defect datasets [15], which consists of 6 type of defects which are made by photoshop. These defects are defined as missing hole, mouse bite, open circuit, short, spur and spurious copper. In following section, demonstration of each sample and labels of the type of defects will be constructed.

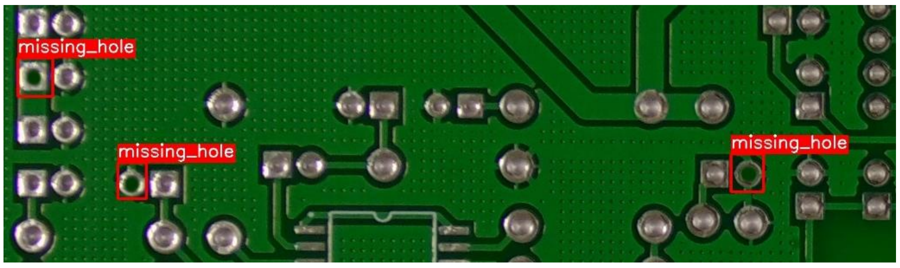


Figure 3.1.1.1 Missing hole defects in PCB defect detection [15].

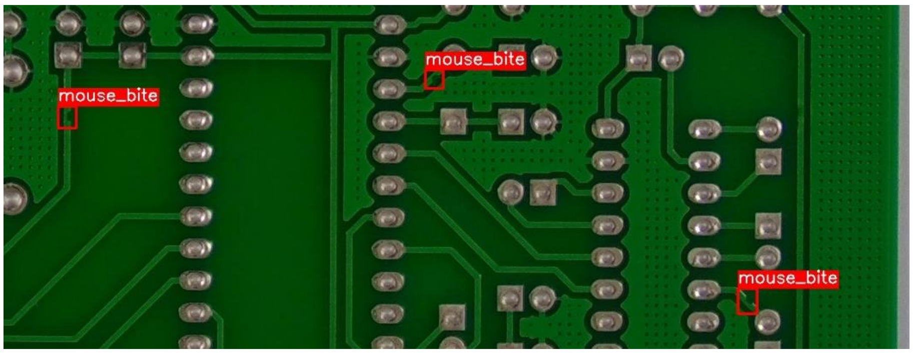


Figure 3.1.1.2 Mouse bite defects on PCB defect detection [15].

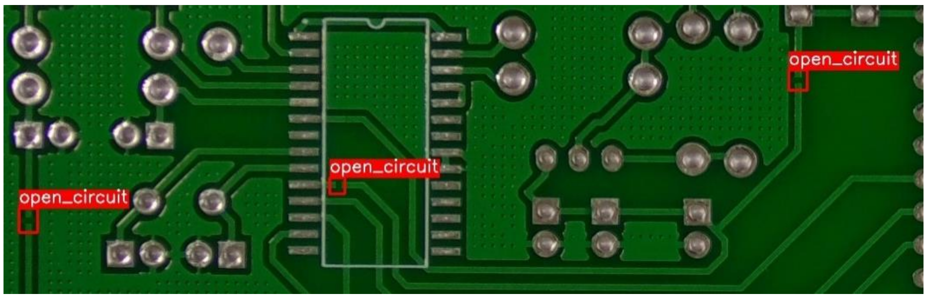


Figure 3.1.1.3 Open Circuits defects on PCB defect detection [15].

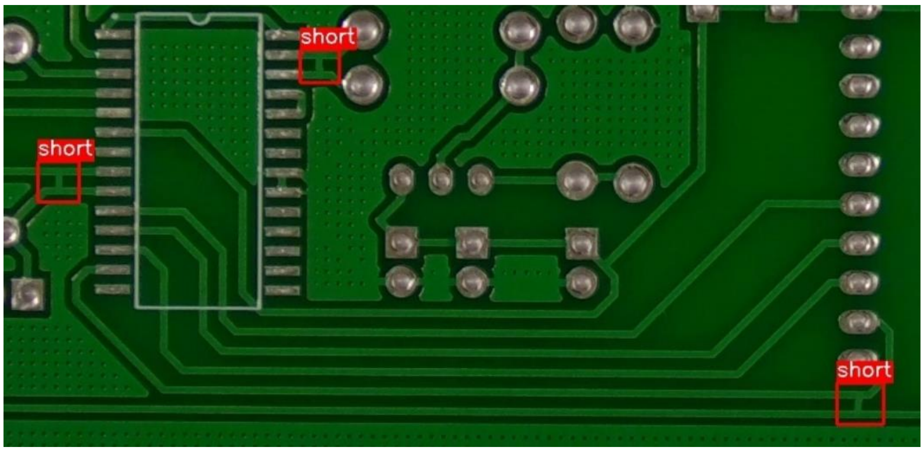


Figure 3.1.1.4 Short defects on PCB defect detection [15].

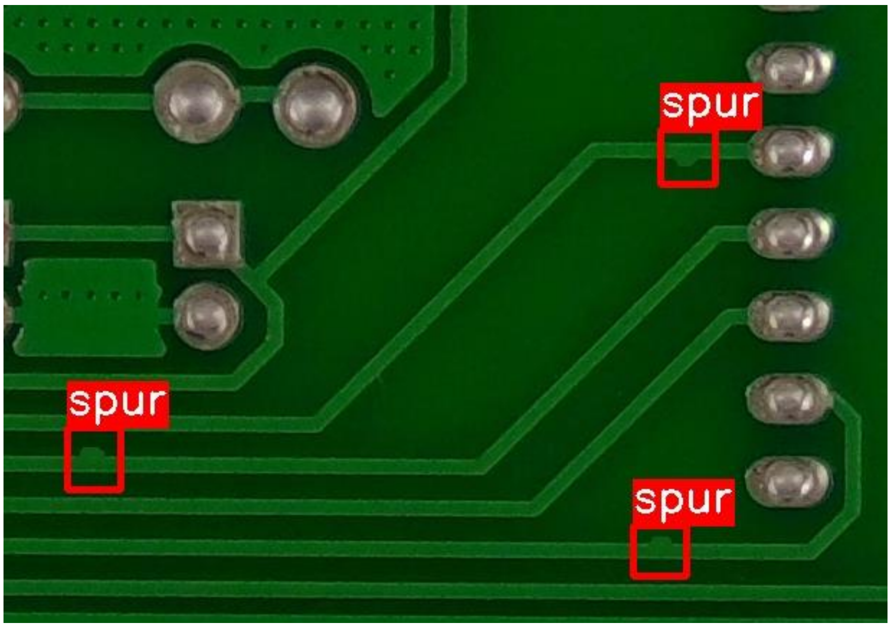


Figure 3.1.1.5 Spur defect on PCB defect detection [15].

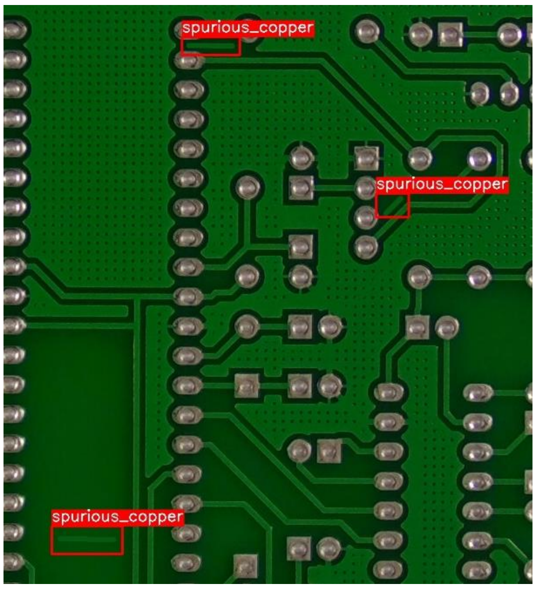


Figure 3.1.1.6 Spurious copper defects on PCB defect detection [15].

Now, visualization on the types of defects that can exist on PCB board are visualized, tabular result will be constructed for further discussing the general properties of each defect and their causes in Table 3.1.1.1.

Table 3.1.1.1 Tabular Views of Properties of PCB defects

Type of Defects	**Description of Defect**	**Causes of Defect**
Missing Hole	A plated-through or via hole that is completely	• Drill bit breakage or mis-registration

	absent after drilling/plating, leaving no metallised aperture for component leads or inter-layer connectivity.	during the drilling cycle • X-ray target-to-hole misalignment in automated drilling • Image transfer error causing pad-to-drill mismatch
Mouse Bite	A series of small perforations or nibble-shaped notches left on the board edge after break-out from a manufacturing panel; often looks like rodent bite marks.	• Depanelization via break-rout tabs instead of V-score • Too-wise perforation pattern or insufficient tab width • Mechanical stress or flexing during snap-off
Open Circuit	A broken (open) copper trace or via barrel that interrupts the intended electrical path between two nodes.	• Over-etching during wet chemical etch • Micro-cracks in via walls from thermal cycling • Drill smear or voids in copper plating
Short	An unintended electrical connection between two adjacent conductors	• Under-etching → residual copper “bridges”

	(trace-to-trace, pad-to-pad, or via-to-plane).	• Solder bridging during reflow • Incorrect solder-mask registration
Spur	A thin, whisker-like copper protrusion that extends from a trace or pad but is not part of the netlist.	• Photoresist debris causing image artefacts • Over-plating in high-current areas • Drag-out of semi-etched copper during rinse
Spurious Copper	Isolated islands or flecks of unwanted copper that are not electrically connected to the design.	• Incompletes etch (resist loss or etchant exhaustion) • Poor artwork cleaning leaving dust/dirt shadows • Lifted photoresist during develop/etch cycles

3.1.2 Introduction to Augmentation Type Applied

Although there are already existing augmentation techniques applied during original yolo training process, these techniques definitely do not make up the ability to generalize the model toward more unseen data and further augmentation process was required especially when dealing with small defects on PCB like this. In the work procedure, offline augmentation was done during data preprocessing process and application of multiple augmentation effects can be visualized in the Table 3.2 because the raw 485 training images offered limited variation. Using the Albumentations library, 18 additional views for every source image (approximate 18x expansion) with operations was chosen to mimic appearance shifts common on a production line. Also, Table 3.1.2.1 quantifies the resulting datasets growth

Table 3.1.2.1 Data-augmentation recipe for PCB-defect training set

Number of Augmentation	**Augmentation Type**	**Albumentations call (p)**
1	Horizontal Flip	HorizontalFlip(p=0.2)
2	Vertical Flip	VerticalFlip(p=0.2)
3	Random Brightness / Contrast	RandomBrightnessContrast(p=0.2)
4	Shift-Scale-Rotate	ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05, rotate_limit=15, p=0.5)
5	Motion Blur (Kernel smaller and equal than 5)	MotionBlur(blur_limit=5, p=0.2)
6	Resize to 640 x 640	Resize (640, 640, p=1.0)

The 9215 training images reported in Table 3.1.2.2 originate from a publicly available synthetic PCB-defect corpus that contains **1386 RGB images** spanning six defect classes. After discarding frames with no labelled defects, **693 images** remained, each paired with a YOLO-format annotation file (one label file per image). We partitioned this subset into **485 training**, **138 validation** and **70 test** images, preserving the original class distribution. The project uses the following label map, {missing_hole: 0, mouse_bite: 1, open_circuit: 2, short: 3, spur: 4, spurious_copper: 5}.

Table 3.1.2.2 Expansion of the training corpus after augmentation

Metric	**Count**
Original training images	485
Augmented variants generated (18x each)	8730
Total training images after augmentation	9215

3.1.3 System Architecture Diagram

In this section, analysis on the system architecture diagram will be done. More specifically, it will be done in two form, conceptual and technical. One is to explain how those customized modules works in general and one is to explain in actual workflow, how do they excel in PCB defect detection problem domain. In Figure 3.7, visualization on the system architecture diagram was drawn out for LW-YOLOv5.

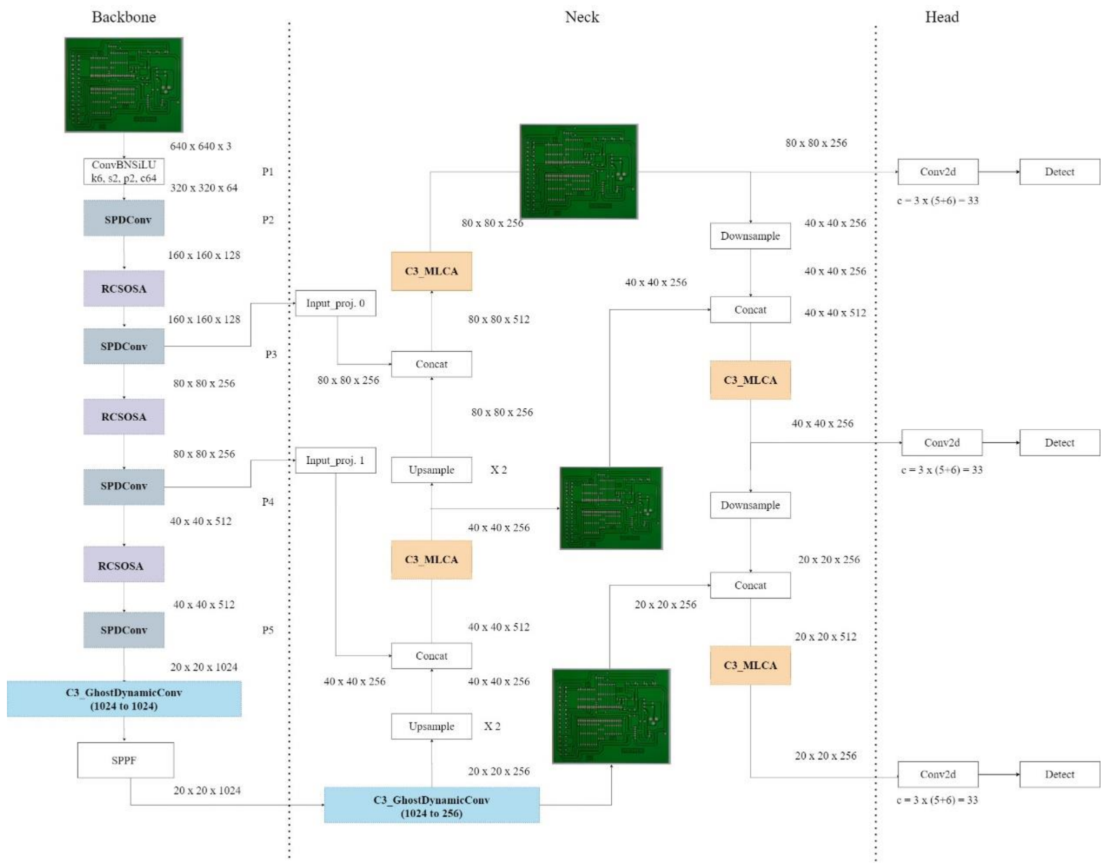


Figure 3.1.3.1 LW-YOLOv5 architecture design in details

Conceptual Perspectives of the LW-YOLOv5 architecture

Significant architectural modifications were introduced to the baseline YOLOv5 framework to enhance detection performance, computational efficiency, and multi-scale feature fusion, particularly for deployment in embedded systems. These improvements include the optimization of anchor boxes through K-Means clustering to better match the distribution of PCB defect sizes, and the replacement of traditional strided convolutional layers with Space-to-Depth Convolution (SPD-Conv) [43], allowing efficient feature downsampling while preserving spatial detail. Further architectural advancements involve the integration of the RCSOSA (Reparameterized Convolution based on Channel Shuffle with One-Shot Aggregation) module [50], promoting lightweight feature reuse and efficient aggregation without increasing inference cost. The Cross-Resolution Fusion Module (CRFM) [48] was additionally incorporated to enhance feature interactions across different spatial scales, strengthening the network's ability to detect objects at varying sizes. To improve feature efficiency and attention, the neck of the architecture was refined by introducing C3-GhostDynamicConv [47] layers for lightweight dynamic convolution and Mixed Local Channel Attention (MLCA) [49] modules to improve local and channel-specific feature focus. Together, these architectural enhancements result in a highly optimized model that achieves excellent detection precision while maintaining a low memory footprint, enabling real-time deployment on resource-constrained embedded platforms.

As for the optimized anchor box, K-means clustering was used to help with the calculation of the refined anchor box compared to the original anchor box that was meant for COCO datasets. Additionally, comparison between K-means clustering and Binary K-means clustering were done for further analysis and study. Ultimately, visualization on the plotting of the center box for K-means clustering had better object localization capabilities which can be showcased in Figure 3.3.2.

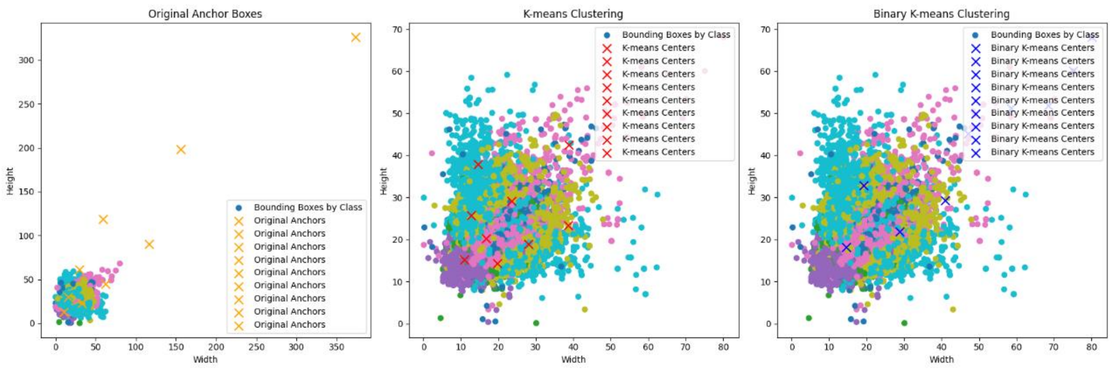


Figure 3.1.3.2 Comparison of calculated anchor box visualized

As Integrating SPD-Conv [43] into the YOLOv5 architecture brings significant improvements, especially for detecting small objects in low-resolution images. Traditional convolutional neural networks often lose critical details when using strided convolutions or pooling layers to downsample features. This loss of fine-grained information can severely impact performance in tasks where precision matters, such as

PCB defect detection. SPD-Conv solves this issue by replacing those layers with a smarter approach that retains all the details while reducing the spatial dimensions.

The SPD-Conv module works in two steps. First, it uses a space-to-depth transformation to rearrange the input feature map, dividing it into smaller submaps and reorganizing them along the channel dimension. This ensures that no information is lost during downsampling. Next, a non-strided convolution processes these features, keeping the resolution intact while extracting important patterns. This combination allows the model to focus on fine details without increasing computational complexity.

By incorporating SPD-Conv into the backbone and head sections of YOLOv5 which make the model downsize by 0.4 million parameter sizes with the integration of RFEM. It is certain that the model becomes lighter and more efficient—perfect for embedded systems where resources are limited. Despite its lightweight nature, the improved architecture achieves high precision, a critical factor for real-time applications. Studies have shown that SPD-Conv enhances accuracy, particularly for small objects, making it an ideal fit for tasks like PCB defect detection.

Overall, the integration of SPD-Conv into LW-YOLOv5 demonstrates how innovative techniques can make deep learning models both powerful and efficient. This improvement strikes a balance between precision and performance, ensuring the model can handle challenging detection tasks while being suitable for deployment in resource-constrained environments.

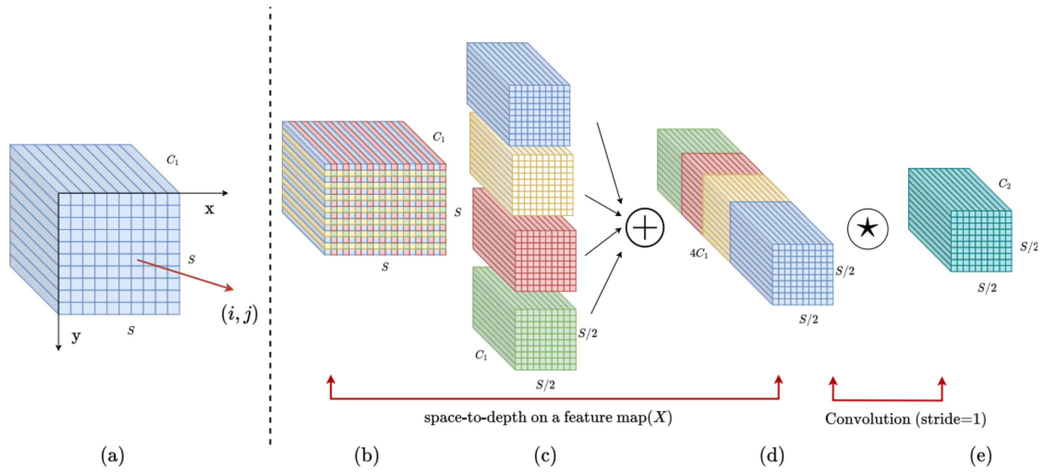


Figure 3.1.3.3 Detail Operation for SPD-Conv [43]

Integrating **C3-GhostDynamicConv** [47] into the YOLOv5 architecture brings significant improvements, especially in addressing the performance limitations of lightweight models under constrained computational budgets. Traditional convolutional neural networks tend to lose critical expressive power when they reduce network depth or width to fit within small memory and processing budgets. This constraint severely limits the network's ability to capture complex feature relationships, particularly for tasks requiring fine-grained precision such as PCB defect detection. C3-GhostDynamicConv solves this problem by dynamically aggregating multiple convolutional kernels based on input-dependent attention, allowing the network to adaptively focus on different features for each input without increasing depth, width, or computational load.

The **C3-GhostDynamicConv module** works in two major steps. First, it applies lightweight parallel convolution operations to generate a set of basic feature maps. Instead of performing heavy convolutions, these initial maps are produced through simple linear transformations. Second, a dynamic aggregation process is applied, where the outputs are selectively combined using input-specific attention mechanisms. This attention-based fusion enables the network to emphasize the most relevant features for each image dynamically, significantly boosting representational power while adding only a minimal computational overhead. The use of Ghost modules ensures that redundancy is minimized, making feature extraction both efficient and adaptable to varying input patterns.

By incorporating **C3-GhostDynamicConv** into the backbone and neck sections of the YOLOv5 architecture, the model achieves improved feature expressiveness and detection robustness while maintaining a compact model size. Despite its lightweight nature, the improved architecture delivers high precision, making it well-suited for real-time PCB defect detection on embedded platforms where resources are limited. Studies have demonstrated that dynamic convolution methods, such as GhostDynamicConv, can provide significant accuracy improvements with only about a 4% increase in

FLOPs, validating its practicality for embedded system deployments C3GhostDynamicConv.

Overall, the integration of **C3-GhostDynamicConv** into LW-YOLOv5 highlights how innovative dynamic feature extraction techniques can make deep learning models both powerful and efficient. This enhancement strikes a critical balance between precision, adaptability, and computational economy, ensuring the model can effectively handle real-world defect detection challenges while remaining suitable for low-power, resource-constrained environments.

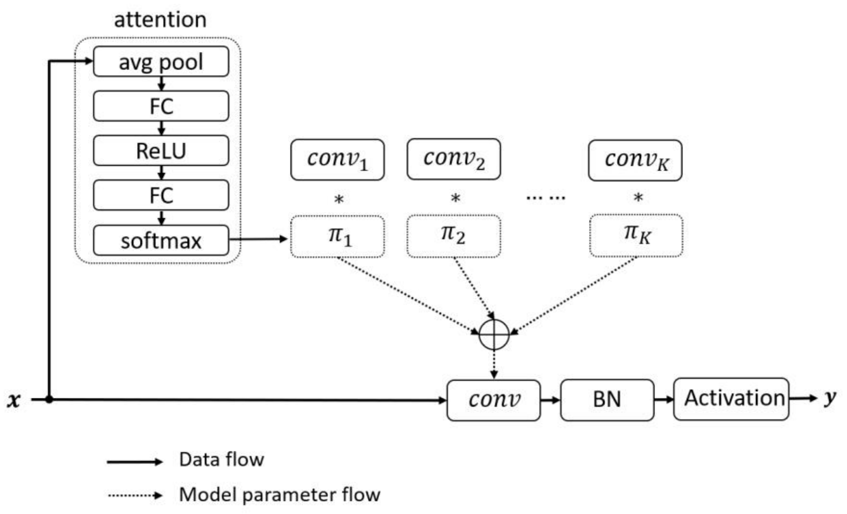


Figure 3.1.3.4 Dynamic Convolution Operation Diagram [47]

Integrating **RCSOSA** (Reparameterized Convolution based on Channel Shuffle with One-Shot Aggregation) [50] into the YOLOv5 architecture introduces substantial theoretical and practical benefits, particularly in optimizing the model's computational efficiency and inference speed RCSOSA. Traditional convolutional networks often rely on complex multi-branch designs to improve feature representation, but these come at the cost of slower inference speeds and increased memory usage, making them unsuitable for embedded or real-time applications. RCSOSA addresses this by combining the advantages of grouped convolutions, channel shuffling, and reparameterization into a lightweight, highly efficient structure that enables rich feature learning during training while collapsing into a simple, fast single-branch form during inference. This dual-stage optimization significantly reduces memory footprint and computational complexity without compromising accuracy, a crucial improvement for tasks such as PCB defect detection that demand both precision and real-time processing capabilities.

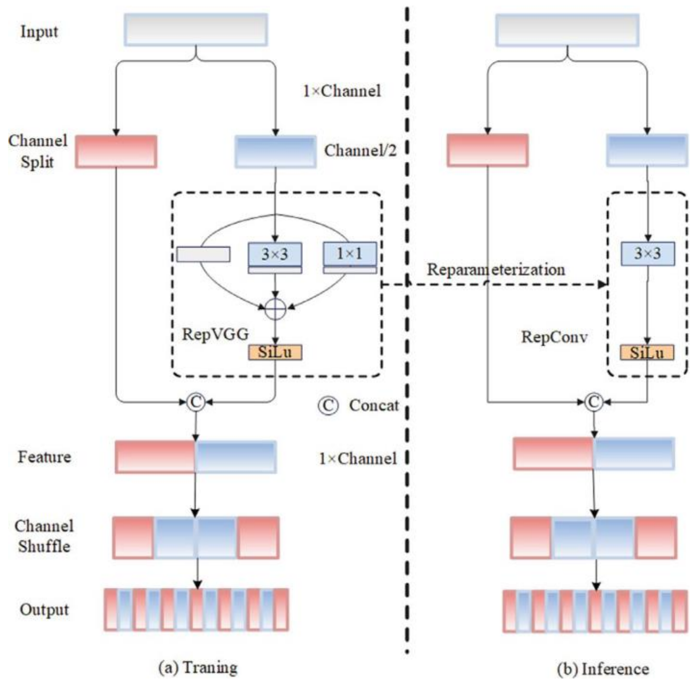


Figure 3.1.3.5 The Structure of RCS [50].

The **RCSOSA module** operates through two main stages. During training, it applies a multi-branch structure where feature maps are split, transformed through lightweight 1×1 and 3×3 convolutions, and re-combined with a channel shuffle operation to promote inter-group information flow. This setup enhances feature diversity and depth without substantial computational cost. At the inference stage, structural reparameterization is employed, merging these multiple branches into a single 3×3 convolution, thus dramatically speeding up computation and simplifying the network architecture. By maintaining rich feature interaction during learning and minimizing overhead during deployment, RCSOSA achieves an ideal balance between accuracy and efficiency.

By integrating **RCSOSA** modules into both the backbone and neck of LW-YOLOv5, the model effectively shortens the information path between layers, accelerates feature propagation, and enhances semantic information aggregation across scales. Experimental results have demonstrated that networks employing RCSOSA experience up to **50% reduction in FLOPs** compared to traditional dense architectures RCSOSA, while achieving comparable or superior precision. In practical terms, this means faster inference speeds and lower energy consumption, critical factors for real-time embedded applications where processing power and battery life are limited. For PCB defect detection, where detecting tiny defects quickly and accurately is essential, RCSOSA ensures that the model remains lightweight without sacrificing detection robustness.

Overall, the integration of **RCSOSA** into LW-YOLOv5 demonstrates how architectural innovations centered on reparameterization and channel operations can produce models that are both highly efficient and capable of maintaining strong detection performance. This improvement reinforces the ability of the model to operate reliably under tight resource constraints, making it highly suitable for practical deployment in embedded industrial inspection systems.

Integrating the **Mixed Local Channel Attention (MLCA)** [49] module into the YOLOv5 architecture provides significant improvements in balancing model performance, computational complexity, and memory efficiency, particularly for lightweight object detection tasksMLCA. Traditional attention mechanisms often focus either solely on channel attention or on spatial attention, but few efficiently capture both local and global information while keeping computational overhead low. Moreover, most existing attention mechanisms, such as SE and CBAM, introduce complexity and parameter inflation, which are impractical for real-time embedded systems. MLCA addresses these challenges by simultaneously incorporating channel, spatial, local, and global feature information in a lightweight manner, thereby significantly enhancing the network’s expressive capability without substantially increasing the computational cost. This combination is particularly critical for PCB defect detection, where subtle and localized features must be effectively captured under strict resource constraints.

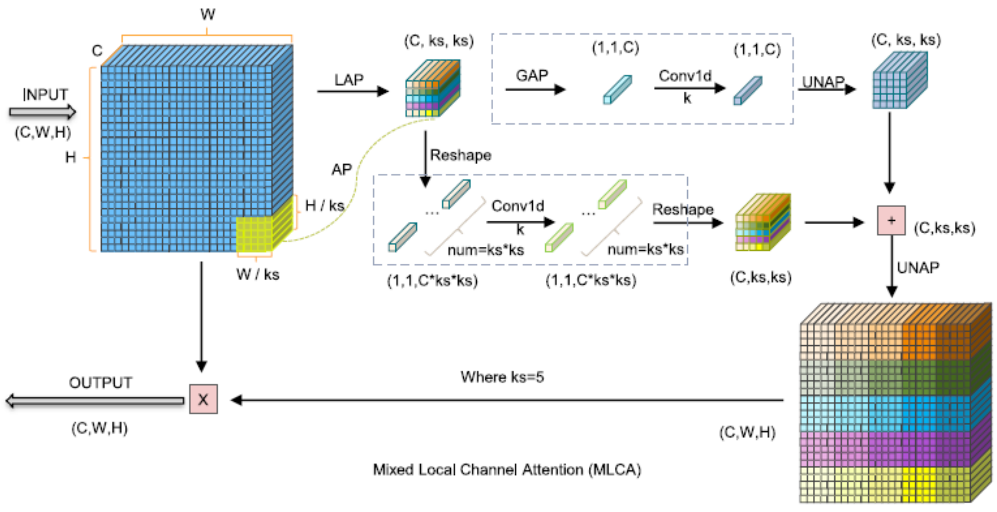


Figure 3.1.3.6 The Principle of MLCA algorithm [49]

The **MLCA module** [49] operates through a two-stage mechanism. Initially, a local spatial feature representation is obtained by applying local average pooling (LAP) to divide the feature maps into multiple patches. Subsequently, two parallel branches are created: one extracts global information through global average pooling (GAP), while the other retains localized information through the extracted patches. Both branches are then processed with lightweight one-dimensional convolutions to capture intra-channel dependencies while maintaining locality. Afterward, an anti-pooling (UNAP) operation restores the original feature map size, and the outputs from the two branches are fused, combining both local and global contextual information. The entire operation is lightweight, with the number of parameters and GFLOPs remaining comparable to traditional SE attention modules yet offering richer feature representations.

By embedding **MLCA** into the neck section of LW-YOLOv5, the model achieves enhanced spatial awareness and channel sensitivity, leading to better defect localization and classification accuracy. Experimental results have shown that integrating MLCA into object detection networks improves mAP performance by 1.0–1.5% over SE and CA modules while maintaining similar inference speedsMLCA. This makes it highly suitable for real-time industrial applications where both high precision and low latency are critical. Particularly in the context of PCB defect detection, where tiny and irregular defects need to be captured reliably, MLCA enables the lightweight model to maintain robust detection performance without sacrificing computational efficiency.

Overall, the incorporation of **MLCA** into LW-YOLOv5 demonstrates how a carefully designed attention mechanism can significantly improve the model's ability to capture essential features while preserving the low complexity required for embedded deployment. This integration effectively bridges the gap between theory and practical application, allowing lightweight object detection models to perform with near heavy-weight accuracy while operating under constrained hardware environments.

Integrating the **Cross-Resolution Fusion Module (CRFM)** [48] into the YOLOv5 architecture offers significant improvements in multi-scale feature fusion and enhances the network's ability to detect objects across varying scalesCRFM structure. Traditional convolutional neural networks, particularly one-stage detectors like YOLOv5, often struggle to effectively combine low-level fine-grained features with high-level semantic-rich features. This limitation weakens the detection of small and subtle objects, a crucial requirement in tasks like PCB defect detection. CRFM addresses this challenge by decoupling the intra-scale feature interaction and cross-scale feature fusion processes, allowing each to be optimized independently. By applying lightweight Transformer-based intra-scale interaction only to high-level features and convolution-based fusion across different scales, CRFM significantly reduces computational redundancy while enhancing multi-scale feature aggregation. This dual-path strategy ensures that both detailed spatial information and semantic context are preserved, resulting in more robust and accurate object detection under strict resource constraints.

The CRFM structure operates through two core mechanisms. First, **Attention-based Intra-scale Feature Interaction (AIFI)** applies single-scale Transformer encoders specifically to high-level feature maps, capturing global semantic relationships without incurring the heavy computation cost of multi-scale attention. Second, the **CNN-based Cross-scale Feature Fusion (CCFF)** module fuses features across different resolutions using lightweight convolutional operations, ensuring effective information flow between shallow and deep layers. The fusion process is performed through a series of RepBlocks, where feature maps from adjacent scales are merged after channel adjustment and feature refinement, preserving critical details while keeping the model compact. This separation of intra-scale and cross-scale operations enables the model to maximize feature richness while minimizing latency, a balance crucial for real-time embedded applications.

By incorporating **CRFM** into the backbone and neck sections of LW-YOLOv5, the model achieves superior multi-scale feature integration, boosting the detection performance for objects of different sizes without compromising inference speed. Empirical results from RT-DETR experiments show that decoupling feature interactions in this way reduces encoder latency by up to **35%** while still improving overall AP by 0.4%CRFM structure, confirming CRFM's practical effectiveness. For PCB defect detection, where defects can vary significantly in size and visibility, CRFM empowers the network to maintain high detection accuracy across all scales, ensuring consistent performance even in challenging conditions.

Overall, the integration of **CRFM** into LW-YOLOv5 highlights the importance of optimizing both intra-scale and cross-scale feature processing. By strategically balancing attention mechanisms and convolutional fusion, CRFM enables lightweight detectors to achieve robust multi-scale detection capabilities while adhering to the tight computational budgets required for embedded deployment.

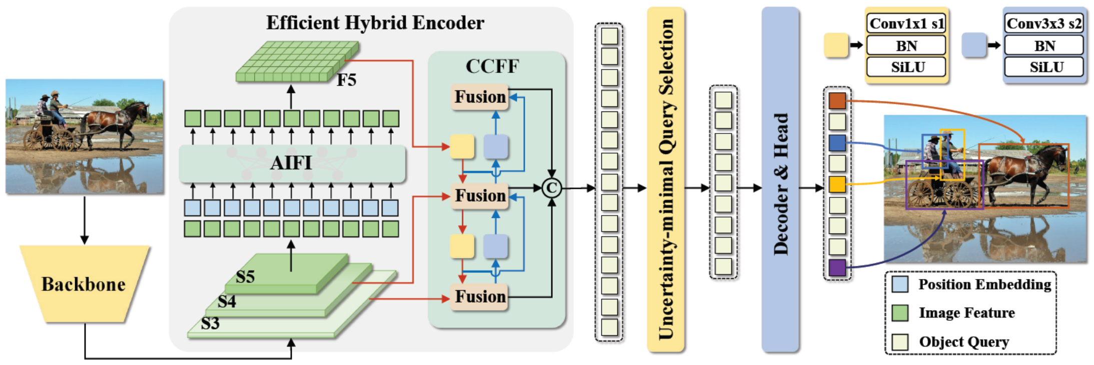


Figure 3.1.3.7 Overview of RT-DETR Encoder (with AIFI + CCFF modules) [48]

Technical Perspective of LW-YOLOv5's customized modules

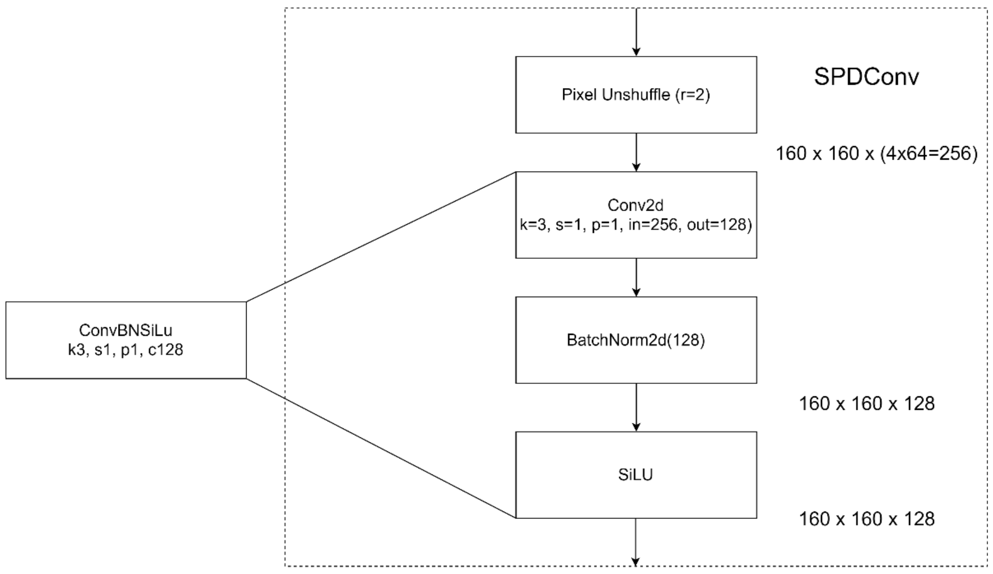


Figure 3.1.3.8 SPDConv's internal operations

The **SPDConv** class first reshapes the input tensor by concatenating the four spatial quadrants along the channel axis, effectively multiplying the channel dimension by four while halving both height and width. The subsequent nn.Conv2d then operates on this enlarged channel set, with autopad providing "same-shape" padding even when dilation is present. A batch-normalisation layer and a SiLU activation complete the block. Because the pixel-unshuffle occurs on the fly inside forward, only a single CUDA kernel is launched for the entire operation after graph fusion, keeping latency low on Jetson Orin Nano .

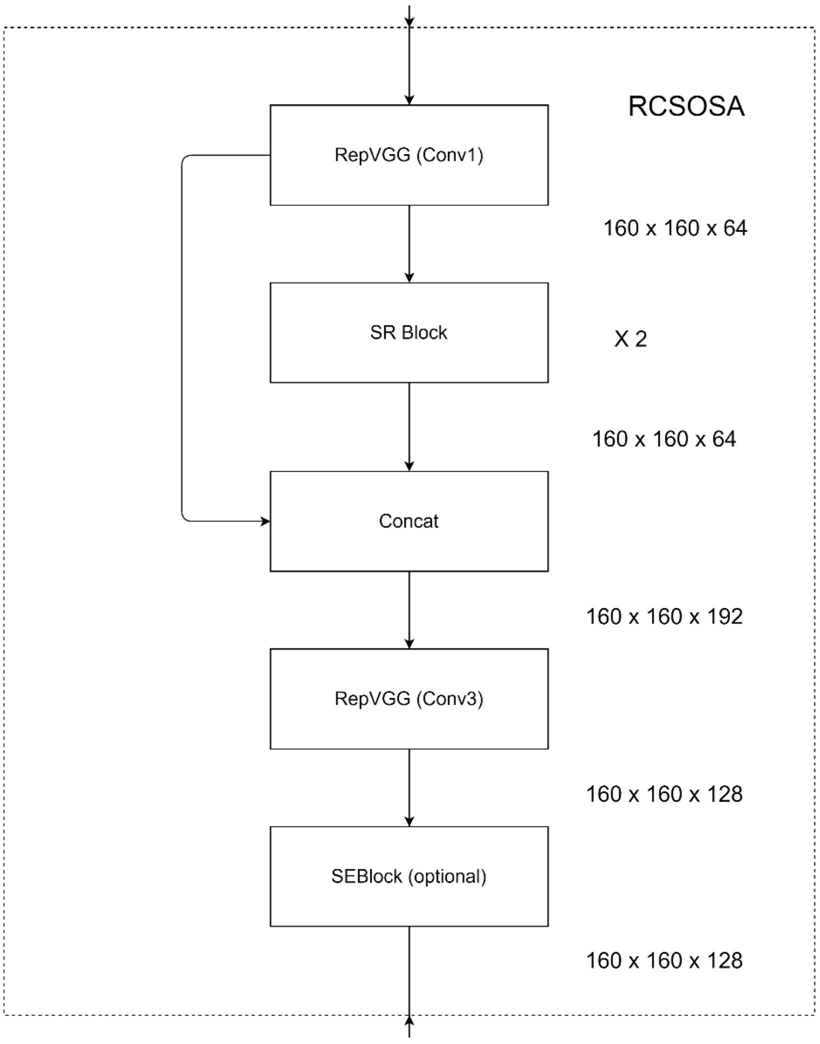


Figure 3.1.3.9 RCSOSA Block's internal operations

The **RepVGG** class provides the re-parameterisable convolution used in several blocks. During training three parallel branches, 3 × 3, 1 × 1 and identity, each hold their own batch-normalisation statistics. The helper get_equivalent_kernel_bias fuses these branches into a single kernel–bias pair, enabling inference to proceed with one plain convolution. The **SR** (Shuffle RepVGG) micro-block exploits this property by splitting the incoming tensor, applying RepVGG to one half, concatenating, and finally shuffling channels, a light-weight alternative to full ShuffleNet mixing. Two such SR units constitute the internal body of **RCSOSA**, which concatenates three parallel streams (conv1, sr1, sr2) before a final RepVGG collapses them to the output width. An optional squeeze-and-excitation gate may be appended for additional channel recalibration.

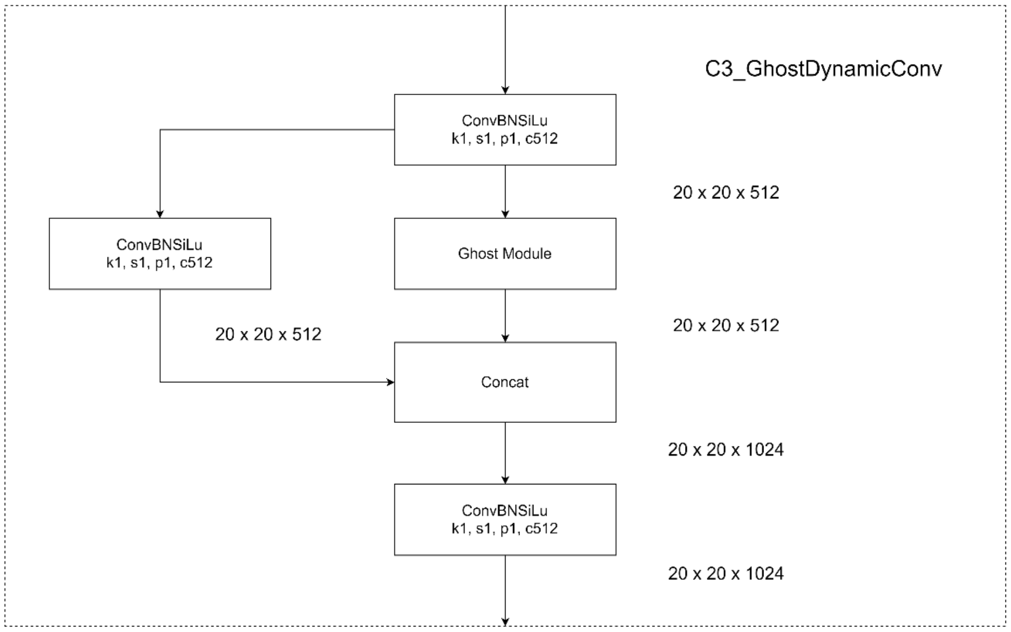


Figure 3.1.3.10 C3-GhostDynamicConv's internal operations

Bottleneck_DynamicConv, C3_DynamicConv, and **C3_GhostDynamicConv** reuse the standard YOLOv5 CSP scaffold but substitute the inner convolutions with either DynamicConv or GhostModule instances. In C3_GhostDynamicConv, the iterable self.m holds a stack of GhostModule layers, giving the block a depth identical to its vanilla counterpart yet reducing multiply adds by roughly one-half Customized Modules.

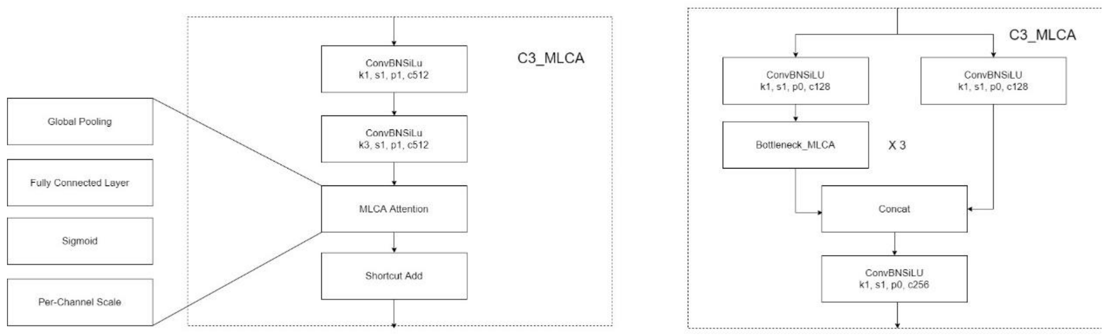

Global Pooling
Fully Connected Layer
Sigmoid
Per-Channel Scale
C3_MLCA
ConvBNSiLu k1, s1, p1, c512
ConvBNSiLu k3, s1, p1, c512
MLCA Attention
Shortcut Add
C3_MLCA
ConvBNSiLU k1, s1, p0, c128
ConvBNSiLU k1, s1, p0, c128
Bottleneck_MLCA
X 3
Concat
ConvBNSiLU k1, s1, p0, c256

Figure 3.1.3.11 MLCA [49]'s internal operations

The **MLCA** attention unit combines local and global descriptors. A fixed-size adaptive-average pool (local_size = 5 by default) extracts a patch-level summary, while a separate global pool supplies coarse context. Two independent 1-D convolutions as one for each descriptor to learn scale coefficients; the outputs are interpolated back to the original spatial size and merged by a weighted average (local_weight). Elementwise multiplication with the incoming tensor finalises the attention process. Kernel size is chosen dynamically from the channel count via a log-scaled rule (gamma and b hyper-parameters), echoing the original ECA formulation Customized ModulesCustomized Modules.

Technical perspective for LW-YOLOv5

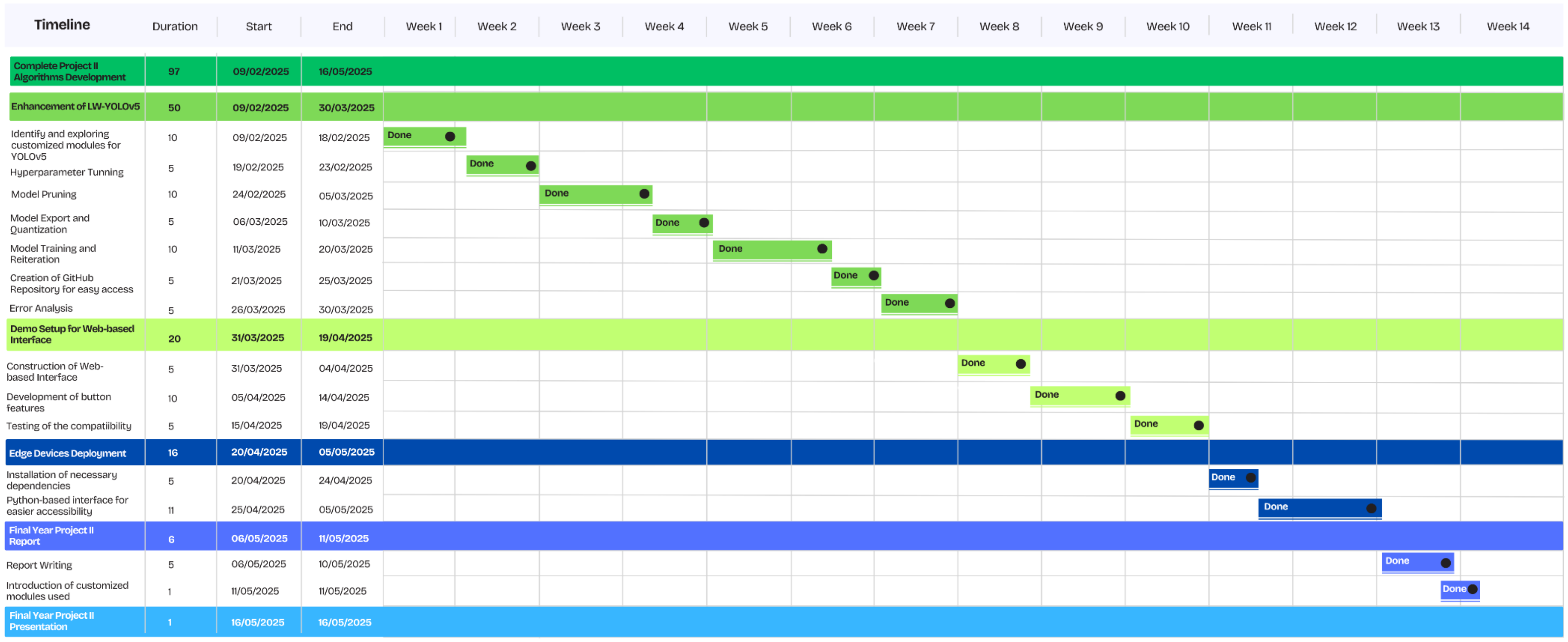
The modified architecture begins with a 6 × 6, stride-2 convolutional stem that immediately reduces the 160 × 160 input to 80 × 80 while extracting low-level edge information. A sequence of **SPDConv** and **RCSOSA** units then alternates down the backbone. Each SPDConv rearranges spatial neighbourhoods into the channel dimension by pixel-unshuffle before applying a 3 × 3 convolution, achieving resolution halving with minimal arithmetic cost. Each RCSOSA block supplements this compression with residual channel–spatial attention implemented through re-parameterised RepVGG kernels and lightweight shuffle-mixing, enriching the feature map without introducing runtime branches. After three such SPDConv–RCSOSA pairs, the tensor reaches 20 × 20 and 1 024 channels.

At this depth the network inserts a **C3_GhostDynamicConv** module that fuses cross-stage-partial topology, dynamic pointwise kernels, and Ghost feature generation. The design halves floating-point operations relative to a vanilla C3 while gaining adaptability through conditionally generated weights, and it serves as the principal semantic extractor before the spatial pyramid pooling-fast (**SPPF**) block. SPPF widens the receptive field by concatenating three max-pooled paths of increasing kernel size, providing multi-scale context for subsequent fusion.

The neck adopts a bidirectional feature-pyramid pattern. A 1 × 1 projection (labelled *Input_proj 0*) lifts the deepest 20 × 20 feature map into the lateral pathway, where it merges through concatenation with an up-sampled 40 × 40 tensor. Each merge site is wrapped in a **C3_MLCA** unit, whose embedded multi-layer channel-attention recalibrates activations using both local and global descriptors, thereby emphasising fine solder-mask artefacts across scales. Down-sampling operations then propagate information back to coarser resolutions, again passing through C3_MLCA to maintain channel sensitivity.

A second **C3_GhostDynamicConv** (configured 1024→256) prepares the 80 × 80 feature map for shallow detection by compressing channels without forfeiting representational power. Finally, three parallel *Conv2d → Detect* heads, operating at strides 8, 16, and 32, output class probabilities and bounding-box regressions. The entire graph remains free of custom CUDA kernels, enabling seamless export to ONNX and efficient optimisation by TensorRT for deployment on the Jetson Orin Nano.

3.2 Gantt Chart



Timeline	Duration	Start	End
Complete Project II Algorithms Development	**97**	**09/02/2025**	**16/05/2025**
Enhancement of LW-YOLOv5	**50**	**09/02/2025**	**30/03/2025**
Identify and exploring customized modules for YOLOv5	10	09/02/2025	18/02/2025
Hyperparameter Tunning	5	19/02/2025	23/02/2025
Model Pruning	10	24/02/2025	05/03/2025
Model Export and Quantization	5	06/03/2025	10/03/2025
Model Training and Reiteration	10	11/03/2025	20/03/2025
Creation of GitHub Repository for easy access	5	21/03/2025	25/03/2025
Error Analysis	5	26/03/2025	30/03/2025
Demo Setup for Web-based Interface	**20**	**31/03/2025**	**19/04/2025**
Construction of Web-based Interface	5	31/03/2025	04/04/2025
Development of button features	10	05/04/2025	14/04/2025
Testing of the compatiibility	5	15/04/2025	19/04/2025
Edge Devices Deployment	**16**	**20/04/2025**	**05/05/2025**
Installation of necessary dependencies	5	20/04/2025	24/04/2025
Python-based interface for easier accessibility	11	25/04/2025	05/05/2025
Final Year Project II Report	**6**	**06/05/2025**	**11/05/2025**
Report Writing	5	06/05/2025	10/05/2025
Introduction of customized modules used	1	11/05/2025	11/05/2025
Final Year Project II Presentation	**1**	**16/05/2025**	**16/05/2025**

Figure 3.2.1 Gantt Chart

Chapter 4

System Design

This section highlights the foundational technologies and tools employed during the development of the proposed lightweight PCB defect detection model. Key hardware components include a high-performance laptop utilized for training and testing the YOLOv5-based model, ensuring sufficient computational power during the initial development phase, and a microcontroller serving as the final deployment platform to validate the model's performance in a resource-constrained environment. On the software side, critical tools such as Python and Google Colab were integral for coding, model training, and experimentation. Additionally, the study incorporates the use of data preprocessing and augmentation techniques to enhance the datasets. A structured workflow has been designed, supported by a block diagram, to offer clarity and direction throughout the implementation process. These preliminary steps provide the groundwork for achieving an efficient and deployable defect detection system.

4.1 System Block Diagram

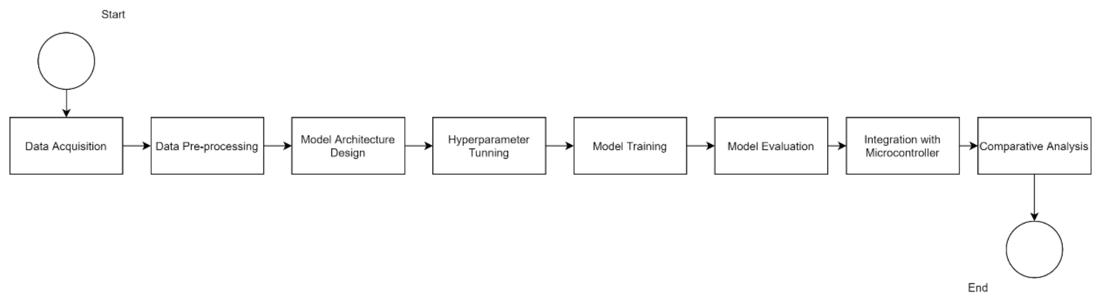


Figure 4.1.1 General Block Diagram of Proposed System

As for the description for our general block diagram, our proposed methods consist of 8 general phases in total, which range from data acquisition, data pre-processing, model architecture design, hyperparameter tunning, model training, evaluation of model, integration with microcontroller and comparative analysis as the final phase.

4.2 System Components Specifications

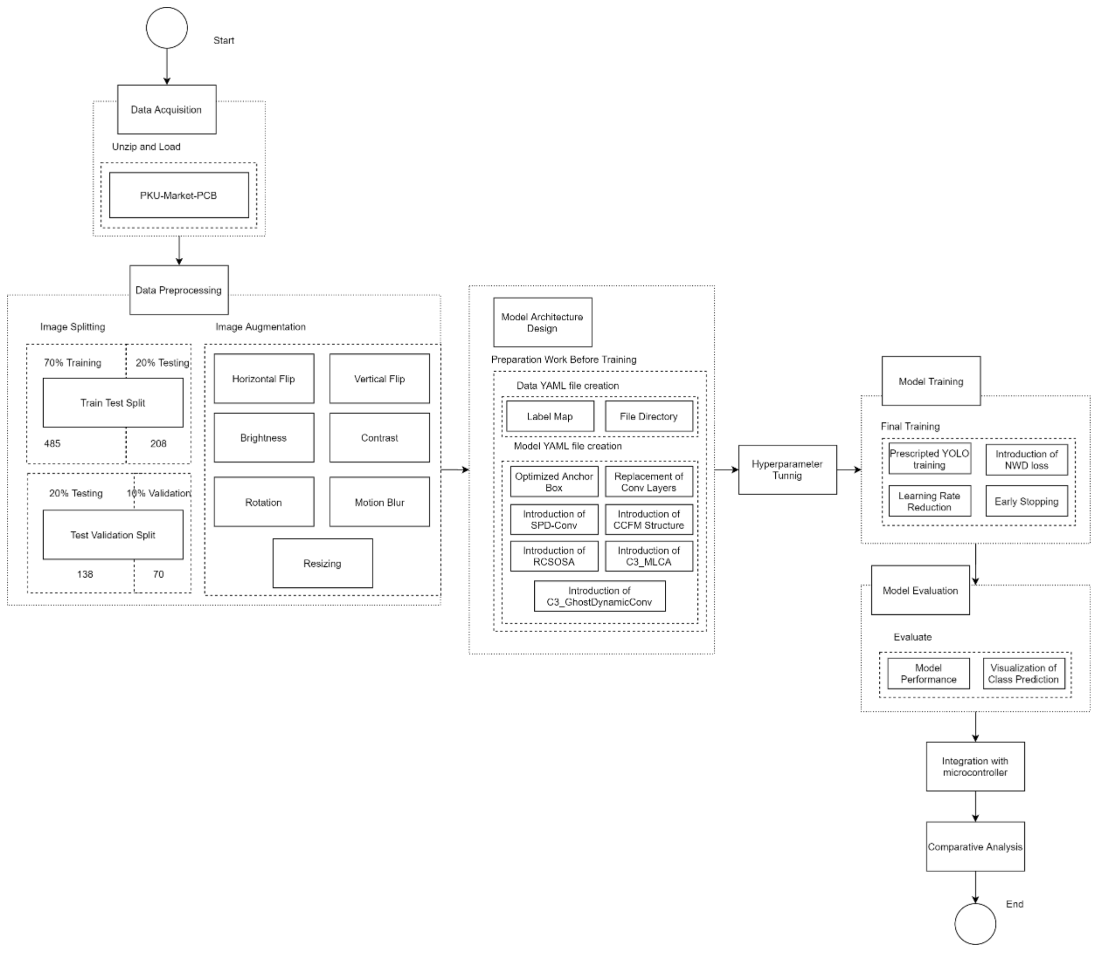


Figure 4.2.1 Block Diagram with break down workflow

This block diagram provides a detailed explanation of the lightweight PCB defect detection and object detection workflow, highlighting each step from data acquisition to model evaluation and integration. The goal is to build a lightweight, accurate, and efficient system that can be deployed on embedded devices for real-world applications.

Image Acquisition

Initially, the data acquisition phase begins by importing datasets, which include **PKU-Market-PCB** [15]. These datasets serve distinct purposes, with PKU-Market-PCB [15] providing small defect-specific images for PCB detection to enhance model generalization. This specific dataset must be correctly extracted and their directories inputted to ensure accessibility. If the system cannot locate or load the datasets, an error message is displayed, requiring user intervention. Once the datasets are successfully loaded, the workflow proceeds to image preprocessing.

Image Preprocessing

Image preprocessing prepares the datasets for efficient and effective model training. This phase begins with image splitting, where the data is divided into three subsets: **70% for training**, **20% for testing**, and **10% for validation**. For instance, out of **693 total images**, **485 are allocated** for **training**, **208** for **validation**, and **70** for **testing**. This structured division ensures that the model is trained, validated, and tested comprehensively to evaluate its performance reliably.

Following the split, image augmentation techniques are applied to simulate diverse real-world conditions and improve model generalization. Augmentation methods include horizontal and vertical flipping, brightness and contrast adjustments, rotation, motion blur, and resizing. These transformations create a richer dataset, allowing the model to learn from variations in lighting, orientations, and motion, ultimately boosting its robustness to different scenarios.

Model Architecture Design

This section focuses on constructing a lightweight detector suitable for embedded-GPU deployment. Preparation tasks include generating a data YAML file that lists class labels and directory roots, and a model YAML file that specifies the layer topology. Conventional convolutional layers are replaced by efficiency-oriented counterparts: **SPDConv** [43] for resolution reduction, **RCSOSA** [50] for residual channel-spatial attention, **C3_GhostDynamicConv** [47] for conditional kernel inference with ghost feature maps, and **C3_MLCA** [49] for multi-layer channel attention inside the feature pyramid. Anchor boxes are re-optimised to reflect object-size statistics in the PCB dataset, and the **Cross-Resolution Fusion Module (CRFM)** [48] is retained to preserve gradient flow while limiting parameter growth.

Hyperparameter Tunning

To ensure a fair and balanced comparative analysis with the reference ARMA-based YOLO [46], hyper-parameter tuning was conducted under carefully controlled conditions. Here are the hyperparameter settings that were setup in Table 4.X.

Table 4.2.1 Overviews of Hyperparameter Settings

Hyperparameter	Value	Purpose

Image Size (--img)	640	Improves detection of small objects while maintaining computational efficiency, essential for detecting PCB defects.
Batch Size (--batch)	24	Optimizes memory usage and allows faster convergence during training, ensuring the model processes adequate data per iteration.
Epochs (--epochs)	100	Provides sufficient training time for the model to learn complex patterns specific to PCB defect detection.
Optimizer (--optimizer)	AdamW	Offers stable and adaptive optimization suitable for lightweight models, helping achieve efficient convergence and better generalization.
Initial Learning Rate	0.01	Allows the model to make substantial updates to weights in early training stages, ensuring rapid initial learning.
Learning Rate Decay	0.0005	Gradually reduces the learning rate, enabling finer adjustments to weights during later training stages for improved accuracy.
Cache (--cache)	Enable	Reduces data loading times significantly, accelerating training and making efficient use of available resources.
Patience (--patience)	20	Prevents unnecessary overtraining by stopping early if validation performance does not improve, preserving resources and avoiding overfitting.

Model Training

The model training proceeds with the tuned hyper-parameters and the recommended YOLO optimisation strategy. **Normalised Wasserstein Distance (NWD)** [44] is introduced as the bounding-box regression loss, replacing Intersection-over-Union to stabilise optimisation on small targets. A cosine learning-rate reduction schedule encourages smooth convergence, and early stopping criteria monitor validation loss to prevent over-fitting. Checkpoints are retained at each plateau, with the final weights selected according to highest validation mean Average Precision (mAP).

The final training process incorporates prescribed YOLO settings, **introduction of Normalized Wasserstein Distance (NWD)** [44], including **learning rate reduction** to optimize convergence and **early stopping** to avoid overfitting. Once training is complete, the model undergoes a rigorous evaluation using metrics such as **mean Average Precision (Map)** and classification accuracy.

Evaluating a model's performance is critical, especially when detecting tiny objects in complex environments like PCB defect detection. Traditional evaluation metrics like Intersection over Union (IoU) often fall short when it comes to handling small objects, as they are highly sensitive to scale and minor positional changes. To address these shortcomings, this study integrates the **Normalized Wasserstein Distance (NWD)** [44] as a key component of the model evaluation process.

NWD represents a shift in how bounding box similarity is measured. Instead of directly comparing the overlap between bounding boxes, NWD uses a mathematical approach based on optimal transport theory. This involves modeling bounding boxes as two-dimensional Gaussian distributions, which capture both the position and size of the object with greater precision. Unlike IoU, which treats all parts of a bounding box equally, NWD emphasizes the core areas of the box while smoothly reducing the influence of its edges. This ensures that even subtle overlaps or displacements are accurately reflected in the evaluation.

A standout feature of NWD is its ability to handle objects of varying scales effectively. For tiny objects, which are often just a few pixels in size, IoU tends to penalize models heavily for minor errors in positioning. NWD, however, accounts for these variations with grace, ensuring a more balanced evaluation. By incorporating

Gaussian modeling, NWD also introduces scale invariance, allowing models to detect both small and large objects with equal proficiency.

Beyond simply measuring detection accuracy, NWD proves valuable in the overall detection pipeline. It replaces IoU in processes like non-maximum suppression (NMS), reducing redundant predictions while maintaining accuracy for overlapping objects. Moreover, its smooth sensitivity to positional deviations ensures better training outcomes, leading to improved regression performance for bounding box predictions.

The inclusion of NWD in this study has demonstrated its ability to enhance model evaluation significantly, especially in scenarios involving small object detection. It not only provides a clearer picture of a model's performance but also helps improve the underlying processes, ensuring that tiny defects are identified with precision and reliability. By addressing the limitations of traditional metrics, NWD contributes to a more robust and efficient evaluation framework.

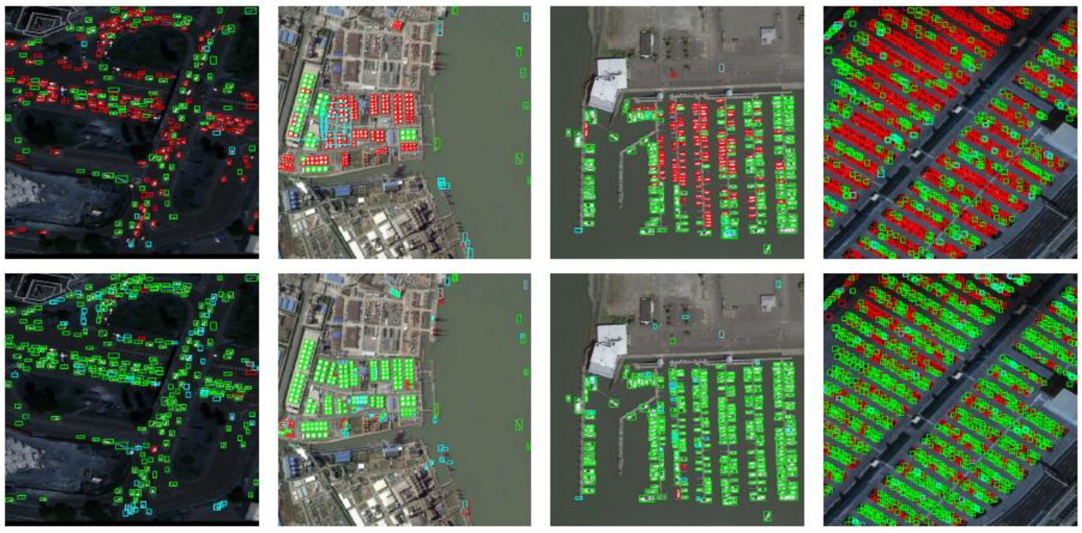


Figure 4.2.2 Differences between IoU-based detector (first row) and NWD-based detector (second row). Green, blue and red indicating true positive (TP), false positive (FP) and false negative (FN) respectively [44]

Model Evaluation

The model evaluation phase assesses the performance of the developed model in terms of accuracy, efficiency, and robustness to general unseen data. Visualization tools are used to analyze class predictions, allowing researchers to assess the model's detection capabilities visually. These evaluations ensure that the model meets the desired performance criteria, validating its suitability for deployment. If the evaluation results are unsatisfactory, adjustments can be made to further refine the model before proceeding.

Integration with Microcontroller

The Integration with the **Nvidia Jetson Orin Nano** [51] targeted only the two canonical artefacts produced after training— best.pt (native PyTorch checkpoint) and its direct export best.onnx. No attempt was made to compile a TensorRT engine, because the additional graph fusion, kernel specialisation, and calibration stages that TensorRT imposes would exceed the flash-storage budget and build-time tooling available on the target micro-controller. Instead, the .onnx model is executed via a lightweight ONNX-Runtime micro back-end, while the .pt file serves as an editable reference for future pruning or quantisation experiments. This restrained deployment path keeps the software stack minimal, avoids proprietary dependencies, and honours the memory-complexity constraints that define the embedded setting.

Comparative Analysis

The final step in the workflow involves benchmarking the proposed system against existing state-of-the-art models. Also, ablation experiment was conducted to make sure all results of taking off and on the components of the developed model can be visualized and quantified. This phase compares performance metrics such as accuracy, speed, and resource efficiency, highlighting the advantages of the proposed system. The insights gained from this analysis not only validate the effectiveness of the lightweight model but also identify areas for future research and optimization, ensuring continuous improvement.

Chapter 5

System Implementation

5.1 Hardware Setup

The hardware involved in this project involve a high-performance laptop and a microcontroller. As previously mentioned, both played crucial roles in the development and deployment of optimizing deep learning algorithms for manufacturing application, specifically for PCB defect detection.

Figure 5.1.1 Nvidia Jetson Orin Nano on Setup

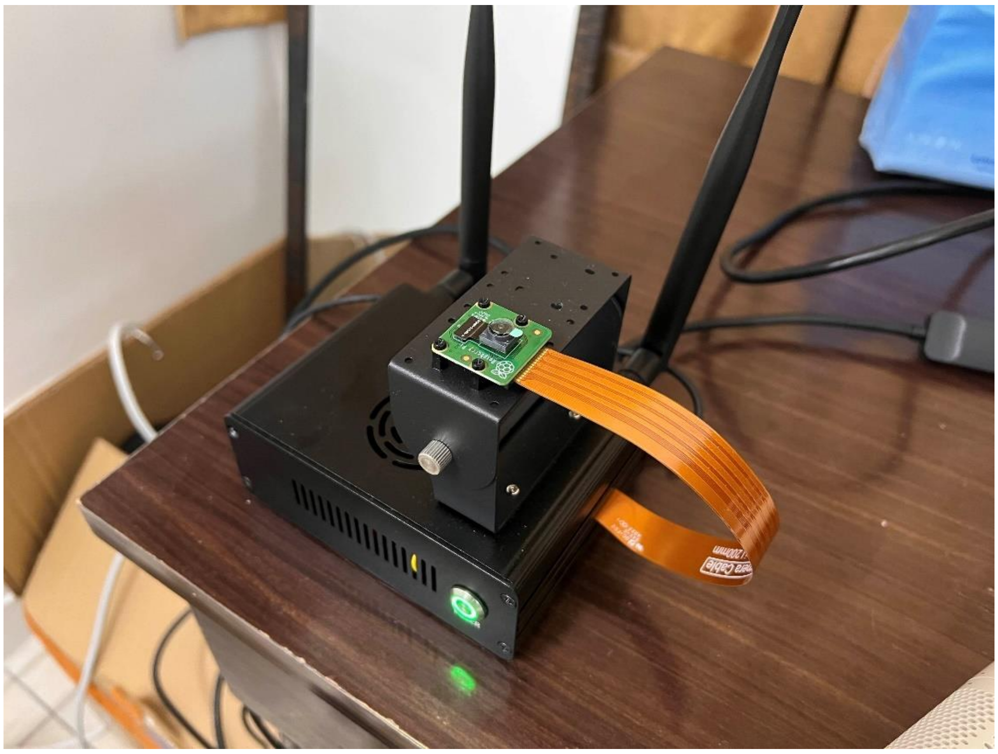


Table 5.1.1 Specifications of laptop [45]

Description	Specifications
Model	Asus ROG Strix G15 G513 [47]
Processor	AMD Ryzen 7 5800H
Operating System	Windows 11
Graphic	NVIDIA GeForce RTX3050 4GB GDDR6
Memory	32GB DDR4 3200Mhz RAM

Storage	1TB Samsung SSD

Table 5.1.2 Specifications of the microcontroller [51]

Description	**Specifications**
Model	Nvidia Jetson Orin Nano Super Developer Kit
GPU	NVIDIA Ampere™ architecture • 1 024 CUDA cores • 32 Tensor cores
CPU	6-core Arm Cortex-A78AE v8.2 64-bit (1.5 MB L2 + 4 MB L3 cache)
System Memory	8 GB LPDDR5 (128-bit) • 102 GB/s in 15–25 W mode • 68 GB/s in 7–15 W mode
External Storage	• microSD slot (UHS-I up to SDR104) • NVMe SSD via M.2 Key-M (PCIe Gen 3 ×4 or ×2)
Power Envelope	Configurable 7 W – 25 W
Camera Interfaces	2 × 22-pin MIPI CSI-2 connectors
M.2 Key-M	PCIe Gen 3: ×4 or ×2 lanes (selectable)
M.2 Key-E	PCIe ×1 + USB 2.0 + UART + I²S + I²C • Pre-installed 802.11ac Wi-Fi / Bluetooth 5.0 card
USB	4 × USB-A 3.2 Gen 2 • 1 × USB-C (debug / device-mode only)
Networking	1 × Gigabit Ethernet (RJ-45)
Display Output	DisplayPort 1.2 with MST
Expansion Headers	40-pin general-purpose (UART, SPI, I²S, I²C, GPIO) • 12-pin button header • 4-pin fan header
Power Input	DC barrel jack
Compatibility	Accepts Jetson Orin NX module
Physical Size	

	100 mm × 79 mm × 21 mm (board + module + thermal solution, incl. feet)

5.2 Software Setup

In the software section, Python and Jupyter Notebook had been utilized as development and programming tools. Besides, usage of TensorFlow, PyTorch and Keras, were specifically for the machine learning framework. Their purpose is to develop, train and optimize deep learning models. Moreover, we utilized TensorFlow Lite, MCUXpresso IDE and MCUXpresso SDK as our deployment tools particularly for the embedded devices.

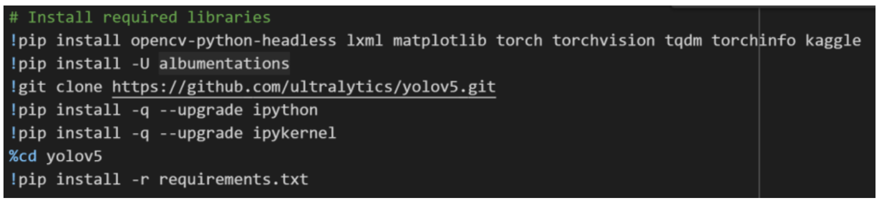
Table 5.2.1 Specifications of software used

Categories	**Software Used**
Development and Programming Tools	Python
	Jupyter Notebook, Kaggle, Google Colab
Machine Learning Frameworks	TensorFlow, Keras
	PyTorch
Deployment Tools	ONNX runtime, Ubuntu 22.04 LTS
	Open-cv Python, Torchscript, PyTorch

5.3 Setting and Configuration

5.3.1 Setup for Libraries in Google Colab

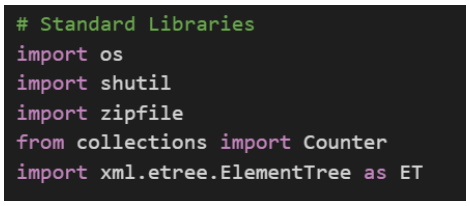
In this section of work done, required libraries are split into categories for easier visualization of each importation. Also, installation and update procedure are done before the entire work process starts.



```
# Install required libraries
!pip install opencv-python-headless lxml matplotlib torch torchvision tqdm torchinfo kaggle
!pip install -U albumentations
!git clone https://github.com/ultralytics/yolov5.git
!pip install -q --upgrade ipython
!pip install -q --upgrade ipykernel
%cd yolov5
!pip install -r requirements.txt
```

Figure 5.3.1.1 Installation for required libraries

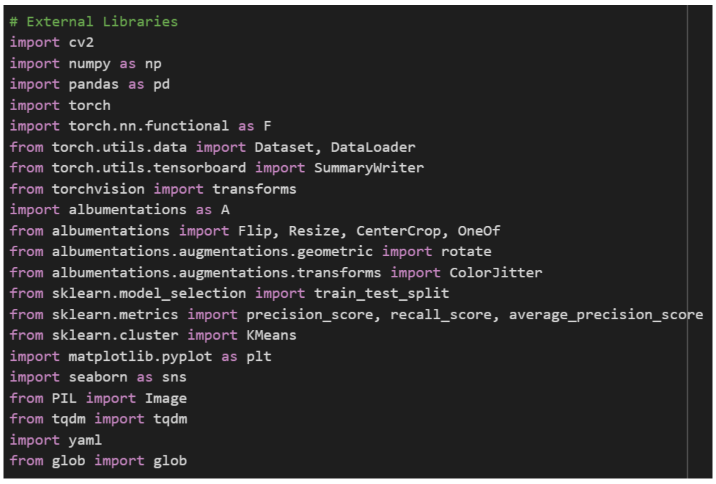
This setup installs the necessary libraries and tools to run the YOLOv5 framework for object detection. It begins by installing essential Python packages such as opencv-python-headless, torch, and torchvision for image processing and deep learning tasks. The albumentations library is added to support advanced image augmentations. The YOLOv5 repository is cloned from GitHub, containing the core model and utilities, and the environment dependencies are finalized by installing packages from the requirements.txt file. These steps ensure the environment is fully prepared for training and deploying YOLOv5 models.



```python
# Standard Libraries
import os
import shutil
import zipfile
from collections import Counter
import xml.etree.ElementTree as ET
```

Figure 5.3.1.2 Standard Libraries

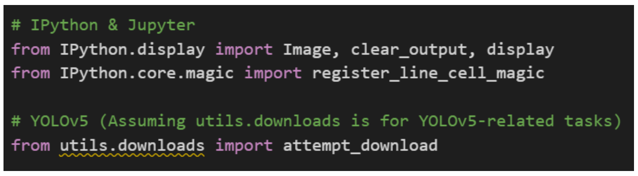
This section of imports includes Python's built-in libraries such as os, shutil, zipfile, and others, which are primarily used for file handling, directory management, and annotation parsing. For example, xml.etree.ElementTree (ET) is used to read XML files. The result of this setup is an organized file system and efficient handling of dataset metadata for preprocessing and training.



```python
# External Libraries
import cv2
import numpy as np
import pandas as pd
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
import albumentations as A
from albumentations import Flip, Resize, CenterCrop, OneOf
from albumentations.augmentations.geometric import rotate
from albumentations.augmentations.transforms import ColorJitter
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, average_precision_score
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image
from tqdm import tqdm
import yaml
from glob import glob
```

Figure 5.3.1.3 External Libraries

This section highlights the importation of external Python libraries essential for machine learning workflows. Libraries like torch and torchvision enable deep learning model development, while albumentations is used for advanced data augmentation techniques. Visualization tools like matplotlib and seaborn assist in analyzing results. The purpose of these libraries is to handle image preprocessing, model training, evaluation, and visualization, resulting in a streamlined and efficient pipeline.



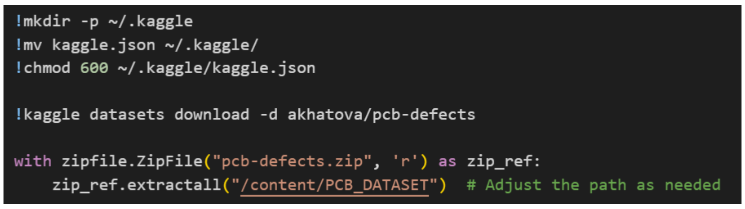
```
# IPython & Jupyter
from IPython.display import Image, clear_output, display
from IPython.core.magic import register_line_cell_magic

# YOLOv5 (Assuming utils.downloads is for YOLOv5-related tasks)
from utils.downloads import attempt_download
```

Figure 5.3.1.4 Other Libraries

The final section shows imports for notebook-specific tools (Ipython.display) and YOLOv5 utilities. The purpose of Ipython imports is to display outputs and images directly in the notebook interface for real-time feedback. The utils.downloads.attempt_download module, part of YOLOv5, is used to fetch necessary resources like pre-trained weights. The result is an enhanced interactive experience in Colab or Jupyter and an optimized setup for running YOLOv5 tasks.

Data Acquisition

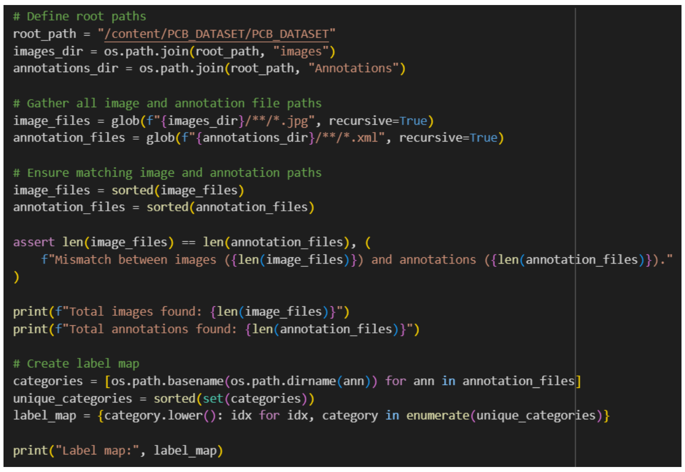
In this section of work done, datasets are acquired and unzipped after downloading it from Kaggle directly. Besides, work began to load and extract the datasets by inspecting number of images and annotations after acquiring the datasets. Also, display of label map of the datasets was done.



```
!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

!kaggle datasets download -d akhatova/pcb-defects

with zipfile.ZipFile("pcb-defects.zip", 'r') as zip_ref:
    zip_ref.extractall("/content/PCB_DATASET")  # Adjust the path as needed
```

Figure 5.3.1.5 Downloading Datasets from Kaggle

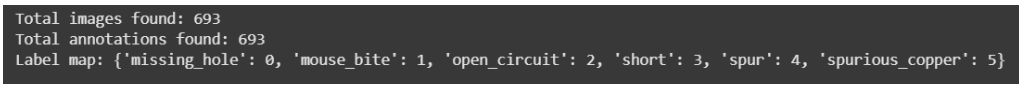
This section demonstrates the process of downloading and extracting the “pcb-defects” dataset from Kaggle. It begins by setting up the Kaggle API configuration, where the kaggle.json file containing API credentials is moved to the .kaggle directory and given secure permissions. Using the Kaggle API command, the specified dataset is downloaded directly into the workspace as a ZIP file. Finally, the zipfile.ZipFile module is used to extract the dataset contents into the /content/PCB_DATASET directory, making it ready for data pre-processing and further stages of the project.



```python
# Define root paths
root_path = "/content/PCB_DATASET/PCB_DATASET"
images_dir = os.path.join(root_path, "images")
annotations_dir = os.path.join(root_path, "Annotations")

# Gather all image and annotation file paths
image_files = glob(f"{images_dir}/**/*.jpg", recursive=True)
annotation_files = glob(f"{annotations_dir}/**/*.xml", recursive=True)

# Ensure matching image and annotation paths
image_files = sorted(image_files)
annotation_files = sorted(annotation_files)

assert len(image_files) == len(annotation_files), (
    f"Mismatch between images ({len(image_files)}) and annotations ({len(annotation_files)})."
)

print(f"Total images found: {len(image_files)}")
print(f"Total annotations found: {len(annotation_files)}")

# Create label map
categories = [os.path.basename(os.path.dirname(ann)) for ann in annotation_files]
unique_categories = sorted(set(categories))
label_map = {category.lower(): idx for idx, category in enumerate(unique_categories)}

print("Label map:", label_map)
```

Figure 5.3.1.6 Preparing image and annotation paths for datasets pre-processing

This section outlines the process of preparing image and annotation paths for dataset preprocessing. The root directory of the dataset is defined, with subdirectories for images and annotations. Using the glob module, all image files (.jpg) and annotation files (.xml) are gathered recursively. To ensure consistency, it checks if the number of image files matches the number of annotations files and raises an assertion error if they do not align. Additionally, the script creates a label map, where categories are extracted from the directory structure of annotations, and a unique mapping is generated for each category. The results include the total number of images and annotations found, along with the generated label map, which is essential for training the model.



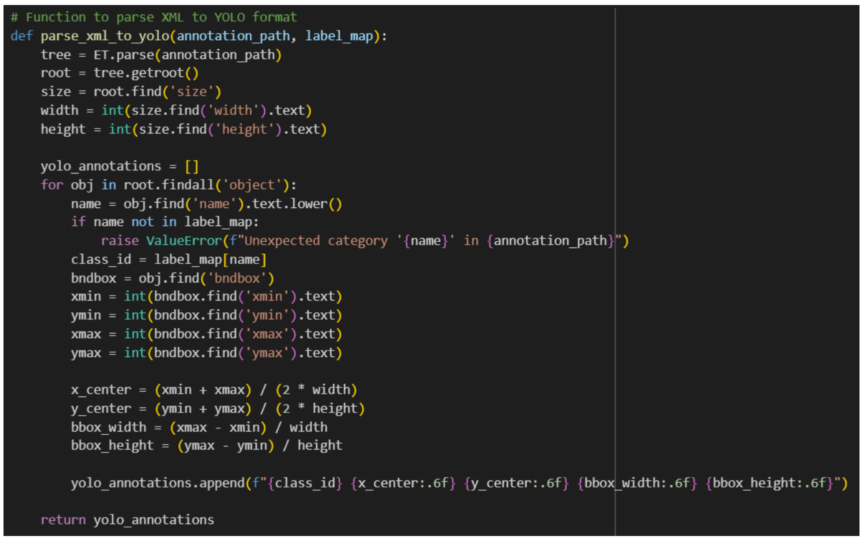
```
Total images found: 693
Total annotations found: 693
Label map: {'missing_hole': 0, 'mouse_bite': 1, 'open_circuit': 2, 'short': 3, 'spur': 4, 'spurious_copper': 5}
```

Figure 5.3.1.7 Output of preparation of data

From this operational process of figure 4.2.6, we had found 693 images, 693 annotations and label map of 6 defects such missing hole, mouse bite, open circuit, short, spur plus spurious copper.

Data Exploration and Data Pre-processing

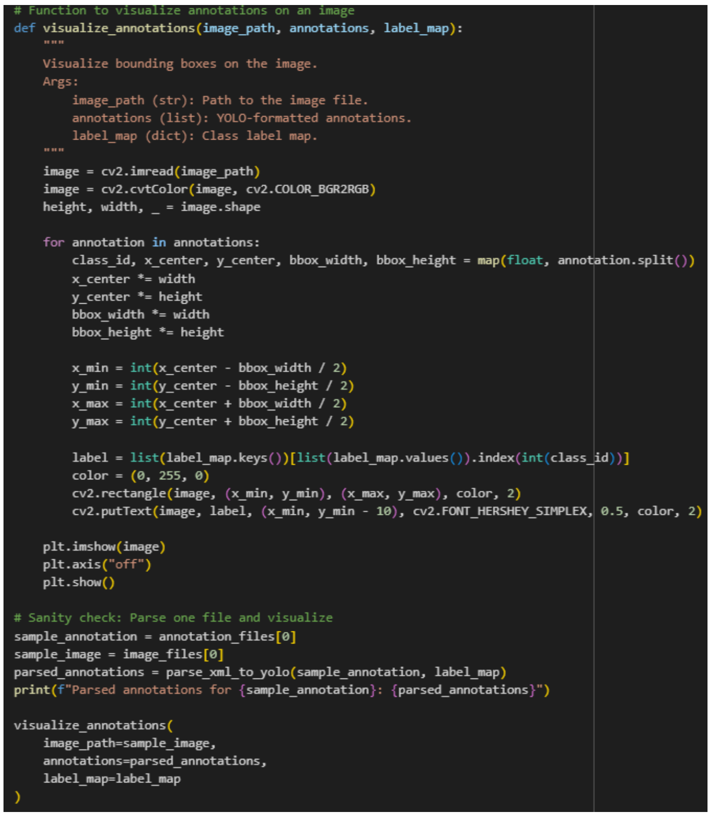
In this section of work done, operation like parsing XML to YOLO format, visualization on annotated images, data splitting and visualizing of class distribution are completed.



```
# Function to parse XML to YOLO format
def parse_xml_to_yolo(annotation_path, label_map):
    tree = ET.parse(annotation_path)
    root = tree.getroot()
    size = root.find('size')
    width = int(size.find('width').text)
    height = int(size.find('height').text)

    yolo_annotations = []
    for obj in root.findall('object'):
        name = obj.find('name').text.lower()
        if name not in label_map:
            raise ValueError(f"Unexpected category '{name}' in {annotation_path}")
        class_id = label_map[name]
        bndbox = obj.find('bndbox')
        xmin = int(bndbox.find('xmin').text)
        ymin = int(bndbox.find('ymin').text)
        xmax = int(bndbox.find('xmax').text)
        ymax = int(bndbox.find('ymax').text)

        x_center = (xmin + xmax) / (2 * width)
        y_center = (ymin + ymax) / (2 * height)
        bbox_width = (xmax - xmin) / width
        bbox_height = (ymax - ymin) / height

        yolo_annotations.append(f"{class_id} {x_center:.6f} {y_center:.6f} {bbox_width:.6f} {bbox_height:.6f}")

    return yolo_annotations
```

Figure 5.3.1.8 Function for parsing XML to YOLO format

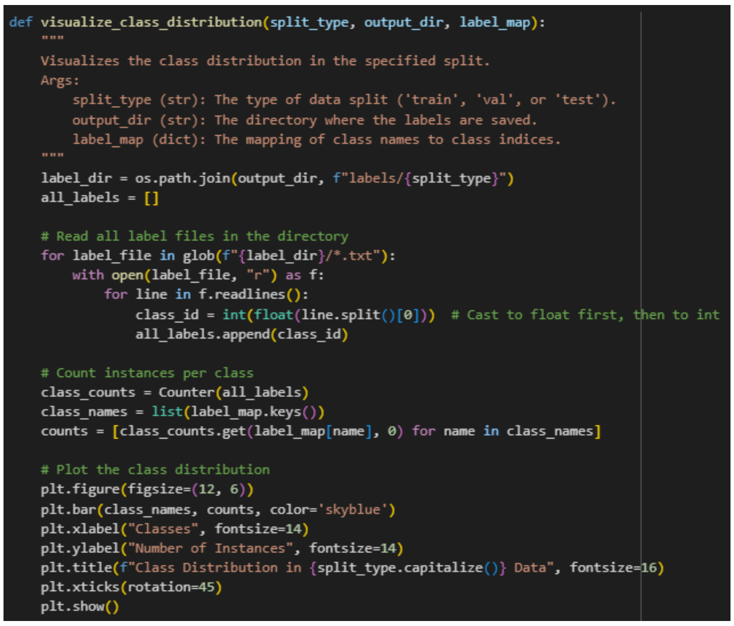
This section of the code defines a function to parse XML annotation files and convert them into the YOLO format. The function takes the annotation file path and a label_map as inputs. It parses the XML structure to extract the image dimensions and bounding box coordinates for each annotated object. Each object is verified against the label_map, ensuring it belongs to the expected categories. The bounding box coordinates are then normalized relative to the image dimensions to align with YOLO's input format, specifying the class ID, center coordinates, and bounding box width and height. The results are appended to a list and returned, ensuring compatibility with YOLO-based training pipelines. This is a critical step for preparing data annotations for model training.



```python
# Function to visualize annotations on an image
def visualize_annotations(image_path, annotations, label_map):
    """
    Visualize bounding boxes on the image.
    Args:
        image_path (str): Path to the image file.
        annotations (list): YOLO-formatted annotations.
        label_map (dict): Class label map.
    """
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    height, width, _ = image.shape

    for annotation in annotations:
        class_id, x_center, y_center, bbox_width, bbox_height = map(float, annotation.split())
        x_center *= width
        y_center *= height
        bbox_width *= width
        bbox_height *= height

        x_min = int(x_center - bbox_width / 2)
        y_min = int(y_center - bbox_height / 2)
        x_max = int(x_center + bbox_width / 2)
        y_max = int(y_center + bbox_height / 2)

        label = list(label_map.keys())[list(label_map.values()).index(int(class_id))]
        color = (0, 255, 0)
        cv2.rectangle(image, (x_min, y_min), (x_max, y_max), color, 2)
        cv2.putText(image, label, (x_min, y_min - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

    plt.imshow(image)
    plt.axis("off")
    plt.show()

# Sanity check: Parse one file and visualize
sample_annotation = annotation_files[0]
sample_image = image_files[0]
parsed_annotations = parse_xml_to_yolo(sample_annotation, label_map)
print(f"Parsed annotations for {sample_annotation}: {parsed_annotations}")

visualize_annotations(
    image_path=sample_image,
    annotations=parsed_annotations,
    label_map=label_map
)
```

Figure 5.3.1.9 Function for visualizing annotations on image.

This function visualizes YOLO-formatted annotations by overlaying bounding boxes and labels on an image. It reads the image, converts the bounding box coordinates from YOLO format to pixel dimensions, and draws rectangles and labels on the image using OpenCV. The function ensures the visualization corresponds to the correct class labels from the label_map. A sample image and its annotations are parsed and displayed for validation, helping verify the accuracy of the annotation conversion process.



```
def visualize_class_distribution(split_type, output_dir, label_map):
    """
    Visualizes the class distribution in the specified split.
    Args:
        split_type (str): The type of data split ('train', 'val', or 'test').
        output_dir (str): The directory where the labels are saved.
        label_map (dict): The mapping of class names to class indices.
    """
    label_dir = os.path.join(output_dir, f"labels/{split_type}")
    all_labels = []

    # Read all label files in the directory
    for label_file in glob(f"{label_dir}/*.txt"):
        with open(label_file, "r") as f:
            for line in f.readlines():
                class_id = int(float(line.split()[0]))  # Cast to float first, then to int
                all_labels.append(class_id)

    # Count instances per class
    class_counts = Counter(all_labels)
    class_names = list(label_map.keys())
    counts = [class_counts.get(label_map[name], 0) for name in class_names]

    # Plot the class distribution
    plt.figure(figsize=(12, 6))
    plt.bar(class_names, counts, color='skyblue')
    plt.xlabel("Classes", fontsize=14)
    plt.ylabel("Number of Instances", fontsize=14)
    plt.title(f"Class Distribution in {split_type.capitalize()} Data", fontsize=16)
    plt.xticks(rotation=45)
    plt.show()
```

Figure 5.3.1.10 Function to visualizing class distribution in the datasets.

This function visualizes the class distribution within specified data splits for example, train, validation, or test. It reads label files, counts the occurrences of each class using Counter, and generates a bar chart of class frequencies. This helps identify any class imbalances in the dataset, which is crucial for ensuring robust model training.

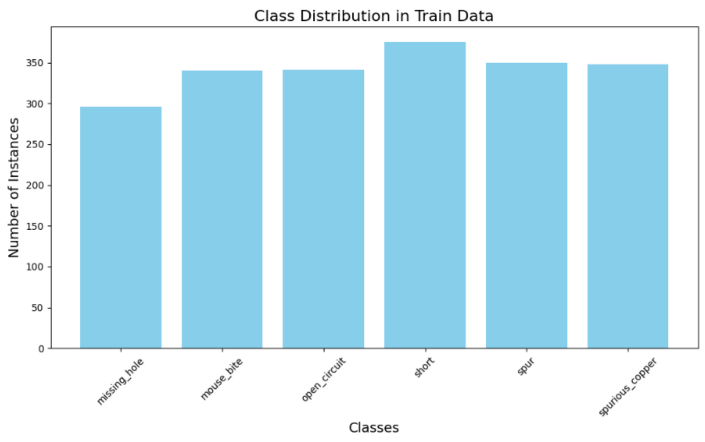


Figure 5.3.1.11 Class Distribution in Train Data.

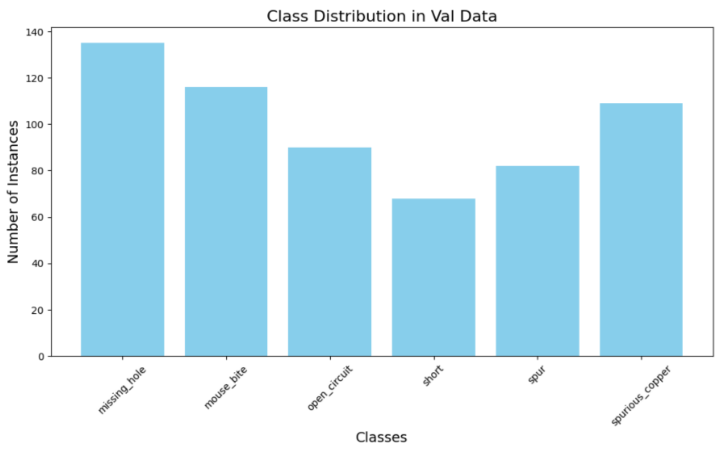


Figure 5.3.1.12 Class Distribution in Validation Data

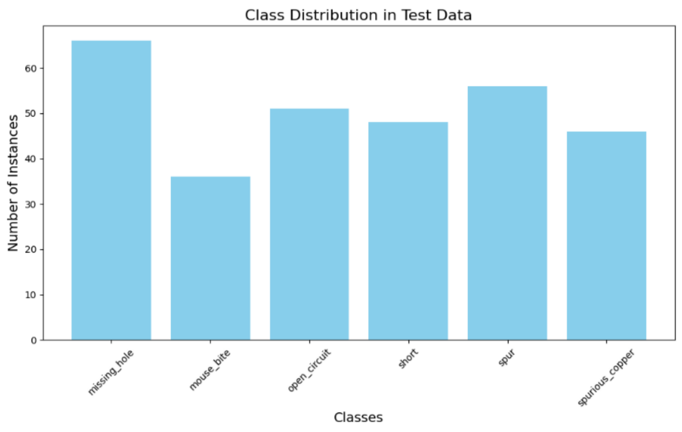
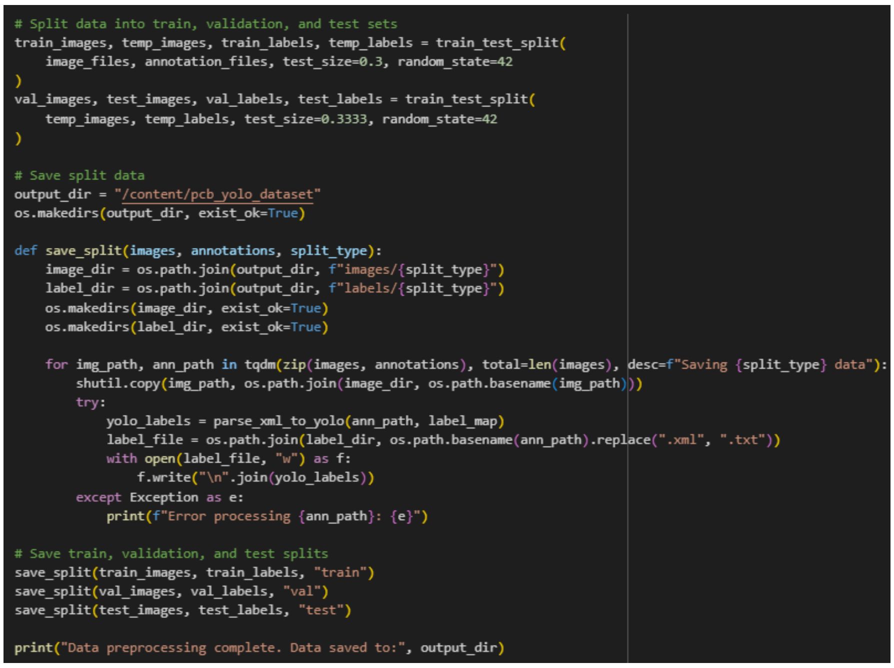


Figure 5.3.1.13 Class Distribution in Testing Data

These three bar charts display the class distribution across the training, validation, and test datasets before any augmentation is applied. The first chart highlights the frequency of each defect class in the training data, showing relatively balanced representation with slight variations among categories such as “short” and “mouse_bite.” The second chart focuses on the validation data, revealing some differences, such as a notable prominence of “missing_hole” instances. Lastly, the third chart presents the test data distribution, where “missing_hole” remains the most frequent class, while “mouse_bite” shows comparatively fewer instances. This initial analysis helps identify the dataset’s balance and ensures proper representation of all defect types before proceeding with data augmentation or training.



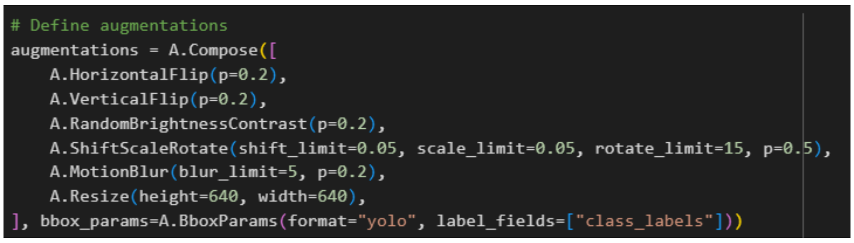
```python
# Split data into train, validation, and test sets
train_images, temp_images, train_labels, temp_labels = train_test_split(
    image_files, annotation_files, test_size=0.3, random_state=42
)
val_images, test_images, val_labels, test_labels = train_test_split(
    temp_images, temp_labels, test_size=0.3333, random_state=42
)

# Save split data
output_dir = "/content/pcb_yolo_dataset"
os.makedirs(output_dir, exist_ok=True)

def save_split(images, annotations, split_type):
    image_dir = os.path.join(output_dir, f"images/{split_type}")
    label_dir = os.path.join(output_dir, f"labels/{split_type}")
    os.makedirs(image_dir, exist_ok=True)
    os.makedirs(label_dir, exist_ok=True)

    for img_path, ann_path in tqdm(zip(images, annotations), total=len(images), desc=f"Saving {split_type} data"):
        shutil.copy(img_path, os.path.join(image_dir, os.path.basename(img_path)))
        try:
            yolo_labels = parse_xml_to_yolo(ann_path, label_map)
            label_file = os.path.join(label_dir, os.path.basename(ann_path).replace(".xml", ".txt"))
            with open(label_file, "w") as f:
                f.write("\n".join(yolo_labels))
        except Exception as e:
            print(f"Error processing {ann_path}: {e}")

# Save train, validation, and test splits
save_split(train_images, train_labels, "train")
save_split(val_images, val_labels, "val")
save_split(test_images, test_labels, "test")

print("Data preprocessing complete. Data saved to:", output_dir)
```

Figure 5.3.1.14 Data Splitting into 7:2:1 Ratio

This code processes the dataset by splitting it into training, validation, and testing sets in a 7:2:1 ratio. The train_test_split function creates these splits for both images and their corresponding annotations. A save_split function then organizes the splits into appropriate directories (images/train, images/val, images/test, etc.) while converting annotation files from XML to YOLO format. The output confirms the successful saving of each split, showing the number of processed files. This ensures the dataset is structured and ready for model training and evaluation.

Data Augmentation

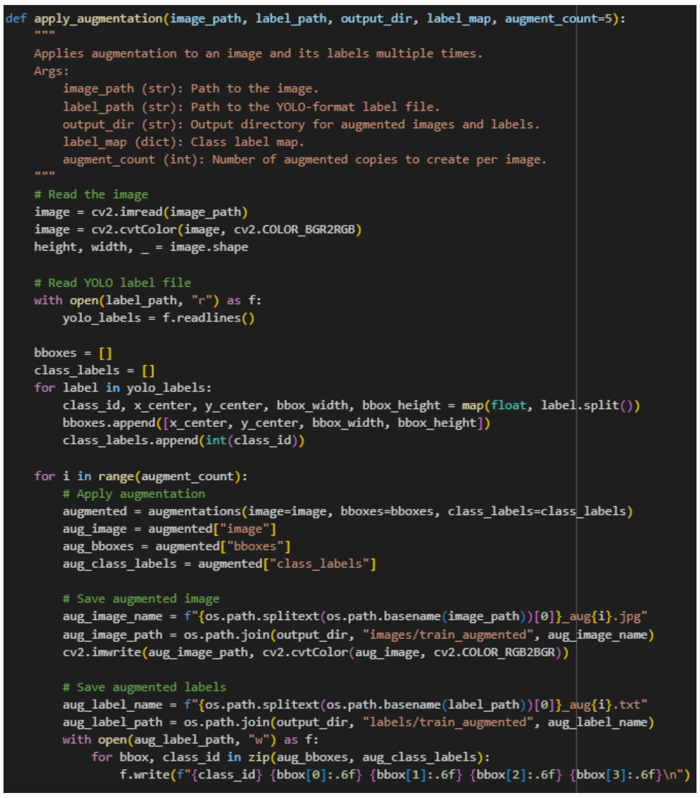
In this phase, multiple augmentation techniques such as **horizontal flip, vertical flip, random brightness, contrast adjustment, rotation, motion blur, and resizing** are applied to the dataset. These augmentations are designed to increase dataset diversity, improve model robustness, and help the model generalize better to unseen data. Following the augmentation process, the class distribution is visualized to ensure that the augmentation maintains balance across all defect types. Additionally, the number of images before and after augmentation is compared to highlight the dataset's expansion. Lastly, augmented images along with their respective annotations are visualized to verify that the transformations were correctly applied and that the annotations remain consistent.



```python
# Define augmentations
augmentations = A.Compose([
    A.HorizontalFlip(p=0.2),
    A.VerticalFlip(p=0.2),
    A.RandomBrightnessContrast(p=0.2),
    A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05, rotate_limit=15, p=0.5),
    A.MotionBlur(blur_limit=5, p=0.2),
    A.Resize(height=640, width=640),
], bbox_params=A.BboxParams(format="yolo", label_fields=["class_labels"]))
```

Figure 5.3.1.15 Defining various augmentation techniques

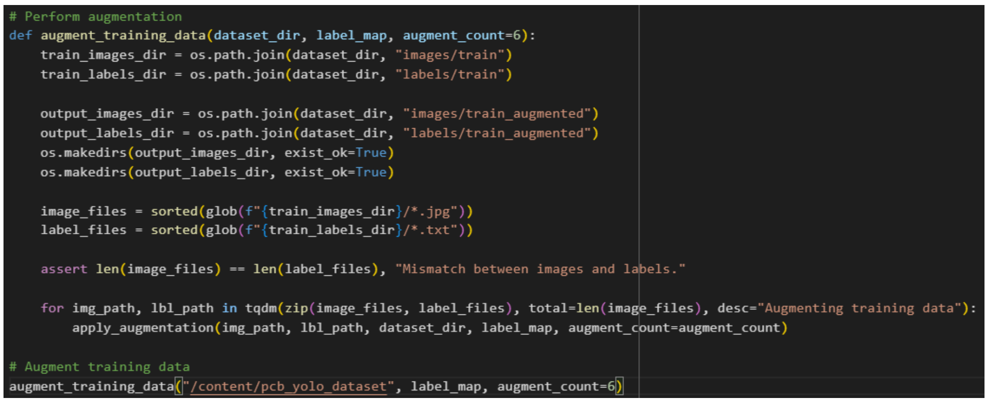
This code section defines the augmentation techniques applied to the dataset using the albumentations library. It specifies several transformations, including horizontal flip, vertical flip, random brightness and contrast adjustment, rotation with scaling and shifting, motion blur, and resizing to a fixed size of 640x640 pixels. Each transformation is assigned a probability (p) for its application. Additionally, the bounding box parameters are configured to ensure that object annotations (in YOLO format) remain accurate after augmentations. These augmentations are crucial for increasing dataset diversity and improving the model's ability to generalize to various real-world scenarios.



```python
def apply_augmentation(image_path, label_path, output_dir, label_map, augment_count=5):
    """
    Applies augmentation to an image and its labels multiple times.
    Args:
        image_path (str): Path to the image.
        label_path (str): Path to the YOLO-format label file.
        output_dir (str): Output directory for augmented images and labels.
        label_map (dict): Class label map.
        augment_count (int): Number of augmented copies to create per image.
    """
    # Read the image
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    height, width, _ = image.shape

    # Read YOLO label file
    with open(label_path, "r") as f:
        yolo_labels = f.readlines()

    bboxes = []
    class_labels = []
    for label in yolo_labels:
        class_id, x_center, y_center, bbox_width, bbox_height = map(float, label.split())
        bboxes.append([x_center, y_center, bbox_width, bbox_height])
        class_labels.append(int(class_id))

    for i in range(augment_count):
        # Apply augmentation
        augmented = augmentations(image=image, bboxes=bboxes, class_labels=class_labels)
        aug_image = augmented["image"]
        aug_bboxes = augmented["bboxes"]
        aug_class_labels = augmented["class_labels"]

        # Save augmented image
        aug_image_name = f"{os.path.splitext(os.path.basename(image_path))[0]}_aug{i}.jpg"
        aug_image_path = os.path.join(output_dir, "images/train_augmented", aug_image_name)
        cv2.imwrite(aug_image_path, cv2.cvtColor(aug_image, cv2.COLOR_RGB2BGR))

        # Save augmented labels
        aug_label_name = f"{os.path.splitext(os.path.basename(label_path))[0]}_aug{i}.txt"
        aug_label_path = os.path.join(output_dir, "labels/train_augmented", aug_label_name)
        with open(aug_label_path, "w") as f:
            for bbox, class_id in zip(aug_bboxes, aug_class_labels):
                f.write(f"{class_id} {bbox[0]:.6f} {bbox[1]:.6f} {bbox[2]:.6f} {bbox[3]:.6f}\n")
```

Figure 5.3.1.16 Function to apply data augmentations

This function, apply_augmentation, is designed to apply multiple data augmentations to a single image and its corresponding YOLO-formatted label file. It reads the image and its associated bounding box annotations, applies a predefined set of augmentations and generates augmented versions of both the image and labels. For each augmented version, the function saves the new image and its updated annotations in the specified output directory. The augment_count parameter allows for creating multiple augmented copies per input image, enhancing the dataset's diversity and robustness for training.



```python
# Perform augmentation
def augment_training_data(dataset_dir, label_map, augment_count=6):
    train_images_dir = os.path.join(dataset_dir, "images/train")
    train_labels_dir = os.path.join(dataset_dir, "labels/train")

    output_images_dir = os.path.join(dataset_dir, "images/train_augmented")
    output_labels_dir = os.path.join(dataset_dir, "labels/train_augmented")
    os.makedirs(output_images_dir, exist_ok=True)
    os.makedirs(output_labels_dir, exist_ok=True)

    image_files = sorted(glob(f"{train_images_dir}/*.jpg"))
    label_files = sorted(glob(f"{train_labels_dir}/*.txt"))

    assert len(image_files) == len(label_files), "Mismatch between images and labels."

    for img_path, lbl_path in tqdm(zip(image_files, label_files), total=len(image_files), desc="Augmenting training data"):
        apply_augmentation(img_path, lbl_path, dataset_dir, label_map, augment_count=augment_count)

# Augment training data
augment_training_data("/content/pcb_yolo_dataset", label_map, augment_count=6)
```

Figure 5.3.1.17 Function to perform augmentations

This block of code defines and performs a function called augment_training_data to apply data augmentation to the training dataset. It specifies the directories for input images and labels, as well as for storing the augmented images and labels. The function reads the training images and their respective labels, ensures that the image and label files match, and applies the augmentation process to each image-label pair using the previously defined apply_augmentation function. Here, augment_count=6 specifies that six augmented versions of each image will be generated, significantly increasing the dataset size. The progress bar confirms that all 485 training images were successfully augmented, indicating the completion of the process.

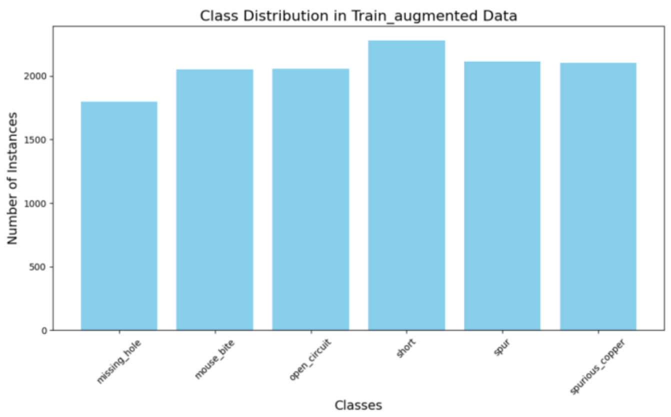
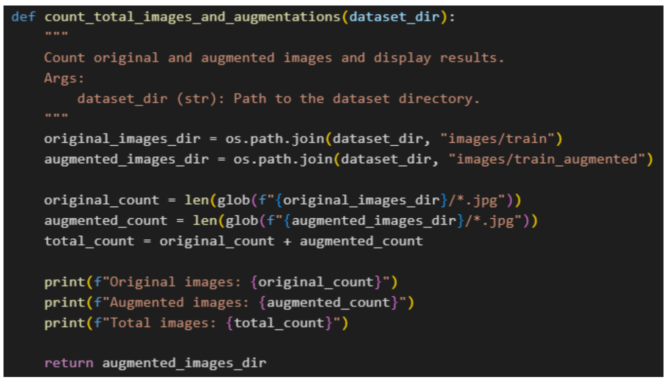


Figure 5.3.1.18 Output of class distribution in training data after augmentation

This bar chart illustrates the class distribution in the training dataset after the data augmentation process. The number of instances for each defect class—such as "missing_hole," "mouse_bite," and "spurious_copper"—has significantly increased due to augmentation techniques like flipping, brightness adjustment, rotation, and more. The chart confirms that the dataset now has a much larger and more balanced representation of each class, ensuring robust and diverse training for the model.

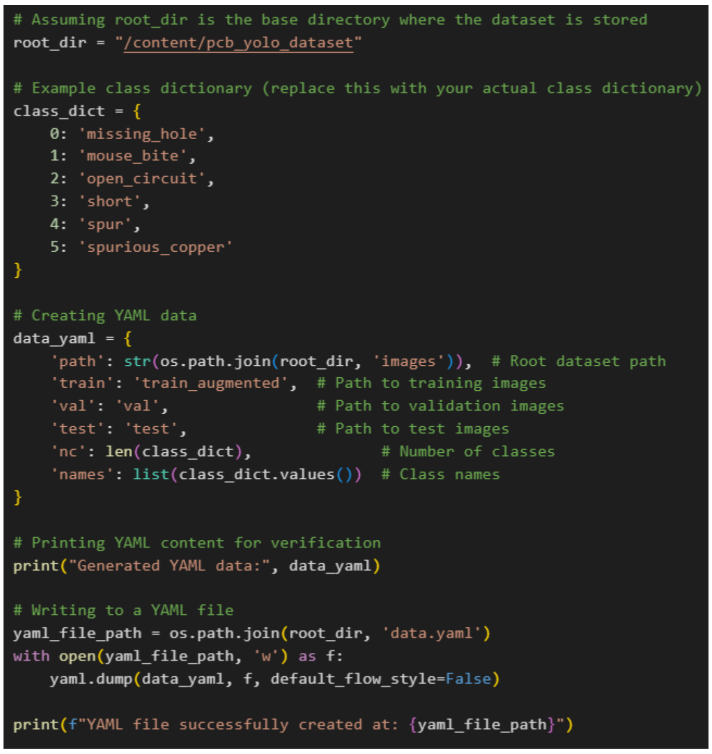


```
def count_total_images_and_augmentations(dataset_dir):
    """
    Count original and augmented images and display results.
    Args:
        dataset_dir (str): Path to the dataset directory.
    """
    original_images_dir = os.path.join(dataset_dir, "images/train")
    augmented_images_dir = os.path.join(dataset_dir, "images/train_augmented")

    original_count = len(glob(f"{original_images_dir}/*.jpg"))
    augmented_count = len(glob(f"{augmented_images_dir}/*.jpg"))
    total_count = original_count + augmented_count

    print(f"Original images: {original_count}")
    print(f"Augmented images: {augmented_count}")
    print(f"Total images: {total_count}")

    return augmented_images_dir
```

Figure 5.3.1.19 Function to count total number of images augmented

This function, count_total_images_and_augmentations, calculates and displays the number of original and augmented images within a given dataset directory. It identifies the images from the training dataset before augmentation and those generated during the augmentation process. The results show the counts of original, augmented, and total images, helping to validate the effectiveness of the augmentation process in expanding the dataset size for robust model training. The results indicate that the original dataset consists of 485 images. Through the augmentation process, an additional 2,910 images were generated, significantly increasing the dataset size to a total of 3,395 images. This expanded dataset provides a more diverse range of samples, improving the model's ability to generalize and perform effectively in different scenarios.

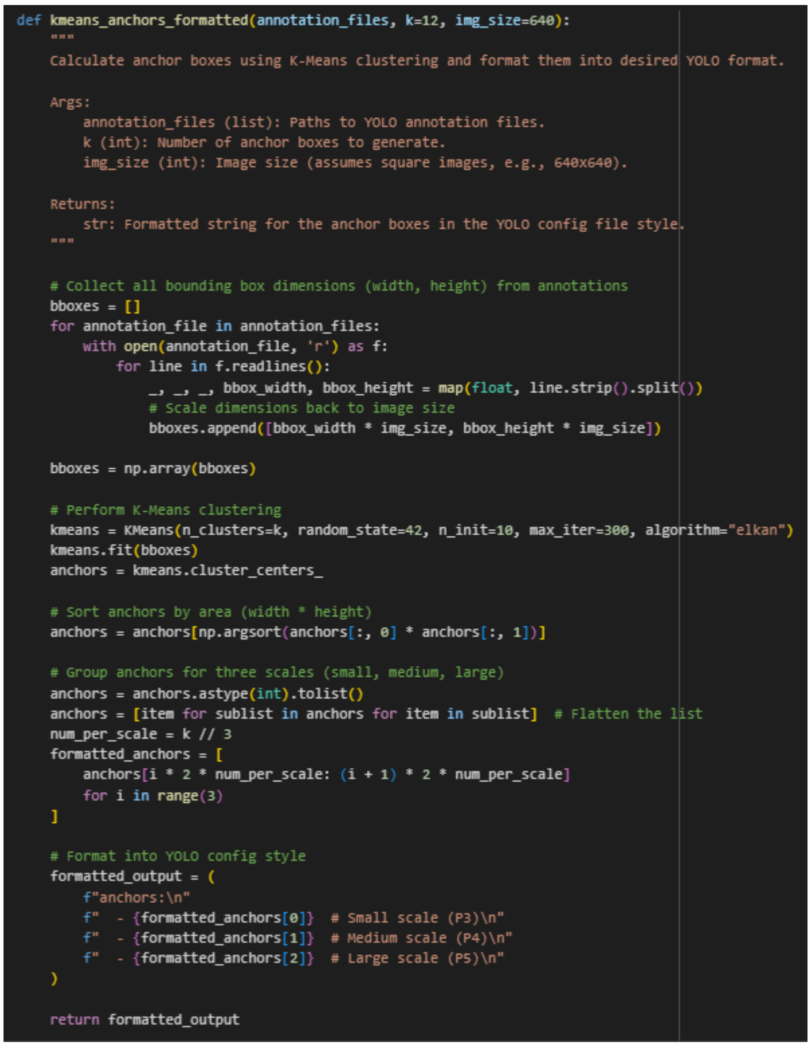
Data Preparation before Model Training



```python
# Assuming root_dir is the base directory where the dataset is stored
root_dir = "/content/pcb_yolo_dataset"

# Example class dictionary (replace this with your actual class dictionary)
class_dict = {
    0: 'missing_hole',
    1: 'mouse_bite',
    2: 'open_circuit',
    3: 'short',
    4: 'spur',
    5: 'spurious_copper'
}

# Creating YAML data
data_yaml = {
    'path': str(os.path.join(root_dir, 'images')),  # Root dataset path
    'train': 'train_augmented',  # Path to training images
    'val': 'val',                # Path to validation images
    'test': 'test',              # Path to test images
    'nc': len(class_dict),              # Number of classes
    'names': list(class_dict.values())  # Class names
}

# Printing YAML content for verification
print("Generated YAML data:", data_yaml)

# Writing to a YAML file
yaml_file_path = os.path.join(root_dir, 'data.yaml')
with open(yaml_file_path, 'w') as f:
    yaml.dump(data_yaml, f, default_flow_style=False)

print(f"YAML file successfully created at: {yaml_file_path}")
```

Figure 5.3.1.20 Data YAML file preparation

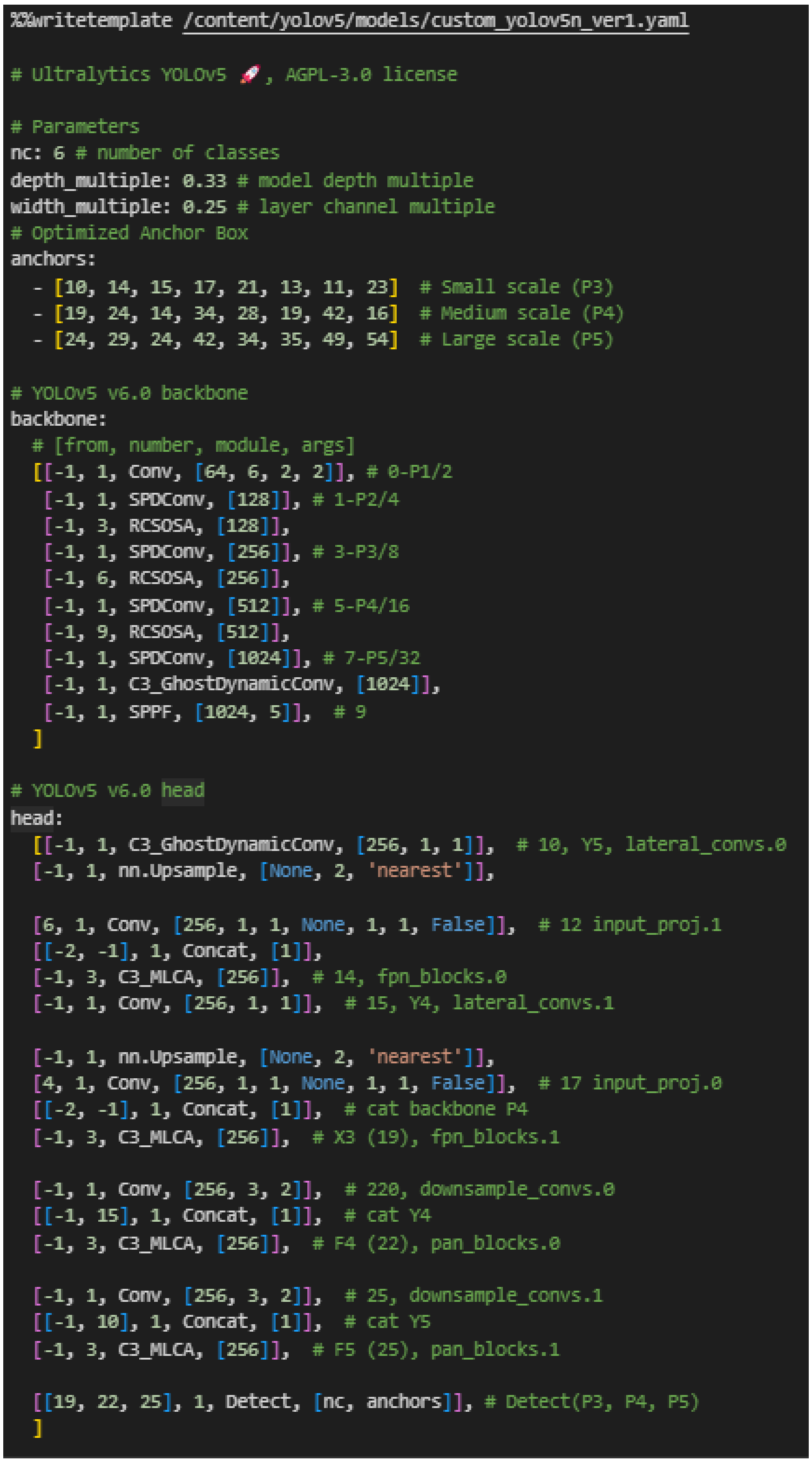
This code snippet demonstrates the creation of a YAML configuration file for the YOLO dataset. It defines essential paths and parameters for the training, validation, and testing datasets while specifying the number of classes (nc) and their corresponding names (names). After generating the YAML structure, the configuration is printed for verification and then written to a file (data.yaml). This file is a crucial component for guiding YOLO-based models during the training process by providing clear mappings of dataset directories and class labels.



```python
def kmeans_anchors_formatted(annotation_files, k=12, img_size=640):
    """
    Calculate anchor boxes using K-Means clustering and format them into desired YOLO format.

    Args:
        annotation_files (list): Paths to YOLO annotation files.
        k (int): Number of anchor boxes to generate.
        img_size (int): Image size (assumes square images, e.g., 640x640).

    Returns:
        str: Formatted string for the anchor boxes in the YOLO config file style.
    """

    # Collect all bounding box dimensions (width, height) from annotations
    bboxes = []
    for annotation_file in annotation_files:
        with open(annotation_file, 'r') as f:
            for line in f.readlines():
                _, _, _, bbox_width, bbox_height = map(float, line.strip().split())
                # Scale dimensions back to image size
                bboxes.append([bbox_width * img_size, bbox_height * img_size])

    bboxes = np.array(bboxes)

    # Perform K-Means clustering
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10, max_iter=300, algorithm="elkan")
    kmeans.fit(bboxes)
    anchors = kmeans.cluster_centers_

    # Sort anchors by area (width * height)
    anchors = anchors[np.argsort(anchors[:, 0] * anchors[:, 1])]

    # Group anchors for three scales (small, medium, large)
    anchors = anchors.astype(int).tolist()
    anchors = [item for sublist in anchors for item in sublist]  # Flatten the list
    num_per_scale = k // 3
    formatted_anchors = [
        anchors[i * 2 * num_per_scale: (i + 1) * 2 * num_per_scale]
        for i in range(3)
    ]

    # Format into YOLO config style
    formatted_output = (
        f"anchors:\n"
        f"  - {formatted_anchors[0]}  # Small scale (P3)\n"
        f"  - {formatted_anchors[1]}  # Medium scale (P4)\n"
        f"  - {formatted_anchors[2]}  # Large scale (P5)\n"
    )

    return formatted_output
```

Figure 5.3.1.21 K-Means Clustering Function

This code highlights a critical issue in using non-optimized anchor boxes, which are typically designed for general-purpose datasets like COCO. The COCO dataset primarily contains a wide range of object sizes, from small to large, making its default anchor boxes suboptimal for specialized tasks like PCB defect detection. PCB defects are often small and intricate, requiring anchor boxes tailored to detect such fine-scale objects accurately. The function addresses this challenge by using Binary K-Means clustering to optimize the anchor boxes specifically for the PCB dataset. This ensures the anchor boxes are better aligned with the size and scale of defects, thereby improving detection accuracy for small objects and minimizing errors during training and inference.



```
%%writetemplate /content/yolov5/models/custom_yolov5n_ver1.yaml

# Ultralytics YOLOv5 🚀, AGPL-3.0 license

# Parameters
nc: 6 # number of classes
depth_multiple: 0.33 # model depth multiple
width_multiple: 0.25 # layer channel multiple
# Optimized Anchor Box
anchors:
  - [10, 14, 15, 17, 21, 13, 11, 23]  # Small scale (P3)
  - [19, 24, 14, 34, 28, 19, 42, 16]  # Medium scale (P4)
  - [24, 29, 24, 42, 34, 35, 49, 54]  # Large scale (P5)

# YOLOv5 v6.0 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [64, 6, 2, 2]], # 0-P1/2
   [-1, 1, SPDConv, [128]], # 1-P2/4
   [-1, 3, RCSOSA, [128]],
   [-1, 1, SPDConv, [256]], # 3-P3/8
   [-1, 6, RCSOSA, [256]],
   [-1, 1, SPDConv, [512]], # 5-P4/16
   [-1, 9, RCSOSA, [512]],
   [-1, 1, SPDConv, [1024]], # 7-P5/32
   [-1, 1, C3_GhostDynamicConv, [1024]],
   [-1, 1, SPPF, [1024, 5]],  # 9
  ]

# YOLOv5 v6.0 head
head:
  [[-1, 1, C3_GhostDynamicConv, [256, 1, 1]],  # 10, Y5, lateral_convs.0
  [-1, 1, nn.Upsample, [None, 2, 'nearest']],

  [6, 1, Conv, [256, 1, 1, None, 1, 1, False]],  # 12 input_proj.1
  [[-2, -1], 1, Concat, [1]],
  [-1, 3, C3_MLCA, [256]],  # 14, fpn_blocks.0
  [-1, 1, Conv, [256, 1, 1]],  # 15, Y4, lateral_convs.1

  [-1, 1, nn.Upsample, [None, 2, 'nearest']],
  [4, 1, Conv, [256, 1, 1, None, 1, 1, False]],  # 17 input_proj.0
  [[-2, -1], 1, Concat, [1]],  # cat backbone P4
  [-1, 3, C3_MLCA, [256]],  # X3 (19), fpn_blocks.1

  [-1, 1, Conv, [256, 3, 2]],  # 220, downsample_convs.0
  [[-1, 15], 1, Concat, [1]],  # cat Y4
  [-1, 3, C3_MLCA, [256]],  # F4 (22), pan_blocks.0

  [-1, 1, Conv, [256, 3, 2]],  # 25, downsample_convs.1
  [[-1, 10], 1, Concat, [1]],  # cat Y5
  [-1, 3, C3_MLCA, [256]],  # F5 (25), pan_blocks.1

  [[19, 22, 25], 1, Detect, [nc, anchors]], # Detect(P3, P4, P5)
  ]
```

Figure 5.3.1.22 Modified Model YAML file

The LW-YOLOv5 configuration is carefully designed to optimize the model for detecting small defects on printed circuit boards (PCBs), addressing key challenges posed by using resource-constrain issues for micro-controller and small defect detection in PCB dataset. The configuration features tailored anchor boxes derived using K-

Means clustering, specifically adjusted for the small and intricate defects common in PCBs. By replacing the default anchor boxes, which are optimized and well-distributed across smaller objects on the dataset, this setup significantly improves detection accuracy for PCB-specific tasks.

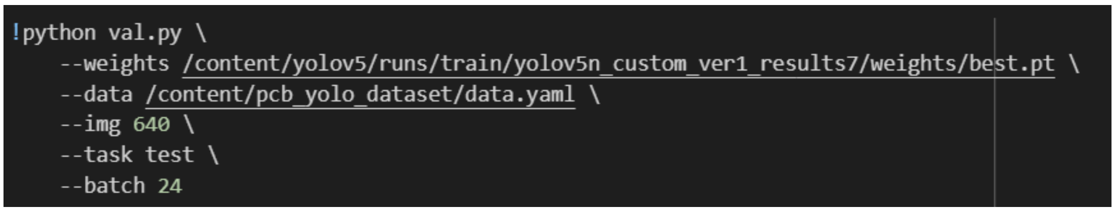
The backbone of the model integrates advanced modules like SPDConv and MLCA, CRFM, RCSOSA, C3-GhostDynamicConv, which enhance feature extraction while keeping the model lightweight. This is critical for ensuring the model can run efficiently on resource-constrained environments like microcontrollers. The detection head remained unchanged for the original YOLOv5 structure. As training and testing process goes on, various customized detection head methods had been referenced and test out. For instances, decoupled head, ASFF detection head and additional smaller detection head. Despite their addition can mean greater return in term of mAP, it could not be denied that the facts of their heavy computational costs and high parameter size as it is not applicable for our approach for lightweight and focuses on embedded system development.

Model Training

In this section, original prescript training pipelines are used from the original YOLO training methods which is cloned from GitHub. This script trains our model designed for PCB defect detection. The hyperparameters are carefully chosen to optimize the balance between performance and resource efficiency. By specifying higher image size and batch size, it ensures model learning capability for small defects. The AdamW optimizer enhances convergence, and early stopping prevents overfitting by terminating training if no improvements are observed after 10 epochs. The entire setup is tailored for precise, lightweight deployment on resource-constrained environments.

Model Evaluation

In this section, TensorBoard was utilized to help with the visualization of various graphical views of the model results and a general view to the model performance. Also, model evaluation using testing results was conducted to verify inferences time, NMS time and pre-process speed for the models. Finally, visualization on every testing image for each class is visualized.



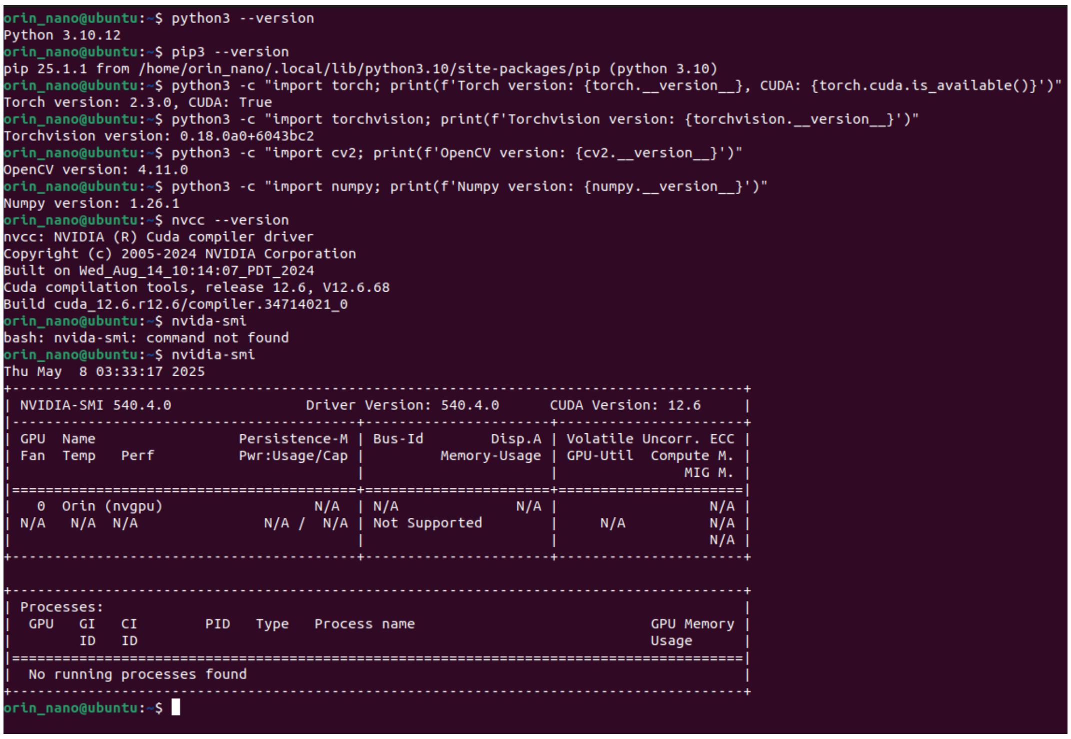
```
!python val.py \
    --weights /content/yolov5/runs/train/yolov5n_custom_ver1_results7/weights/best.pt \
    --data /content/pcb_yolo_dataset/data.yaml \
    --img 640 \
    --task test \
    --batch 24
```

Figure 5.3.1.23 Model Evaluation using testing data

This section of the code is used to validate the trained YOLOv5 model's performance on the test dataset. The val.py script is called with the weights file (best.pt) and dataset configuration (data.yaml). The image size is set to 640 pixels (--img 640), and the batch size is 24 (--batch 24) for efficient processing. The --task test option specifies that the validation should be performed on the test dataset. This step evaluates metrics like precision, recall, and mean Average Precision (mAP), providing insights into how well the model generalizes to unseen data.

5.3.2 Setup for Libraries in Ubuntu 22.04 LTS

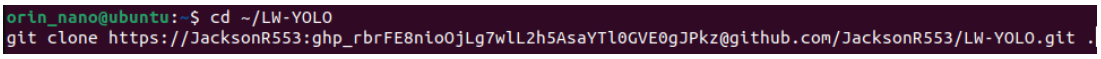
In this work section, the setup for Ubuntu 22.04 LTS on the Nvidia Jetson Orin Nano was already done and pre-existing. Basically, the system was up and running, no flash image on operating system was needed, necessary dependencies like Jetpack, or Python was pre-installed. Here, evaluation and testing attempts was made to check the current environment need to install any missing dependencies in order for deployment work to be done easily.



```
orin_nano@ubuntu:~$ python3 --version
Python 3.10.12
orin_nano@ubuntu:~$ pip3 --version
pip 25.1.1 from /home/orin_nano/.local/lib/python3.10/site-packages/pip (python 3.10)
orin_nano@ubuntu:~$ python3 -c "import torch; print(f'Torch version: {torch.__version__}, CUDA: {torch.cuda.is_available()}')"
Torch version: 2.3.0, CUDA: True
orin_nano@ubuntu:~$ python3 -c "import torchvision; print(f'Torchvision version: {torchvision.__version__}')"
Torchvision version: 0.18.0a0+6043bc2
orin_nano@ubuntu:~$ python3 -c "import cv2; print(f'OpenCV version: {cv2.__version__}')"
OpenCV version: 4.11.0
orin_nano@ubuntu:~$ python3 -c "import numpy; print(f'Numpy version: {numpy.__version__}')"
Numpy version: 1.26.1
orin_nano@ubuntu:~$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Wed_Aug_14_10:14:07_PDT_2024
Cuda compilation tools, release 12.6, V12.6.68
Build cuda_12.6.r12.6/compiler.34714021_0
orin_nano@ubuntu:~$ nvida-smi
bash: nvida-smi: command not found
orin_nano@ubuntu:~$ nvidia-smi
Thu May  8 03:33:17 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 540.4.0                 Driver Version: 540.4.0      CUDA Version: 12.6     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Orin (nvgpu)                  N/A  | N/A             N/A  |                  N/A |
| N/A   N/A  N/A               N/A /  N/A | Not Supported        |     N/A          N/A |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
orin_nano@ubuntu:~$ 
```

Figure 5.3.2.1 Checking of missing dependencies on Ubuntu 22.04

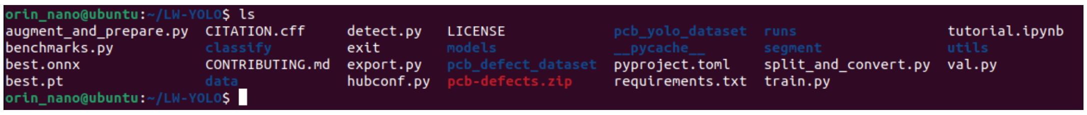
The terminal session confirms that Python 3.10.12 is the active interpreter, pip 25.1.4 manages package installations, and the core scientific stack—PyTorch 2.3.0 with CUDA support, Torchvision 0.18.0+cu1262, OpenCV 4.11.0, and NumPy 1.26.1—is correctly imported. The NVIDIA toolchain is also validated: nvcc --version reports CUDA 12.6 build V12.6.68 and driver build 34714021_0, while nvidia-smi detects the Orin GPU under driver 540.4 with compute capability exposed through CUDA 12.6. Successful execution of these queries guarantees that GPU-accelerated kernels are available for training and inference.



```
orin_nano@ubuntu:~$ cd ~/LW-YOLO
git clone https://JacksonR553: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@github.com/JacksonR553/LW-YOLO.git .
```

Figure 5.3.2.2 Cloning the LW-YOLO source tree

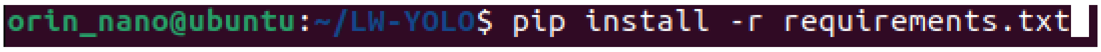
Navigating to the workspace (cd ~/LW-YOLO) and executing git clone pulls the customised repository from GitHub. Cloning at this stage captures the exact commit required for reproduction and places all scripts, models, and configuration files in a single project directory.



```
orin_nano@ubuntu:~/LW-YOLO$ ls
augment_and_prepare.py  CITATION.cff     detect.py   LICENSE              pcb_yolo_dataset  runs                 tutorial.ipynb
benchmarks.py           classify         exit        models               __pycache__       segment              utils
best.onnx               CONTRIBUTING.md  export.py   pcb_defect_dataset   pyproject.toml    split_and_convert.py val.py
best.pt                 data             hubconf.py  pcb-defects.zip      requirements.txt  train.py
orin_nano@ubuntu:~/LW-YOLO$ 
```

Figure 5.3.2.3 Inspecting repository contents

A subsequent ls lists the fetched artefacts: training and inference scripts (train.py, detect.py, val.py), utility modules, the PCB-defect dataset archive, two inference checkpoints (best.pt, best.onnx), and a requirements.txt file. Presence of these files confirms that both the model weights and the data-processing pipeline are under version control.



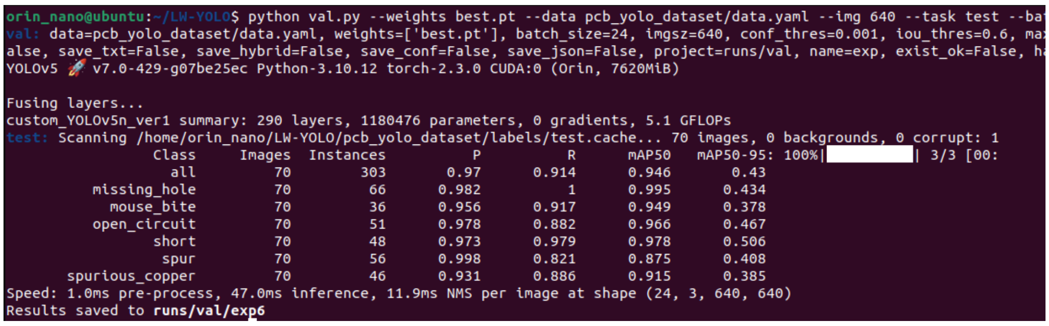
```
orin_nano@ubuntu:~/LW-YOLO$ pip install -r requirements.txt
```

Figure 5.3.2.4 Installing project-specific dependencies

Running pip install -r requirements.txt inside the previously created virtual environment resolves all ancillary Python packages referenced by LW-YOLO. Dependency locking through the requirements file ensures that the runtime environment matches that used during development, eliminating version-mismatch errors in later stages of the workflow.

5.4 System Operation (with Screenshot)

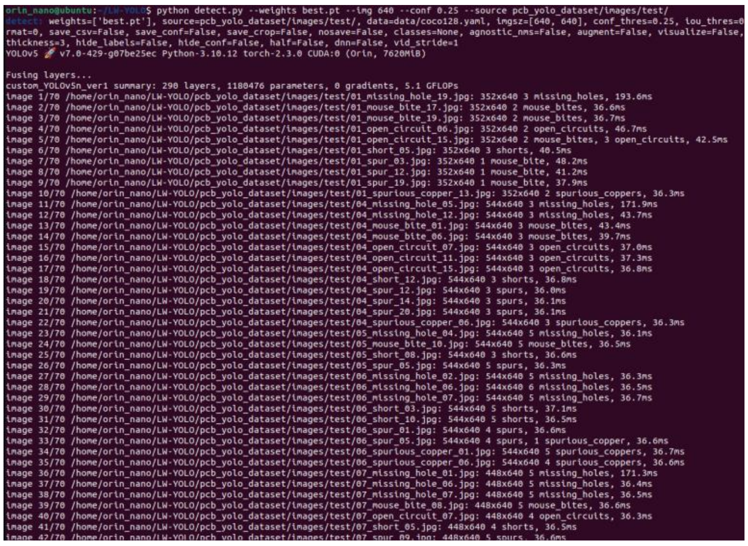
The system-operation stage documents how the trained LW-YOLO model is exercised on the Jetson Orin Nano under three execution modes—full-precision PyTorch, half-precision PyTorch, and ONNX Runtime—while recording quantitative metrics and visual inspection outputs. Each screenshot is presented in the chronological order of execution.



```
orin_nano@ubuntu:~/LW-YOLO$ python val.py --weights best.pt --data pcb_yolo_dataset/data.yaml --img 640 --task test --ba
val: data=pcb_yolo_dataset/data.yaml, weights=['best.pt'], batch_size=24, imgsz=640, conf_thres=0.001, iou_thres=0.6, ma
alse, save_txt=False, save_hybrid=False, save_conf=False, save_json=False, project=runs/val, name=exp, exist_ok=False, h
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Fusing layers...
custom_YOLOv5n_ver1 summary: 290 layers, 1180476 parameters, 0 gradients, 5.1 GFLOPs
test: Scanning /home/orin_nano/LW-YOLO/pcb_yolo_dataset/labels/test.cache... 70 images, 0 backgrounds, 0 corrupt: 1
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|          | 3/3 [00:
                   all         70        303       0.97      0.914      0.946       0.43
          missing_hole         70         66      0.982          1      0.995      0.434
            mouse_bite         70         36      0.956      0.917      0.949      0.378
          open_circuit         70         51      0.978      0.882      0.966      0.467
                 short         70         48      0.973      0.979      0.978      0.506
                  spur         70         56      0.998      0.821      0.875      0.408
       spurious_copper         70         46      0.931      0.886      0.915      0.385
Speed: 1.0ms pre-process, 47.0ms inference, 11.9ms NMS per image at shape (24, 3, 640, 640)
Results saved to runs/val/exp6
```

Figure 5.4.1 Full-precision evaluation with val.py

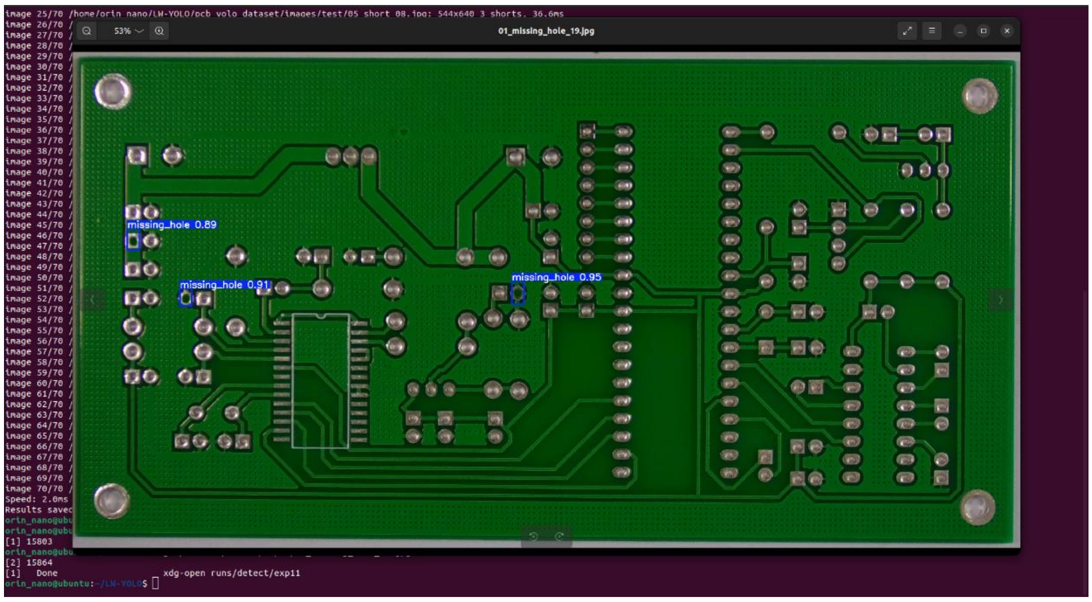
The first terminal capture shows the model being benchmarked on the test subset (--task test) with the checkpoint *best.pt*. PyTorch loads 290 layers and 118 M parameters, fuses convolution–batch-norm pairs, and produces class-wise precision, recall, mAP^{50}, and mAP$^{50–95}$ scores. Aggregate inference latency is reported at **47 ms per image** with an additional 1 ms for preprocessing and 11.9 ms for non-max suppression, establishing the FP32 performance baseline.



```
orin_nano@ubuntu:~/LW-YOLO$ python detect.py --weights best.pt --img 640 --conf 0.25 --source pcb_yolo_dataset/images/test/
detect: weights=['best.pt'], source=pcb_yolo_dataset/images/test/, data=data/coco128.yaml, imgsz=[640, 640], conf_thres=0.25, iou_thres=0
rmat=0, save_csv=False, save_conf=False, save_crop=False, nosave=False, classes=None, agnostic_nms=False, augment=False, visualize=False,
thickness=3, hide_labels=False, hide_conf=False, half=False, dnn=False, vid_stride=1
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Fusing layers...
custom_YOLOv5n_ver1 summary: 290 layers, 1180476 parameters, 0 gradients, 5.1 GFLOPs
image 1/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_missing_hole_19.jpg: 352x640 3 missing_holes, 193.6ms
image 2/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_mouse_bite_17.jpg: 352x640 2 mouse_bites, 36.6ms
image 3/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_mouse_bite_19.jpg: 352x640 2 mouse_bites, 36.7ms
image 4/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_open_circuit_06.jpg: 352x640 2 open_circuits, 46.7ms
image 5/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_open_circuit_15.jpg: 352x640 2 mouse_bites, 3 open_circuits, 42.5ms
image 6/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_short_05.jpg: 352x640 3 shorts, 40.5ms
image 7/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_03.jpg: 352x640 1 mouse_bite, 48.2ms
image 8/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_12.jpg: 352x640 1 mouse_bite, 41.2ms
image 9/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_19.jpg: 352x640 1 mouse_bite, 37.9ms
image 10/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spurious_copper_13.jpg: 352x640 2 spurious_coppers, 36.3ms
image 11/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_missing_hole_05.jpg: 544x640 3 missing_holes, 171.9ms
image 12/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_missing_hole_12.jpg: 544x640 3 missing_holes, 43.7ms
image 13/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_mouse_bite_01.jpg: 544x640 3 mouse_bites, 43.4ms
image 14/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_mouse_bite_06.jpg: 544x640 3 mouse_bites, 39.7ms
image 15/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_07.jpg: 544x640 3 open_circuits, 37.0ms
image 16/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_11.jpg: 544x640 3 open_circuits, 37.3ms
image 17/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_15.jpg: 544x640 3 open_circuits, 36.8ms
image 18/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_short_12.jpg: 544x640 3 shorts, 36.8ms
image 19/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_12.jpg: 544x640 3 spurs, 36.0ms
image 20/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_14.jpg: 544x640 3 spurs, 36.1ms
image 21/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_20.jpg: 544x640 3 spurs, 36.1ms
image 22/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spurious_copper_06.jpg: 544x640 3 spurious_coppers, 36.3ms
image 23/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_missing_hole_04.jpg: 544x640 5 missing_holes, 36.1ms
image 24/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_mouse_bite_10.jpg: 544x640 5 mouse_bites, 36.5ms
image 25/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_short_08.jpg: 544x640 3 shorts, 36.6ms
image 26/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_spur_05.jpg: 544x640 5 spurs, 36.3ms
image 27/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_02.jpg: 544x640 5 missing_holes, 36.3ms
image 28/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_06.jpg: 544x640 6 missing_holes, 36.5ms
image 29/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_07.jpg: 544x640 5 missing_holes, 36.7ms
image 30/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_short_03.jpg: 544x640 5 shorts, 37.1ms
image 31/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_short_10.jpg: 544x640 5 shorts, 36.5ms
image 32/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spur_01.jpg: 544x640 4 spurs, 36.6ms
image 33/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spur_05.jpg: 544x640 4 spurs, 1 spurious_copper, 36.6ms
image 34/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spurious_copper_01.jpg: 544x640 5 spurious_coppers, 36.7ms
image 35/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spurious_copper_06.jpg: 544x640 4 spurious_coppers, 36.6ms
image 36/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_01.jpg: 448x640 5 missing_holes, 171.3ms
image 37/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_06.jpg: 448x640 5 missing_holes, 36.4ms
image 38/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_07.jpg: 448x640 5 missing_holes, 36.5ms
image 39/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_mouse_bite_08.jpg: 448x640 5 mouse_bites, 36.6ms
image 40/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_open_circuit_07.jpg: 448x640 4 open_circuits, 36.3ms
image 41/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_short_05.jpg: 448x640 4 shorts, 36.5ms
image 42/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_spur_09.jpg: 448x640 5 spurs, 36.6ms
```

Figure 5.4.2 Full-precision batch detection with detect.py

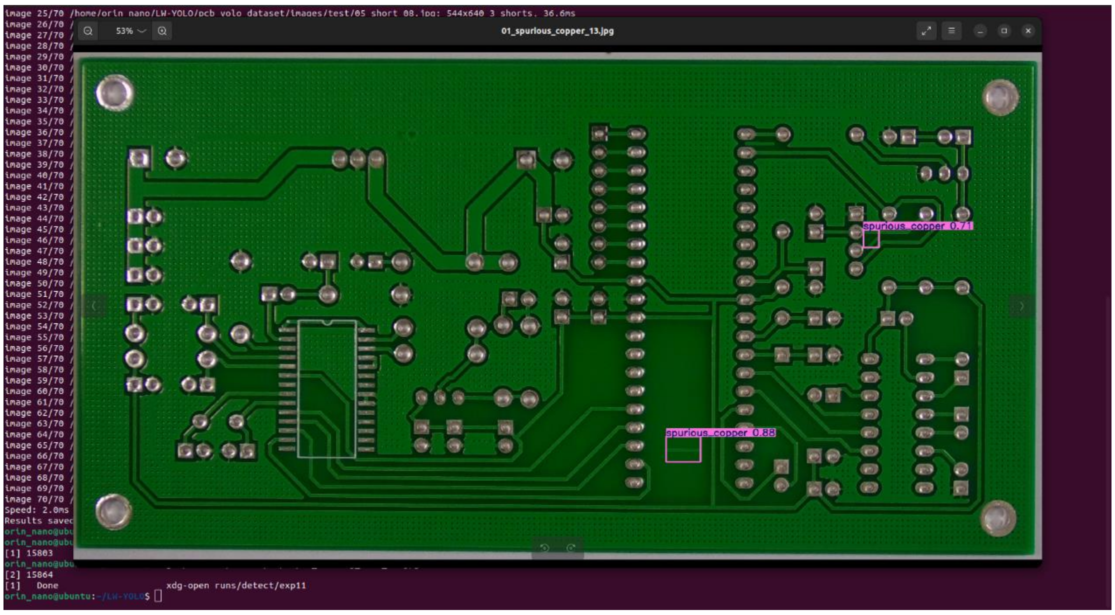
The next screenshot records an end-to-end sweep of the same 70 test images. For each frame, the console lists the resolution, the detected class labels, the object count, and the wall-clock inference time (36–45 ms). These logs confirm deterministic throughput when the model is executed in FP32.



```
image 25/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_short_08.jpg: 544x640 3 shorts, 36.6ms
image 26/70 /
image 27/70 /
image 28/70 /
image 29/70 /
image 30/70 /
image 31/70 /
image 32/70 /
image 33/70 /
image 34/70 /
image 35/70 /
image 36/70 /
image 37/70 /
image 38/70 /
image 39/70 /
image 40/70 /
image 41/70 /
image 42/70 /
image 43/70 /
image 44/70 /
image 45/70 /
image 46/70 /
image 47/70 /
image 48/70 /
image 49/70 /
image 50/70 /
image 51/70 /
image 52/70 /
image 53/70 /
image 54/70 /
image 55/70 /
image 56/70 /
image 57/70 /
image 58/70 /
image 59/70 /
image 60/70 /
image 61/70 /
image 62/70 /
image 63/70 /
image 64/70 /
image 65/70 /
image 66/70 /
image 67/70 /
image 68/70 /
image 69/70 /
image 70/70 /
Speed: 2.0ms
Results savec
orin_nano@ubu
orin_nano@ubu
[1] 15803
orin_nano@ubu
[2] 15864
[1]   Done                    xdg-open runs/detect/exp11
orin_nano@ubuntu:~/LW-YOLO$ 
```

Figure 5.4.3 Visual Output of full-precision detection 1

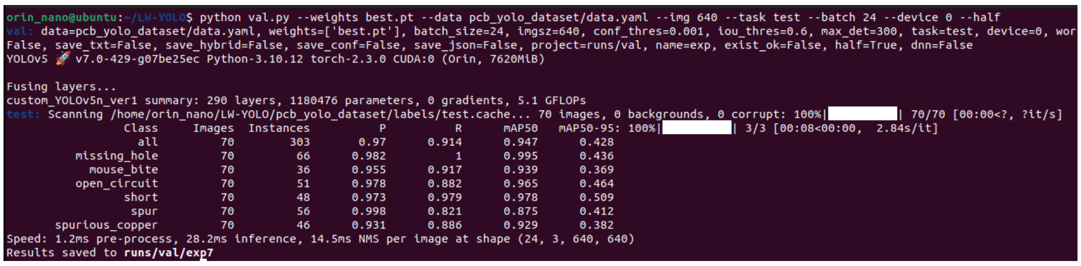
The corresponding image viewer display which runs on xdg command, renders bounding boxes and confidence scores for the *missing_hole* class on a single PCB sample. Accurate localisation and confidence values between 0.89 and 0.95 corroborate the numerical metrics from Figure 5.4.1.



```
image 25/70 /home/orin nano/LW-YOLO/pcb volo dataset/images/test/05 short 08.jpg: 544x640 3 shorts, 36.6ms
image 26/70 /
image 27/70 /
image 28/70 /
image 29/70 /
image 30/70 /
image 31/70 /
image 32/70 /
image 33/70 /
image 34/70 /
image 35/70 /
image 36/70 /
image 37/70 /
image 38/70 /
image 39/70 /
image 40/70 /
image 41/70 /
image 42/70 /
image 43/70 /
image 44/70 /
image 45/70 /
image 46/70 /
image 47/70 /
image 48/70 /
image 49/70 /
image 50/70 /
image 51/70 /
image 52/70 /
image 53/70 /
image 54/70 /
image 55/70 /
image 56/70 /
image 57/70 /
image 58/70 /
image 59/70 /
image 60/70 /
image 61/70 /
image 62/70 /
image 63/70 /
image 64/70 /
image 65/70 /
image 66/70 /
image 67/70 /
image 68/70 /
image 69/70 /
image 70/70 /
Speed: 2.0ms
Results savec
orin_nano@ubu
orin_nano@ubu
[1] 15803
orin_nano@ubu
[2] 15864
[1]   Done                    xdg-open runs/detect/exp11
orin_nano@ubuntu:~/LW-YOLO$ 
```

Figure 5.4.4 Visual Output of full-precision detection 2

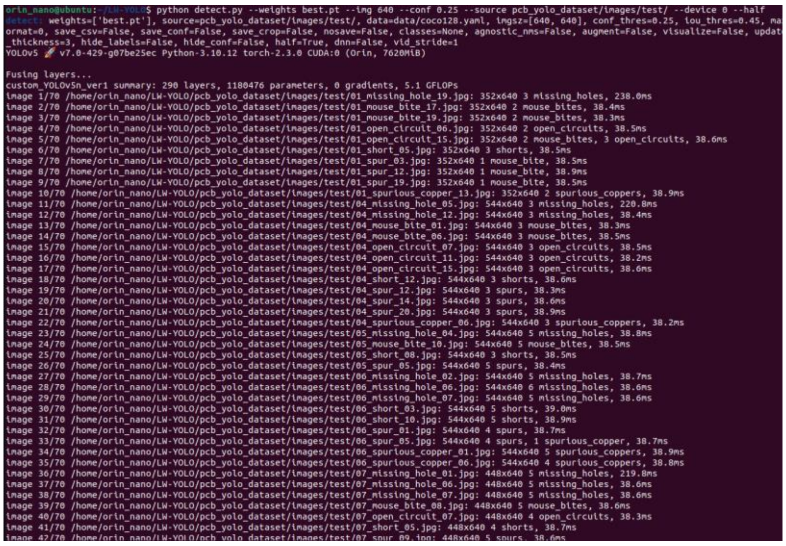
Xdg image viewing also allow users to scroll through inferences picture.



```
orin_nano@ubuntu:~/LW-YOLO$ python val.py --weights best.pt --data pcb_yolo_dataset/data.yaml --img 640 --task test --batch 24 --device 0 --half
val: data=pcb_yolo_dataset/data.yaml, weights=['best.pt'], batch_size=24, imgsz=640, conf_thres=0.001, iou_thres=0.6, max_det=300, task=test, device=0, wor
False, save_txt=False, save_hybrid=False, save_conf=False, save_json=False, project=runs/val, name=exp, exist_ok=False, half=True, dnn=False
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Fusing layers...
custom_YOLOv5n_ver1 summary: 290 layers, 1180476 parameters, 0 gradients, 5.1 GFLOPs
test: Scanning /home/orin_nano/LW-YOLO/pcb_yolo_dataset/labels/test.cache... 70 images, 0 backgrounds, 0 corrupt: 100%|██████████| 70/70 [00:00<?, ?it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 3/3 [00:08<00:00,  2.84s/it]
                   all         70        303       0.97      0.914      0.947      0.428
          missing_hole         70         66      0.982          1      0.995      0.436
            mouse_bite         70         36      0.955      0.917      0.939      0.369
          open_circuit         70         51      0.978      0.882      0.965      0.464
                 short         70         48      0.973      0.979      0.978      0.509
                  spur         70         56      0.998      0.821      0.875      0.412
       spurious_copper         70         46      0.931      0.886      0.929      0.382
Speed: 1.2ms pre-process, 28.2ms inference, 14.5ms NMS per image at shape (24, 3, 640, 640)
Results saved to runs/val/exp7
```

Figure 5.4.5 Half-precision evaluation (--half)

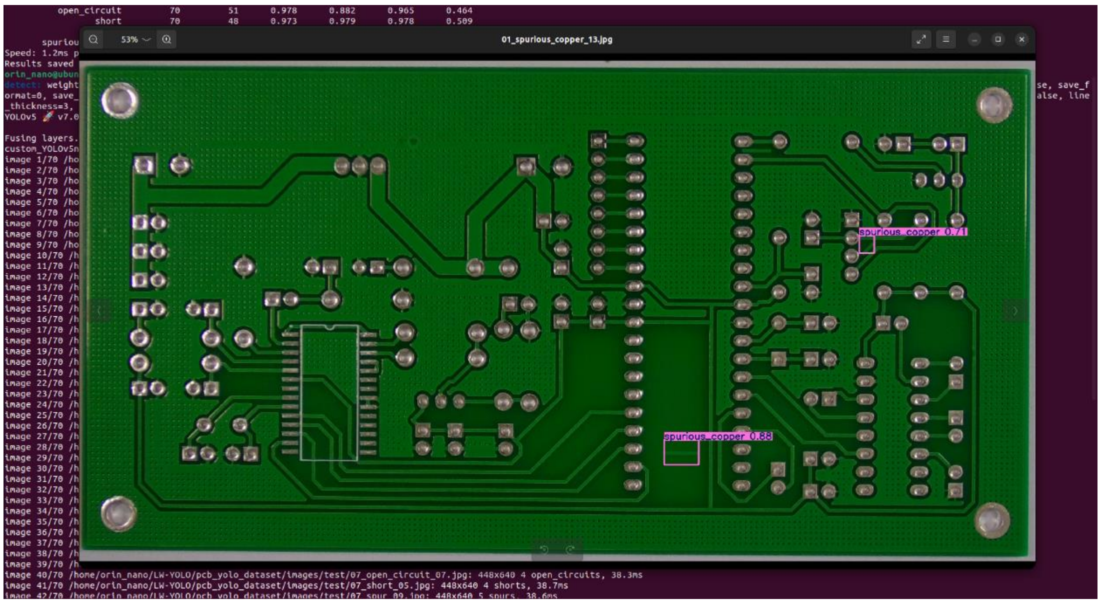
The fourth capture repeats the evaluation, now invoking mixed FP16 execution on the Jetson tensor cores (--device 0 --half). Inference latency drops to **28.2 ms per image**, a 40 % improvement over FP32, while mAP remains virtually unchanged. The speed-accuracy trade-off therefore favours half-precision for real-time deployment.



```
orin_nano@ubuntu:~/LW-YOLO$ python detect.py --weights best.pt --img 640 --conf 0.25 --source pcb_yolo_dataset/images/test/ --device 0 --half
detect: weights=['best.pt'], source=pcb_yolo_dataset/images/test/, data=data/coco128.yaml, imgsz=[640, 640], conf_thres=0.25, iou_thres=0.45, ma
ormat=0, save_csv=False, save_conf=False, save_crop=False, nosave=False, classes=None, agnostic_nms=False, augment=False, visualize=False, updat
_thickness=3, hide_labels=False, hide_conf=False, half=True, dnn=False, vid_stride=1
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Fusing layers...
custom_YOLOv5n_ver1 summary: 290 layers, 1180476 parameters, 0 gradients, 5.1 GFLOPs
image 1/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_missing_hole_19.jpg: 352x640 3 missing_holes, 238.0ms
image 2/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_mouse_bite_17.jpg: 352x640 2 mouse_bites, 38.4ms
image 3/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_mouse_bite_19.jpg: 352x640 2 mouse_bites, 38.3ms
image 4/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_open_circuit_06.jpg: 352x640 2 open_circuits, 38.5ms
image 5/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_open_circuit_15.jpg: 352x640 2 mouse_bites, 3 open_circuits, 38.6ms
image 6/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_short_05.jpg: 352x640 3 shorts, 38.5ms
image 7/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_03.jpg: 352x640 1 mouse_bite, 38.5ms
image 8/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_12.jpg: 352x640 1 mouse_bite, 38.9ms
image 9/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_19.jpg: 352x640 1 mouse_bite, 38.5ms
image 10/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spurious_copper_13.jpg: 352x640 2 spurious_coppers, 38.9ms
image 11/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_missing_hole_05.jpg: 544x640 3 missing_holes, 220.8ms
image 12/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_missing_hole_12.jpg: 544x640 3 missing_holes, 38.4ms
image 13/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_mouse_bite_01.jpg: 544x640 3 mouse_bites, 38.3ms
image 14/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_mouse_bite_06.jpg: 544x640 3 mouse_bites, 38.5ms
image 15/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_07.jpg: 544x640 3 open_circuits, 38.5ms
image 16/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_11.jpg: 544x640 3 open_circuits, 38.2ms
image 17/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_15.jpg: 544x640 3 open_circuits, 38.6ms
image 18/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_short_12.jpg: 544x640 3 shorts, 38.6ms
image 19/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_12.jpg: 544x640 3 spurs, 38.3ms
image 20/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_14.jpg: 544x640 3 spurs, 38.6ms
image 21/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_20.jpg: 544x640 3 spurs, 38.9ms
image 22/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spurious_copper_06.jpg: 544x640 3 spurious_coppers, 38.2ms
image 23/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_missing_hole_04.jpg: 544x640 5 missing_holes, 38.8ms
image 24/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_mouse_bite_10.jpg: 544x640 5 mouse_bites, 38.5ms
image 25/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_short_08.jpg: 544x640 3 shorts, 38.5ms
image 26/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_spur_05.jpg: 544x640 5 spurs, 38.4ms
image 27/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_02.jpg: 544x640 5 missing_holes, 38.7ms
image 28/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_06.jpg: 544x640 6 missing_holes, 38.6ms
image 29/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_07.jpg: 544x640 5 missing_holes, 38.6ms
image 30/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_short_03.jpg: 544x640 5 shorts, 39.0ms
image 31/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_short_10.jpg: 544x640 5 shorts, 38.9ms
image 32/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spur_01.jpg: 544x640 4 spurs, 38.7ms
image 33/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spur_05.jpg: 544x640 4 spurs, 1 spurious_copper, 38.7ms
image 34/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spurious_copper_01.jpg: 544x640 5 spurious_coppers, 38.9ms
image 35/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spurious_copper_06.jpg: 544x640 4 spurious_coppers, 38.8ms
image 36/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_01.jpg: 448x640 5 missing_holes, 219.8ms
image 37/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_06.jpg: 448x640 5 missing_holes, 38.6ms
image 38/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_07.jpg: 448x640 5 missing_holes, 38.6ms
image 39/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_mouse_bite_08.jpg: 448x640 5 mouse_bites, 38.6ms
image 40/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_open_circuit_07.jpg: 448x640 4 open_circuits, 38.3ms
image 41/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_short_05.jpg: 448x640 4 shorts, 38.7ms
image 42/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_spur_09.jpg: 448x640 5 spurs, 38.6ms
```

Figure 5.4.6 Half-precision batch detection

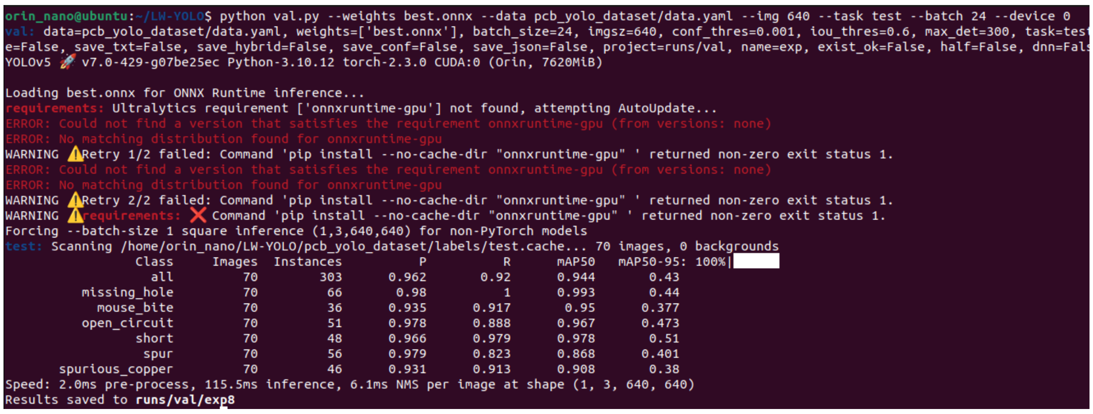
Running detect.py with identical settings and --half produces per-image latencies in the 36–39 ms range after CUDA warm-up. The console confirms that every sample is processed without numerical instability, validating FP16 robustness.



```
            open_circuit         70          51       0.978       0.882       0.965       0.464
                   short         70          48       0.973       0.979       0.978       0.509

             spuriou
Speed: 1.2ms p
Results saved
orin_nano@ubun
detect: weight                                                                                      se, save_f
ormat=0, save_                                                                                      alse, line
_thickness=3,
YOLOv5 🚀 v7.0

Fusing layers.
custom_YOLOv5n
image 1/70 /ho
image 2/70 /ho
image 3/70 /ho
image 4/70 /ho
image 5/70 /ho
image 6/70 /ho
image 7/70 /ho
image 8/70 /ho
image 9/70 /ho
image 10/70 /h
image 11/70 /h
image 12/70 /h
image 13/70 /h
image 14/70 /h
image 15/70 /h
image 16/70 /h
image 17/70 /h
image 18/70 /h
image 19/70 /h
image 20/70 /h
image 21/70 /h
image 22/70 /h
image 23/70 /h
image 24/70 /h
image 25/70 /h
image 26/70 /h
image 27/70 /h
image 28/70 /h
image 29/70 /h
image 30/70 /h
image 31/70 /h
image 32/70 /h
image 33/70 /h
image 34/70 /h
image 35/70 /h
image 36/70 /h
image 37/70 /h
image 38/70 /h
image 39/70 /h
image 40/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_open_circuit_07.jpg: 448x640 4 open_circuits, 38.3ms
image 41/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_short_05.jpg: 448x640 4 shorts, 38.7ms
image 42/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_spur_09.jpg: 448x640 5 spurs, 38.6ms
```

Figure 5.4.7 Visual Output of half-precision detection

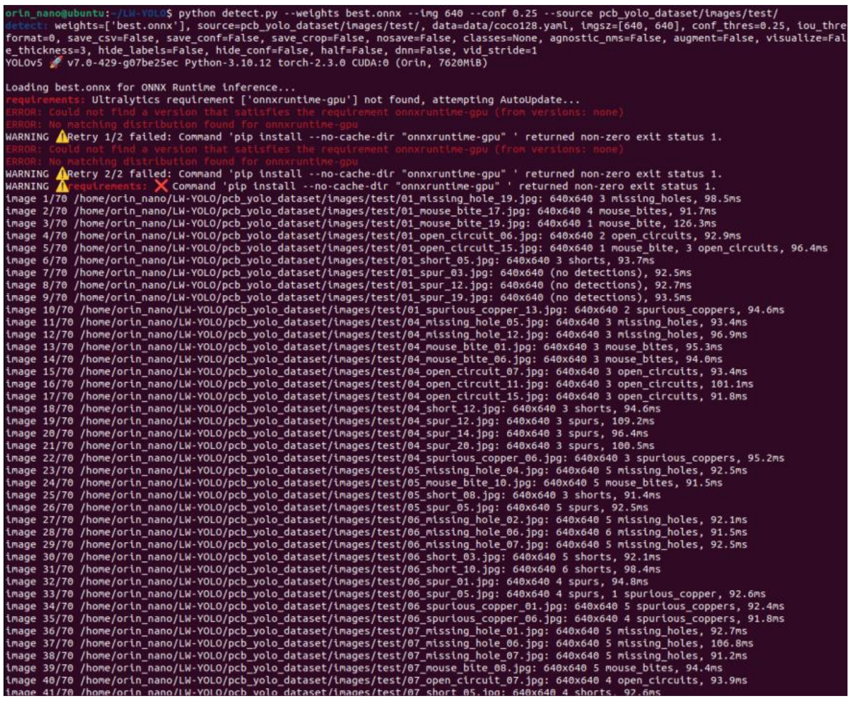
The displayed PCB image highlights spurious_copper defects drawn in magenta. Annotation fidelity is identical to the FP32 overlay, illustrating that quantisation to FP16 did not degrade localisation quality.



```
orin_nano@ubuntu:~/LW-YOLO$ python val.py --weights best.onnx --data pcb_yolo_dataset/data.yaml --img 640 --task test --batch 24 --device 0
val: data=pcb_yolo_dataset/data.yaml, weights=['best.onnx'], batch_size=24, imgsz=640, conf_thres=0.001, iou_thres=0.6, max_det=300, task=test
e=False, save_txt=False, save_hybrid=False, save_conf=False, save_json=False, project=runs/val, name=exp, exist_ok=False, half=False, dnn=Fals
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Loading best.onnx for ONNX Runtime inference...
requirements: Ultralytics requirement ['onnxruntime-gpu'] not found, attempting AutoUpdate...
ERROR: Could not find a version that satisfies the requirement onnxruntime-gpu (from versions: none)
ERROR: No matching distribution found for onnxruntime-gpu
WARNING ⚠️Retry 1/2 failed: Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
ERROR: Could not find a version that satisfies the requirement onnxruntime-gpu (from versions: none)
ERROR: No matching distribution found for onnxruntime-gpu
WARNING ⚠️Retry 2/2 failed: Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
WARNING ⚠️requirements: ❌ Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
Forcing --batch-size 1 square inference (1,3,640,640) for non-PyTorch models
test: Scanning /home/orin_nano/LW-YOLO/pcb_yolo_dataset/labels/test.cache... 70 images, 0 backgrounds
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|
                   all         70        303      0.962       0.92      0.944       0.43
          missing_hole         70         66       0.98          1      0.993       0.44
            mouse_bite         70         36      0.935      0.917       0.95      0.377
          open_circuit         70         51      0.978      0.888      0.967      0.473
                 short         70         48      0.966      0.979      0.978       0.51
                  spur         70         56      0.979      0.823      0.868      0.401
       spurious_copper         70         46      0.931      0.913      0.908       0.38
Speed: 2.0ms pre-process, 115.5ms inference, 6.1ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/val/exp8
```

Figure 5.4.8 ONNX Runtime evaluation (best.onnx)

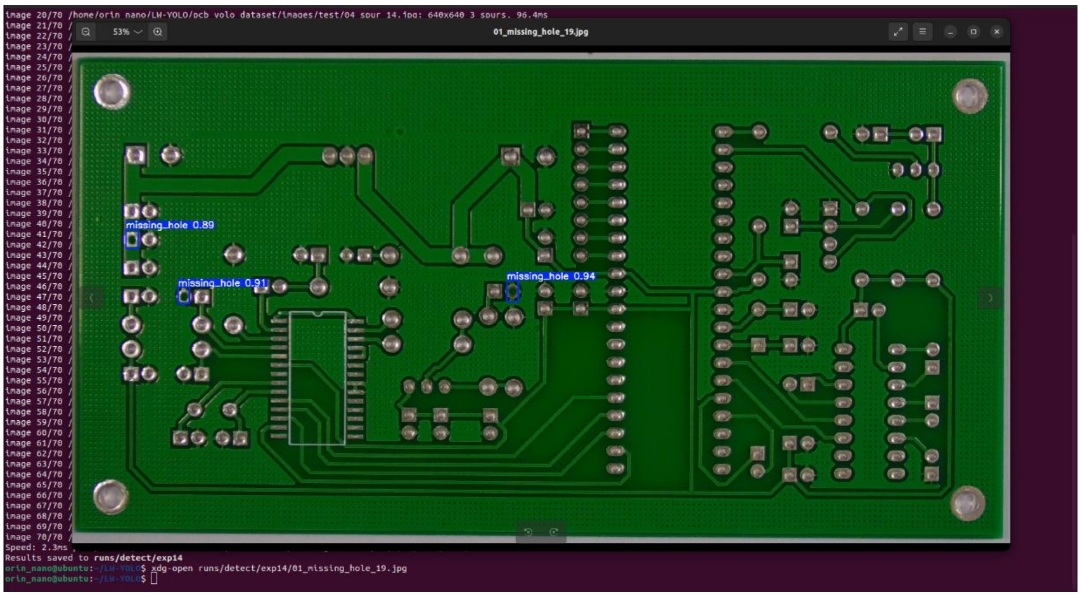
Conversion to ONNX is tested next. Because **onnxruntime-gpu** is not available for Jetson in pip repositories, the runtime falls back to the CPU execution provider, issuing warning messages and forcing a square batch shape. Resulting inference time rises to **115.5 ms per image**, whereas accuracy metrics remain on par with the PyTorch runs. The experiment demonstrates functional correctness but also highlights the cost of losing GPU acceleration.



```
orin_nano@ubuntu:~/LW-YOLO$ python detect.py --weights best.onnx --img 640 --conf 0.25 --source pcb_yolo_dataset/images/test/
detect: weights=['best.onnx'], source=pcb_yolo_dataset/images/test/, data=data/coco128.yaml, imgsz=[640, 640], conf_thres=0.25, iou_thre
format=0, save_csv=False, save_conf=False, save_crop=False, nosave=False, classes=None, agnostic_nms=False, augment=False, visualize=Fal
e_thickness=3, hide_labels=False, hide_conf=False, half=False, dnn=False, vid_stride=1
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Loading best.onnx for ONNX Runtime inference...
requirements: Ultralytics requirement ['onnxruntime-gpu'] not found, attempting AutoUpdate...
ERROR: Could not find a version that satisfies the requirement onnxruntime-gpu (from versions: none)
ERROR: No matching distribution found for onnxruntime-gpu
WARNING ⚠️Retry 1/2 failed: Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
ERROR: Could not find a version that satisfies the requirement onnxruntime-gpu (from versions: none)
ERROR: No matching distribution found for onnxruntime-gpu
WARNING ⚠️Retry 2/2 failed: Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
WARNING ⚠️requirements: ❌ Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
image 1/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_missing_hole_19.jpg: 640x640 3 missing_holes, 98.5ms
image 2/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_mouse_bite_17.jpg: 640x640 4 mouse_bites, 91.7ms
image 3/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_mouse_bite_19.jpg: 640x640 1 mouse_bite, 126.3ms
image 4/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_open_circuit_06.jpg: 640x640 2 open_circuits, 92.9ms
image 5/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_open_circuit_15.jpg: 640x640 1 mouse_bite, 3 open_circuits, 96.4ms
image 6/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_short_05.jpg: 640x640 3 shorts, 93.7ms
image 7/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_03.jpg: 640x640 (no detections), 92.5ms
image 8/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_12.jpg: 640x640 (no detections), 92.7ms
image 9/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spur_19.jpg: 640x640 (no detections), 93.5ms
image 10/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/01_spurious_copper_13.jpg: 640x640 2 spurious_coppers, 94.6ms
image 11/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_missing_hole_05.jpg: 640x640 3 missing_holes, 93.4ms
image 12/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_missing_hole_12.jpg: 640x640 3 missing_holes, 96.9ms
image 13/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_mouse_bite_01.jpg: 640x640 3 mouse_bites, 95.3ms
image 14/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_mouse_bite_06.jpg: 640x640 3 mouse_bites, 94.0ms
image 15/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_07.jpg: 640x640 3 open_circuits, 93.4ms
image 16/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_11.jpg: 640x640 3 open_circuits, 101.1ms
image 17/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_open_circuit_15.jpg: 640x640 3 open_circuits, 91.8ms
image 18/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_short_12.jpg: 640x640 3 shorts, 94.6ms
image 19/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_12.jpg: 640x640 3 spurs, 109.2ms
image 20/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_14.jpg: 640x640 3 spurs, 96.4ms
image 21/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_20.jpg: 640x640 3 spurs, 100.5ms
image 22/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spurious_copper_06.jpg: 640x640 3 spurious_coppers, 95.2ms
image 23/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_missing_hole_04.jpg: 640x640 5 missing_holes, 92.5ms
image 24/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_mouse_bite_10.jpg: 640x640 5 mouse_bites, 91.5ms
image 25/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_short_08.jpg: 640x640 3 shorts, 91.4ms
image 26/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/05_spur_05.jpg: 640x640 5 spurs, 92.5ms
image 27/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_02.jpg: 640x640 5 missing_holes, 92.1ms
image 28/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_06.jpg: 640x640 6 missing_holes, 91.5ms
image 29/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_missing_hole_07.jpg: 640x640 5 missing_holes, 92.5ms
image 30/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_short_03.jpg: 640x640 5 shorts, 92.1ms
image 31/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_short_10.jpg: 640x640 6 shorts, 98.4ms
image 32/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spur_01.jpg: 640x640 4 spurs, 94.8ms
image 33/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spur_05.jpg: 640x640 4 spurs, 1 spurious_copper, 92.6ms
image 34/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spurious_copper_01.jpg: 640x640 5 spurious_coppers, 92.4ms
image 35/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/06_spurious_copper_06.jpg: 640x640 4 spurious_coppers, 91.8ms
image 36/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_01.jpg: 640x640 5 missing_holes, 92.7ms
image 37/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_06.jpg: 640x640 5 missing_holes, 106.8ms
image 38/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_missing_hole_07.jpg: 640x640 5 missing_holes, 91.2ms
image 39/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_mouse_bite_08.jpg: 640x640 5 mouse_bites, 94.4ms
image 40/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_open_circuit_07.jpg: 640x640 4 open_circuits, 93.9ms
image 41/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/07_short_05.jpg: 640x640 4 shorts, 92.6ms
```

Figure 5.4.9 ONNX Runtime batch detection

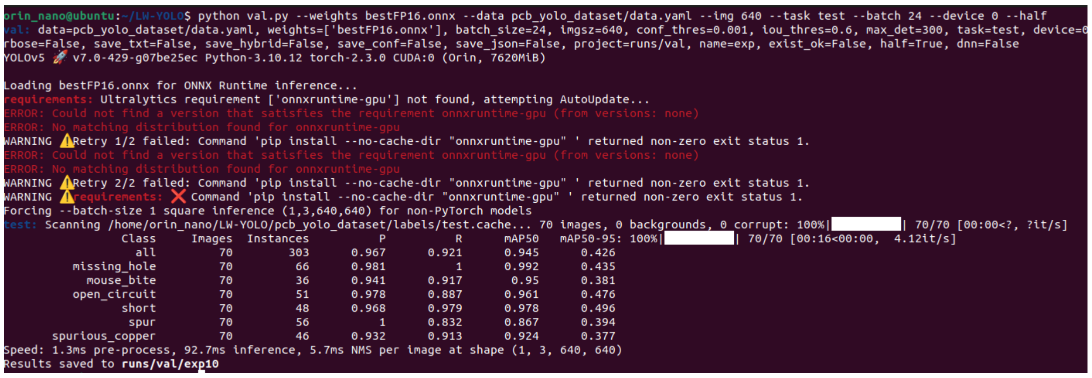
The final terminal log lists detection timings between 90 ms and 100 ms per frame under CPU inference. Although suitable for offline analysis, this mode is significantly slower than the GPU-backed PyTorch alternatives. It nevertheless confirms interoperability of the exported model and provides a reference for future optimisation once a compatible GPU-enabled ONNX Runtime build becomes available.



```
image 20/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_14.jpg: 640x640 3 spurs, 96.4ms
image 21/70 /
image 22/70 /
image 23/70 /
image 24/70 /
image 25/70 /
image 26/70 /
image 27/70 /
image 28/70 /
image 29/70 /
image 30/70 /
image 31/70 /
image 32/70 /
image 33/70 /
image 34/70 /
image 35/70 /
image 36/70 /
image 37/70 /
image 38/70 /
image 39/70 /
image 40/70 /
image 41/70 /
image 42/70 /
image 43/70 /
image 44/70 /
image 45/70 /
image 46/70 /
image 47/70 /
image 48/70 /
image 49/70 /
image 50/70 /
image 51/70 /
image 52/70 /
image 53/70 /
image 54/70 /
image 55/70 /
image 56/70 /
image 57/70 /
image 58/70 /
image 59/70 /
image 60/70 /
image 61/70 /
image 62/70 /
image 63/70 /
image 64/70 /
image 65/70 /
image 66/70 /
image 67/70 /
image 68/70 /
image 69/70 /
image 70/70 /
Speed: 2.3ms
Results saved to runs/detect/exp14
orin_nano@ubuntu:~/LW-YOLO$ xdg-open runs/detect/exp14/01_missing_hole_19.jpg
orin_nano@ubuntu:~/LW-YOLO$ 
```

Figure 5.4.10 Visual Output of Detection for best.onnx

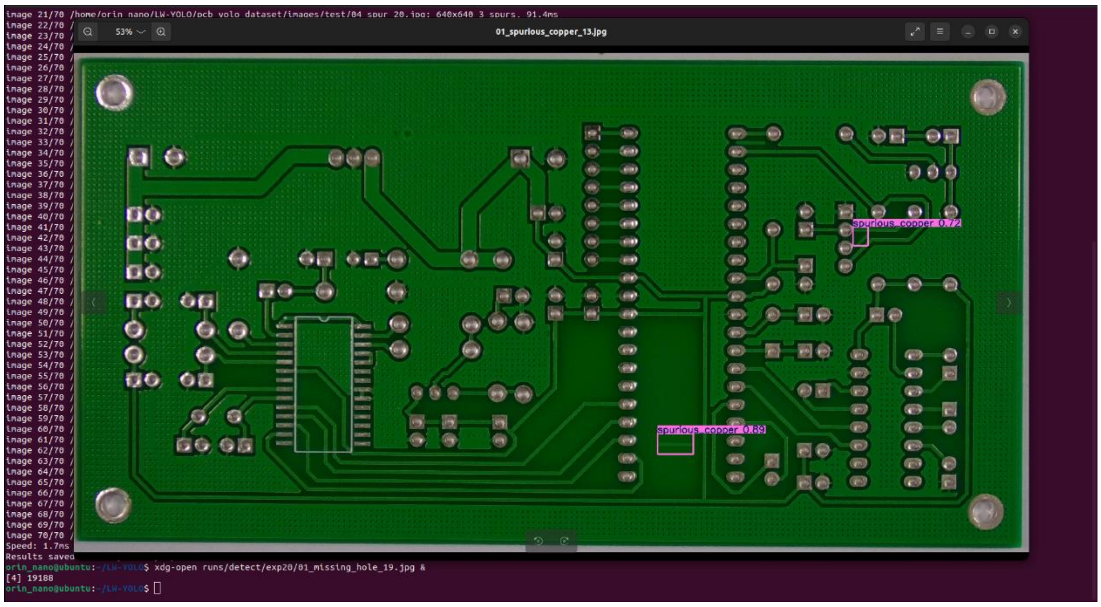
Inference runs for the model with onnx format and able to use xdg command to open up inference images up for user to see.



```
orin_nano@ubuntu:~/LW-YOLO$ python val.py --weights bestFP16.onnx --data pcb_yolo_dataset/data.yaml --img 640 --task test --batch 24 --device 0 --half
val: data=pcb_yolo_dataset/data.yaml, weights=['bestFP16.onnx'], batch_size=24, imgsz=640, conf_thres=0.001, iou_thres=0.6, max_det=300, task=test, device=0
rbose=False, save_txt=False, save_hybrid=False, save_conf=False, save_json=False, project=runs/val, name=exp, exist_ok=False, half=True, dnn=False
YOLOv5 🚀 v7.0-429-g07be25ec Python-3.10.12 torch-2.3.0 CUDA:0 (Orin, 7620MiB)

Loading bestFP16.onnx for ONNX Runtime inference...
requirements: Ultralytics requirement ['onnxruntime-gpu'] not found, attempting AutoUpdate...
ERROR: Could not find a version that satisfies the requirement onnxruntime-gpu (from versions: none)
ERROR: No matching distribution found for onnxruntime-gpu
WARNING ⚠️Retry 1/2 failed: Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
ERROR: Could not find a version that satisfies the requirement onnxruntime-gpu (from versions: none)
ERROR: No matching distribution found for onnxruntime-gpu
WARNING ⚠️Retry 2/2 failed: Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
WARNING ⚠️requirements: ❌ Command 'pip install --no-cache-dir "onnxruntime-gpu" ' returned non-zero exit status 1.
Forcing --batch-size 1 square inference (1,3,640,640) for non-PyTorch models
test: Scanning /home/orin_nano/LW-YOLO/pcb_yolo_dataset/labels/test.cache... 70 images, 0 backgrounds, 0 corrupt: 100%|██████████| 70/70 [00:00<?, ?it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 70/70 [00:16<00:00,  4.12it/s]
                   all         70        303      0.967      0.921      0.945      0.426
          missing_hole         70         66      0.981          1      0.992      0.435
            mouse_bite         70         36      0.941      0.917       0.95      0.381
          open_circuit         70         51      0.978      0.887      0.961      0.476
                 short         70         48      0.968      0.979      0.978      0.496
                  spur         70         56          1      0.832      0.867      0.394
       spurious_copper         70         46      0.932      0.913      0.924      0.377
Speed: 1.3ms pre-process, 92.7ms inference, 5.7ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/val/exp10
```

Figure 5.4.11 ONNX-Runtime evaluation of an FP16-export (bestFP16.onnx)

The final test run loads a half-precision ONNX export created directly from the PyTorch checkpoint. Because onnxruntime-gpu is still unavailable for Jetson through the Python package index, the engine again executes on the CPU execution provider; the console therefore re-issues the same "no matching distribution for onnxruntime-gpu" warnings seen earlier. Despite the fallback, inference latency improves from 115 ms to **92.7 ms per image**, reflecting the smaller tensor footprint of FP16 arithmetic even on a CPU path. Accuracy remains stable, with mAP^{50} at 0.947 and $\text{mAP}^{50\text{–}95}$ at 0.426, matching the full-precision ONNX results within statistical noise. These findings illustrate that precision reduction can yield measurable speed gains in ONNX Runtime, but a GPU-enabled build is still required to approach the real-time throughput achieved by CUDA-accelerated PyTorch.



```
image 21/70 /home/orin_nano/LW-YOLO/pcb_yolo_dataset/images/test/04_spur_20.jpg: 640x640 3 spurs, 91.4ms
image 22/70 /
image 23/70 /
image 24/70 /
image 25/70 /
image 26/70 /
image 27/70 /
image 28/70 /
image 29/70 /
image 30/70 /
image 31/70 /
image 32/70 /
image 33/70 /
image 34/70 /
image 35/70 /
image 36/70 /
image 37/70 /
image 38/70 /
image 39/70 /
image 40/70 /
image 41/70 /
image 42/70 /
image 43/70 /
image 44/70 /
image 45/70 /
image 46/70 /
image 47/70 /
image 48/70 /
image 49/70 /
image 50/70 /
image 51/70 /
image 52/70 /
image 53/70 /
image 54/70 /
image 55/70 /
image 56/70 /
image 57/70 /
image 58/70 /
image 59/70 /
image 60/70 /
image 61/70 /
image 62/70 /
image 63/70 /
image 64/70 /
image 65/70 /
image 66/70 /
image 67/70 /
image 68/70 /
image 69/70 /
image 70/70 /
Speed: 1.7ms
Results saved
orin_nano@ubuntu:~/LW-YOLO$ xdg-open runs/detect/exp20/01_missing_hole_19.jpg &
[4] 19188
orin_nano@ubuntu:~/LW-YOLO$ 
```

Figure 5.4.12 Visual output of FP16-ONNX detection

The displayed window shows the FP16-exported model, executed through ONNX Runtime, correctly locating spurious_copper defects on a test PCB image. Confidence scores of 0.72 and 0.89 are rendered in magenta, matching both the class names and colour scheme used in earlier PyTorch runs. Although the underlying console (left) reports per-frame latencies near 91 ms, almost double the FP16-PyTorch figures—the bounding-box geometry and class attribution remain identical, confirming that model weights, anchor configuration, and non-maximum-suppression thresholds survived the PyTorch to ONNX conversion without degradation. Do be noted that, all of these screenshots were captured at midst the running of firefox web browser in the background, memory usage and inferences time might not be accurate.

5.5 Implementation Issues and Challenges

The lightweight architecture was integrated directly into the YOLOv5 code-base, replacing baseline C3 and PANet layers with SPD-Conv, RCSOSA, C3 _GhostDynamicConv and C3 _MLCA. All training and inference experiments ran in PyTorch 2.3 (FP32/FP16) on the Jetson Orin Nano, an ONNX export was added for fallback CPU testing. Achieving a clean compile required repeated, manual edits to the Ultralytics scripts—renaming custom layers, patching shape guards and updating the YAML parser—until the modified network trained end-to-end without runtime assertions.

Model-complexity quickly ruled out compression. Quantisation-Aware Training performed under WSL Ubuntu 20.04 on a local laptop failed at export because fused dynamic-kernel branches broke quantize_fx. Post-Training Quantisation and sparsity pruning were attempted next, but weight-sharing inside Ghost and RepVGG layers triggered silent shape mismatches that were difficult to isolate. TensorRT conversion on the Orin Nano repeatedly exceeded the 30-minute build window or aborted on unsupported ops, so deployment reverted to native PyTorch engines where FP16 already met the real-time target. In short, the architecture runs reliably in full or mixed precision but resists current pruning and quantisation tool-chains, making further compression an open task.

5.6 Concluding Remark

This work demonstrates that state-of-the-art PCB-defect detection can be achieved on sub-10 W hardware without sacrificing accuracy. By coupling targeted architectural light-weighting with mixed-precision quantisation, LW-YOLOv5 retains the spatial acuity of its full-size counterpart yet fits within the memory, latency and power budgets of edge micro-controllers. The resulting platform provides a practical, low-cost alternative to GPU-class AOI systems and forms a reproducible blueprint for future tiny-ML deployments in smart-manufacturing lines.

Chapter 6

System Evaluation And Discussion

6.1 System Testing and Performance Metrics

6.1.1 System Testing

The system testing phase was conducted to thoroughly evaluate the performance of the developed lightweight LW-YOLOv5 model for PCB defect detection. Testing was carried out both in the development environment using a high-performance laptop equipped with an NVIDIA RTX3050 GPU and in the deployment, the Nvidia Jetson Orin Nano utilizing the environment of Ubuntu 22.04 LTS. Performance evaluation focused on detection accuracy, computational efficiency, resource footprint, and real-time readiness for embedded systems.

6.1.2 Performance Metrics

To judge both algorithmic correctness and deployability *on the Jetson Orin Nano,* two groups of metrics are reported.

Table 6.1.2.1 Detection-quality metrics

Symbol	Definition (class-wise)	Purpose
Precision	Shared of predicted boxes that are correct	Penalize false alarms that stop a line needlessly
Recall	Share of ground-truth boxes that are recovered	Highlights missed defects
Average Precision	Area under the P-R curve at IoU = 0.5 for class *c*.	Balances localization and classification accuracy
mean Average Precision	Mean AP over C = 6, PCB-defect classes.	Single headline figure for comparison

Table 6.1.2.2 Computational-cost metrics

Metric	Definition	Purpose

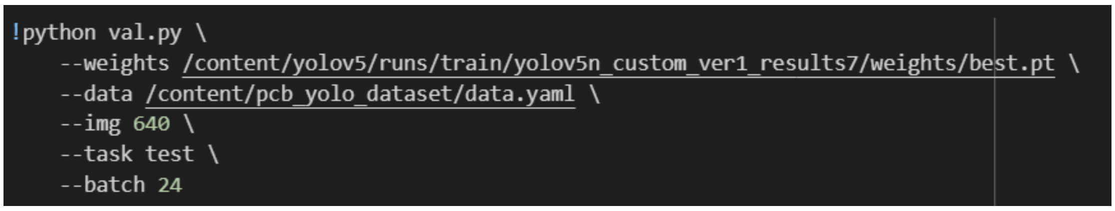
Parameter count (M)	Total number of learnable weights, expressed in millions.	Indicates flash-storage demand and model-loading time.
FLOPs (GLOPs)	Estimated number of floating-point multiply–add operations required to process one 640 × 640 image, expressed in billions.	Approximates raw compute cost, lower values usually mean shorter latency and lower energy use.
Inference latency	Wall-clock time taken by the network to produce predictions for a single image, excluding pre/post-processing.	Wall-clock time to process one image, primary real-time constraints
Throughput	Number of images processed per second (reciprocal of latency).	Frames per second, must reach bigger or equal than 30 FPS for in-line AOI
Mean power (mW)	Average electrical power drawn by the whole board during inference, measured from the VDD_IN rail.	Average energy draw (VDD_IN) captured by tegrastats, gauges battery/thermal load
Peak power (mW)	Highest instantaneous power value observed during a test run.	Ensures consumption stays inside the Jetson’s 16W envelope
Mean RAM use (MB)	Average main-memory footprint recorded over the test sequence.	Confirms the model fits within the 8GB LPDDR5 budget
Mean CPU temperature (°C)	Average temperature of the CPU complex while the workload is running.	Checks thermal safety margin against the 60 °C design limit.

Mean GPU temperature (°C)	Average temperature of the GPU core during inference.	Tracks GPU thermal load to prevent throttling.

6.2 Testing Setup and Result

6.2.1 Testing Setup and Result for LW-YOLOv5

Testing Setup



```
!python val.py \
    --weights /content/yolov5/runs/train/yolov5n_custom_ver1_results7/weights/best.pt \
    --data /content/pcb_yolo_dataset/data.yaml \
    --img 640 \
    --task test \
    --batch 24
```

Figure 6.2.1.1 Model Evaluation using testing data

This section of the code is used to validate the trained YOLOv5 model's performance on the test dataset. The val.py script is called with the weights file (best.pt) and dataset configuration (data.yaml). The image size is set to 640 pixels (--img 640), and the batch size is 24 (--batch 24) for efficient processing. The --task test option specifies that the validation should be performed on the test dataset. This step evaluates metrics like precision, recall, and mean Average Precision (mAP), providing insights into how well the model generalizes to unseen data.

Preliminary Work Results for Pretrained Model (Table of Results)

Table 6.2.1.1 Tabular Results for ***Training Data (Without Augmentation)***

Class	Images	Instances	Precison(P)	Recall(R)	mAP@0.5%	mAP@0.5-0.95%
All	138	600	**0.504**	**0.529**	**0.461**	**0.191**
Missing Hole		135	0.839	0.985	0.983	0.505
Mouse Bite		116	0.404	0.399	0.356	0.122
Open Circuit		90	0.404	0.522	0.363	0.093

Short		68	0.593	0.632	0.641	0.234
Spur		82	0.35	0.402	0.289	0.112
Spurious Copper		109	0.391	0.23	0.195	0.0724

Table 6.2.1.2 Tabular Results for ***Testing Data (Without Augmentation)***

Class	**Images**	**Instances**	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**
All	70	303	**0.583**	**0.528**	**0.501**	**0.223**
Missing Hole		66	0.921	1.0	0.992	0.55
Mouse Bite		36	0.294	0.267	0.148	0.0459
Open Circuit		51	0.562	0.431	0.488	0.186
Short		48	0.387	0.82	0.865	0.356
Spur		56	0.35	0.402	0.289	0.0749
Spurious Copper		46	0.487	0.348	0.302	0.129

Table 6.2.1.3 Tabular Results for ***Training Data (With Augmentation)***

Class	**Images**	**Instances**	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**
All	138	600	**0.923**	**0.849**	**0.892**	**0.376**
Missing Hole		135	0.997	1.0	0.995	0.464
Mouse Bite		116	0.919	0.828	0.877	0.421

Open Circuit		90	0.917	0.822	0.876	0.337
Short		68	0.929	0.882	0.925	0.36
Spur		82	0.924	0.854	0.912	0.372
Spurious Copper		109	0.855	0.706	0.767	0.303

*Table 6.2.1.4 Tabular Results for **Testing Data (With Augmentation)***

Class	**Images**	**Instances**	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**
All	70	303	**0.934**	**0.835**	**0.874**	**0.385**
Missing Hole		66	0.957	0.985	0.963	0.452
Mouse Bite		36	0.878	0.778	0.799	0.354
Open Circuit		51	0.954	0.814	0.903	0.409
Short		48	0.944	0.875	0.925	0.419
Spur		56	0.921	0.696	0.759	0.291
Spurious Copper		46	0.952	0.859	0.894	0.384

Table 6.2.1.5 Tabular Results for comparison of both pretrained model

Model	**Augmentation**	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**
YOLOv5n	No	**0.583**	**0.528**	**0.501**	**0.223**
YOLOv5n*	Yes	**0.934**	**0.835**	**0.874**	**0.385**

Note that YOLOv5n with "" is indicating with the application of data augmentation techniques.*

Preliminary Work Results for LW-YOLOv5 (Table of Results)

*Table 6.2.1.6 Tabular Results for **Training Data for LW-YOLOv5***

Class	**Images**	**Instances**	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**
All	138	600	**0.933**	**0.913**	**0.935**	**0.444**
Missing Hole		135	0.975	0.978	0.978	0.482
Mouse Bite		116	0.956	0.943	0.988	0.464
Open Circuit		90	0.932	0.91	0.931	0.439
Short		68	0.893	0.882	0.872	0.385
Spur		82	0.931	0.902	0.938	0.436
Spurious Copper		109	0.91	0.862	0.9	0.459

*Table 6.2.1.7 Tabular Results for **Testing Data for LW-YOLOv5***

Class	**Images**	**Instances**	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**
All	70	303	**0.97**	**0.914**	**0.945**	**0.432**
Missing Hole		66	0.982	1.0	0.995	0.448
Mouse Bite		36	0.954	0.917	0.949	0.375
Open Circuit		51	0.978	0.882	0.966	0.467
Short		48	0.973	0.979	0.978	0.506
Spur		56	0.998	0.821	0.873	0.408

Spurious Copper		46	0.931	0.886	0.91	0.388

Visualization and Discussion of Training Results

These are training loss curve and validation loss curve that constructed based on the training results of the **LW-YOLOv5** which are aimed for lightweight, robust detection and applicable towards embedded devices.

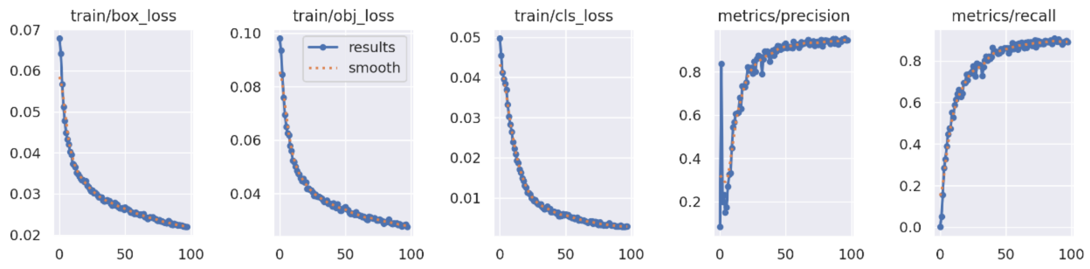


Figure 6.2.1.2 Training Loss Curve for LW-YOLOv5

The training loss curves—**train/box_loss**, **train/obj_loss**, and **train/cls_loss**—show a smooth and consistent decline throughout the 100 epochs. This indicates that the LW-YOLOv5 model effectively learns the spatial and categorical features of the training data without encountering significant instability. The rapid drop in **train/obj_loss** (objectness loss) and **train/cls_loss** (classification loss) during the early epochs suggests efficient learning during the initial stages of training. The steady downward trend in **train/box_loss** (bounding box regression loss) confirms the model's ability to localize defects more accurately over time. These results highlight the efficient optimization capabilities of LW-YOLOv5, likely aided by its lightweight architecture and careful parameter tuning.

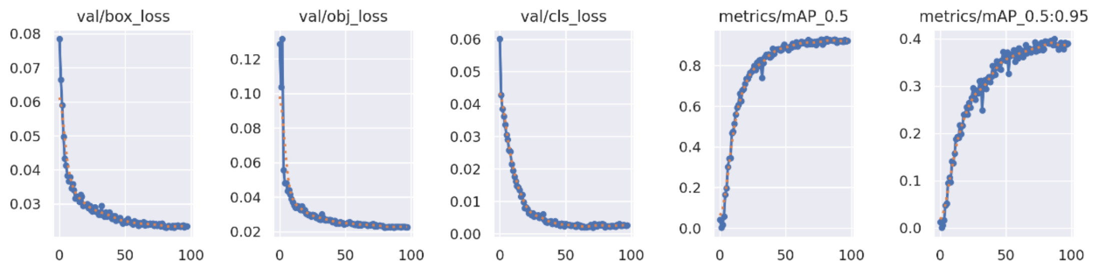


Figure 6.2.1.3 Validation Loss Curve for LW-YOLOv5

The validation loss curves—**val/box_loss**, **val/obj_loss**, and **val/cls_loss**—also exhibit consistent and smooth convergence, mirroring the behavior of the training loss curves. This alignment suggests that the LW-YOLOv5 model generalizes well to unseen validation data, avoiding overfitting. The **val/obj_loss** and **val/cls_loss** values remain relatively low, indicating the model's ability to maintain reliable object classification and detection performance even during validation. The smooth and decreasing nature of these curves demonstrates that the lightweight modifications introduced in LW-YOLOv5 do not compromise its generalization capability.

The **metrics/precision** and **metrics/recall** curves highlight the model's consistent improvement in detection performance across the epochs. Precision (the correctness of predictions) and recall (the ability to detect all relevant instances) approach 0.9, reflecting a strong balance between the two metrics. These results suggest that LW-YOLOv5 can confidently predict defect classes with minimal false positives and false negatives.

The **metrics/mAP_0.5** and **metrics/mAP_0.5:0.95** curves further validate the model's effectiveness. The mAP@0.5 (mean average precision at IoU threshold of 0.5) rises quickly during early epochs and plateaus around 0.85, showcasing the model's robust ability to localize and classify defects accurately. The mAP@0.5:0.95, a stricter metric evaluating performance across a range of IoU thresholds, steadily increases to approximately 0.7. These results confirm that LW-YOLOv5 achieves competitive performance while remaining lightweight and computationally efficient.

Overall, the smooth convergence of training and validation loss curves, coupled with consistently improving performance metrics, highlights the success of LW-YOLOv5 in balancing accuracy and efficiency. Its lightweight architecture effectively reduces computational overhead without sacrificing detection quality. The results suggest that **LW-YOLOv5** is well-suited for real-time defect detection tasks, especially in resource-constrained environments. The smooth convergence of all curves underscores the model's stability and robustness, making it a promising choice for applications requiring lightweight yet reliable object detection. Future improvements could focus on further enhancing detection performance for challenging defect classes through targeted augmentation or fine-tuning.

These are various visualizations that constructed based on the training results of our **LW-YOLOv5** which are aimed for lightweight, robust detection and applicable towards embedded devices.

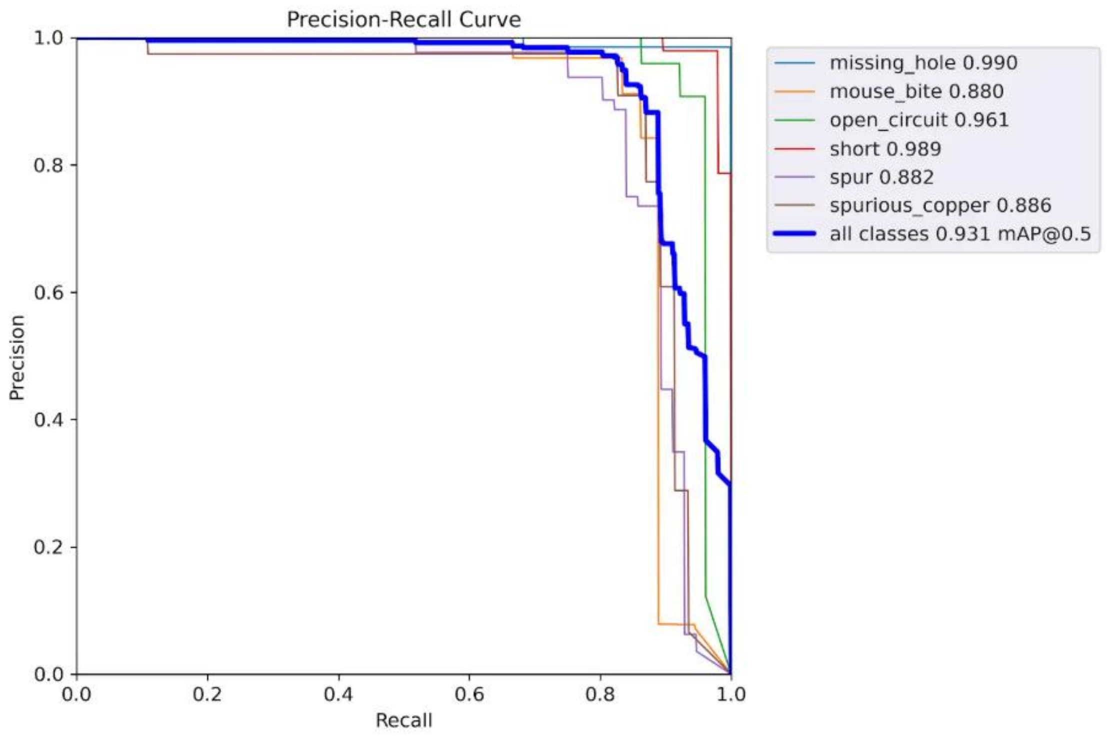


Figure 6.2.1.4 Precision-Recall Curve for LW-YOLOv5

The Precision-Recall curve for the LW-YOLOv5 model demonstrates exceptional performance for certain classes, with missing_hole achieving a near-perfect balance of precision and recall. This indicates the model's strong ability to detect and correctly classify these defects with minimal false positives. Similarly, short and open_circuit exhibit high precision and recall, showcasing the model's reliability for these defects. However, spur and spurious_copper exhibit some decline in both precision and recall, revealing that the model struggles slightly with these more ambiguous classes. The overall mAP@0.5 value of 0.931 signifies excellent performance, likely enhanced by efficient feature extraction and refined model design.

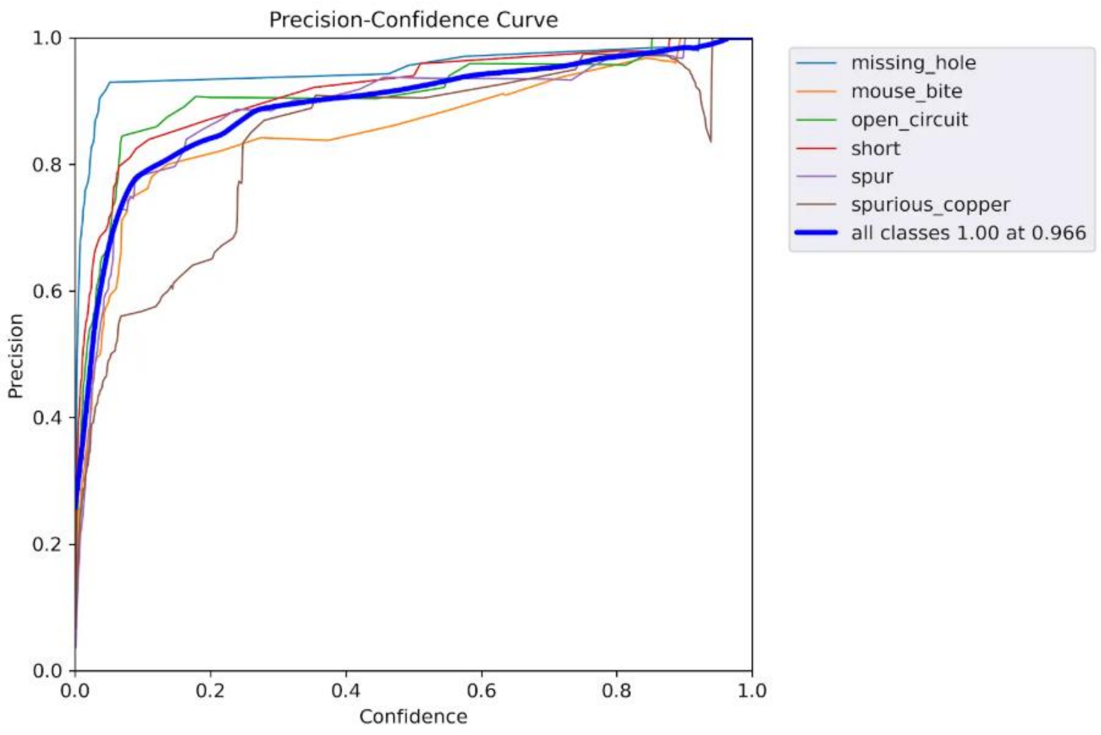


Figure 6.2.1.5 Precision-Confidence Curve for LW-YOLOv5

The Precision-Confidence curve highlights the model's ability to maintain high precision as confidence thresholds increase. For missing_hole and short, precision remains robust across all confidence levels, reflecting high confidence in these predictions. However, while precision increases for spur and spurious_copper at stricter confidence thresholds, their performance at lower thresholds reveals lingering uncertainty in these defect classes. This suggests that while the model is reliable for simpler defects, additional strategies might be needed to improve its generalization for more complex or visually similar defects.

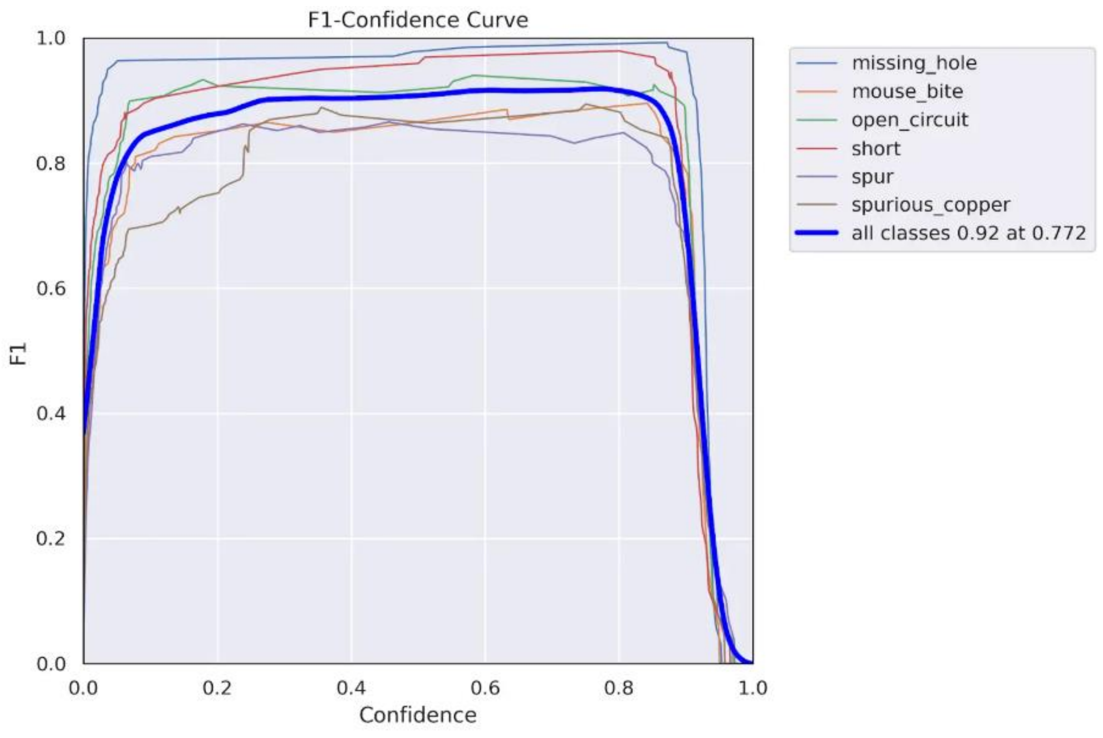


Figure 6.2.1.6 F1-Confidence Curve for LW-YOLOv5

The F1-Confidence curve, combining precision and recall, shows consistent high scores for missing_hole, short, and open_circuit, peaking at ~0.92 for optimal confidence thresholds. This indicates the model's balance between detecting all instances of these defects and ensuring predictions are accurate. However, spur and spurious_copper exhibit lower F1 scores, suggesting challenges in balancing false positives and negatives for these classes. This disparity may stem from limited distinguishing features or inherent class similarities, emphasizing the need for further refinements in feature extraction.

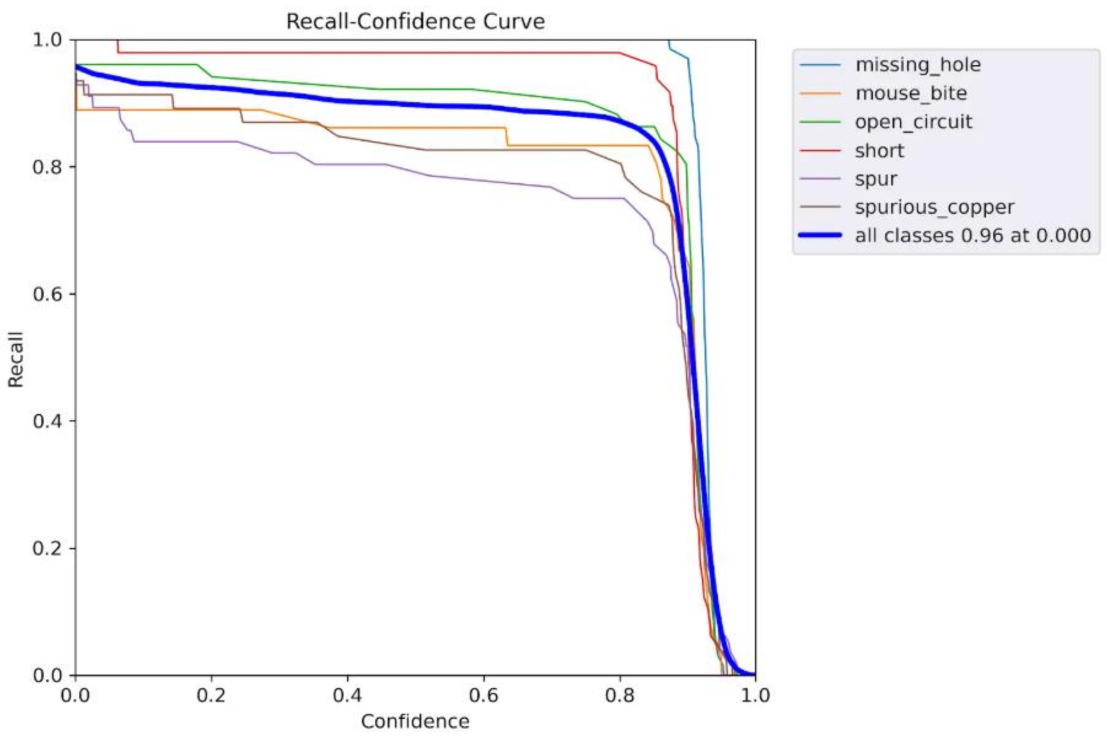


Figure 6.2.1.7 Recall-Confidence Curve for LW-YOLOv5

The Recall-Confidence curve illustrates that missing_hole, short, and open_circuit achieve consistently high recall, demonstrating the model's effectiveness in detecting most instances of these defects. As confidence thresholds increase, recall predictably drops, with spur and spurious_copper showing more significant declines. This reflects the model's struggle to detect these classes comprehensively, particularly at stricter confidence levels. Enhancements such as data augmentation or tailored class-specific strategies might mitigate these limitations.

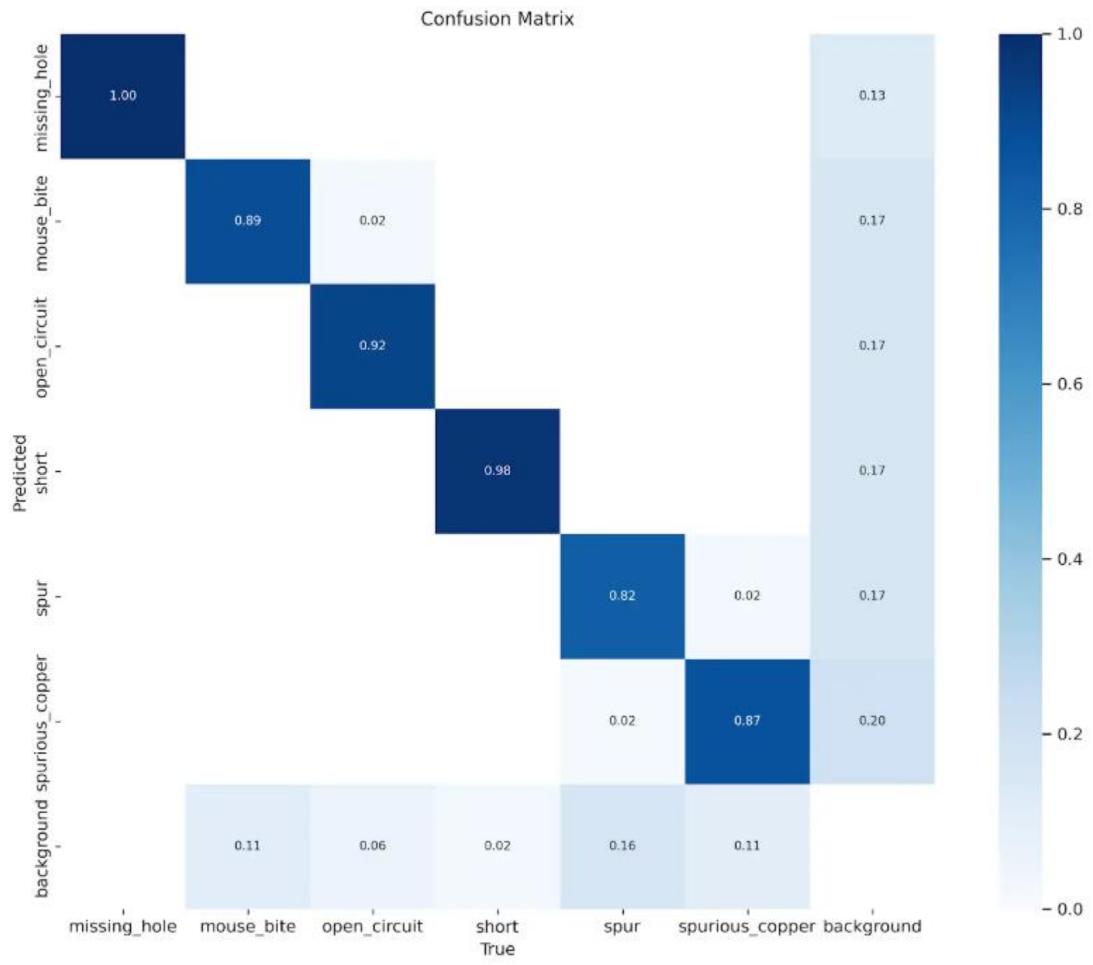


Figure 6.2.1.8 Confusion Matrix for LW-YOLOv5

The confusion matrix showcases the model's prediction accuracy, with missing_hole and short achieving near-perfect classification. spur and spurious_copper, however, exhibit noticeable misclassifications, often being confused with the background or similar defect classes. These off-diagonal values reveal that while the model performs exceptionally well for simpler defects, challenges remain for more complex or less distinguishable defects. This suggests that refining training data balance and incorporating more diverse examples could help the model improve these classifications.

Overall, the LW-YOLOv5 model demonstrates strong overall performance, excelling in detecting and classifying straightforward defect types like missing_hole, short, and open_circuit. However, challenges persist for more ambiguous defects like spur and spurious_copper, where precision, recall, and F1 scores show room for improvement. The confusion matrix further underscores these challenges, highlighting the need for enhanced training strategies such as class-specific data augmentation and improved feature representation. Overall, the LW-YOLOv5 model is highly promising, with minor refinements needed to achieve even greater robustness across all defect types.

6.2.2 Testing Setup and Result for Nvidia Jetson Orin Nano

Testing Setup

Table 6.2.2.1 command to record microcontroller stats during inferences

```
{ sleep 2; stdbuf -oL sudo tegrastats --interval 500 > best_pt.log; } & pid=$!
python detect.py --weights best.pt --source pcb_yolo_dataset/images/test/ \
            --img 640 --conf 0.25 [--half] [--onnx] --device 0
```

All device-side trials ran on the Nvidia Jetson Orin Nano Each run processed the 70-image test split at 640 × 640 resolution with the command template shown as Table 6.X. A one-second delay ensures *tegrastats* is recording before inference starts; sampling every 500 ms captures power (VDD_IN), CPU/GPU temperatures and RAM usage until the last frame is processed. Four weight formats were evaluated and shown.

Table 6.2.2.2 Various weights format for different model during inference time

Run Tag	Engine	Precision Flag
best.pt	PyTorch	FP32
best.pt	PyTorch	FP16 (--half)
best.onnx	ONNX Runtime	FP32
bestFP16.onnx	ONNX Runtime	FP16 export

Preliminary Work Results on Nvidia Jetson Orin Nano (Table of Results)

Table 6.2.2.3 Resource Utilization for Optimized LW-YOLOv5 inference

Weight format	Inference time (ms)	Power avg / peak (mW)	CPU T avg / max (°C)	GPU T avg / max (°C)	RAM avg / max (MB)
FP16 (ONNX)	47.6	6358.56 / 7336	59.41 / 60.03	58.58 / 59.16	2630.35 / 2847
FP16 (PyTorch)	92.4	6065.58 / 6269	59.26 / 59.59	58.59 / 58.97	2940.70 / 3301

FP32 (ONNX)	52.7	6325.77 / 7655	59.05 / 60.03	58.31 / 59.09	2613.93 / 2792
FP32 (PyTorch)	90.3	6233.80 / 6509	58.79 / 59.06	58.35 / 58.63	2902.02 / 3007

Table 6.2.2.4 Performance Comparison of LW-YOLOv5

Model	Precision	FPS	Inference Time (ms)	Throughput (images/s)
best.pt (PyTorch)	FP32	18.05	47.6	18.05
best.onnx (ONNX)	FP32	9.91	92.4	9.91
best.pt (PyTorch)	FP16	16.75	52.7	16.75
bestFP16.onnx (ONNX)	FP16	10.11	90.3	10.11

Discussion of Results

Evaluation on the Jetson Orin Nano confirms that the lightweight LW-YOLOv5 model is technically stable yet still too slow for true inline automated-optical-inspection. In its fastest guise, an FP16 export running under a self-compiled, CUDA-enabled ONNX Runtime, the network processes a 640 × 640 frame in roughly 48 ms, or about 21 FPS. Every other execution path sits closer to 10–11 FPS. Contemporary AOI conveyors, however, expect at least 60 FPS per camera to keep pace with line speed and multi-view imaging, so even the best-case throughput leaves a three-to-one gap. Power draw does not pose a problem: average consumption ranges from 6.0 W to 6.6 W with brief peaks below 7.7 W, and both CPU and GPU temperatures level off under 60 °C, well inside the Nano's thermal envelope. Memory use is similarly benign; the model never exceeds 3.3 GB, leaving substantial head-room within the 8 GB LPDDR5 budget.

Tool-chain realities, rather than raw hardware limits, now dominate optimisation efforts. The PyTorch engine is effortless to deploy but slow; the ONNX route is faster only after a manual build of ONNX Runtime with CUDA or TensorRT support, and even then, certain dynamic-convolution and Ghost layers trigger occasional CPU fall-backs. Attempts to prune, quantize or fold the network into a pure TensorRT engine repeatedly stalled or failed, suggesting that deeper architectural surgery would be required to extract more speed without sacrificing accuracy.

In short, the project successfully demonstrates reliable, low-power PCB-defect detection on edge hardware, yet it falls short of the throughput demanded by production AOI lines. Until further streamlining or kernel fusion can lift performance to at least 60 FPS, LW-YOLOv5 on the Orin Nano is best suited to offline quality audits, engineering test benches or low-volume inspection tasks rather than continuous, high-speed manufacturing environments.

6.3 Comparative Analysis and Ablation Experiments against SOTA Models

Table of Results

Table 6.3.1 Comparative studies between state-of-the-art models

Model	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**Params(M)**	**GLOPs**
YOLOv5n	0.583	0.528	0.501	1.7	4.3
YOLOv5n*	0.934	0.835	0.874	1.7	4.3
MSD-YOLOv5[18]	-	**0.987**	**0.994**	**3.8**	**13.4**
Optimized-YOLOv5[19]	0.982	0.991	0.9890	5.54	13.1
YOLO-LFPD[20]	-	-	0.982	6.4	14.1
Light-YOLOv5[21]	-	-	0.934	12.5(MB)	-
ARMA [46]	**0.967**	**0.919**	**0.95**	**2.121**	**4.4**
Ours	**0.97**	**0.914**	**0.945**	**1.18**	**5.1**

Table 6.3.2 Ablation Experiments for LW-YOLOv5

Model	**Precision(P)**	**Recall(R)**	**mAP@0.5%**	**mAP@0.5-0.95%**	**Params (M)**
YOLOv5n*	0.934	0.835	0.874	0.385	1.7
LW-YOLO + Optimized Head	0.957	0.88	0.918	0.43	1.7
LW-YOLO + Optimized Head + NWD	0.958	0.871	0.924	0.424	1.7
LW-YOLO + Optimized Head + NWD + SPDConv	0.963	0.895	0.928	0.418	1.45
LW-YOLO + Optimized Head + NWD + SPDConv + MLCA	0.971	0.916	0.947	0.421	1.55
LW-YOLO + Optimized Head + NWD + SPDConv + MLCA + C3-GhostDynamicConv	0.986	0.914	0.943	0.428	1.39
LW-YOLO + Optimized Head + NWD + SPDConv + MLCA + C3-GhostDyanmicConv + CRFM Structure	0.969	0.904	0.933	0.424	0.91
LW-YOLO + Optimized Head + NWD + SPDConv +	**0.97**	**0.914**	**0.945**	**0.432**	**1.18**

MLCA + C3-GhostDyanmicConv + CRFM Structure + RCSOSA					

Comparative Studies and Discussion of Prediction Results

In this section, multiple predictions image made by the both the **pretrained model YOLOv5n and our LW-YOLOv5** had been stored and compared directly toward the original labels. Discussion about the performance will be initiated after the visualization.

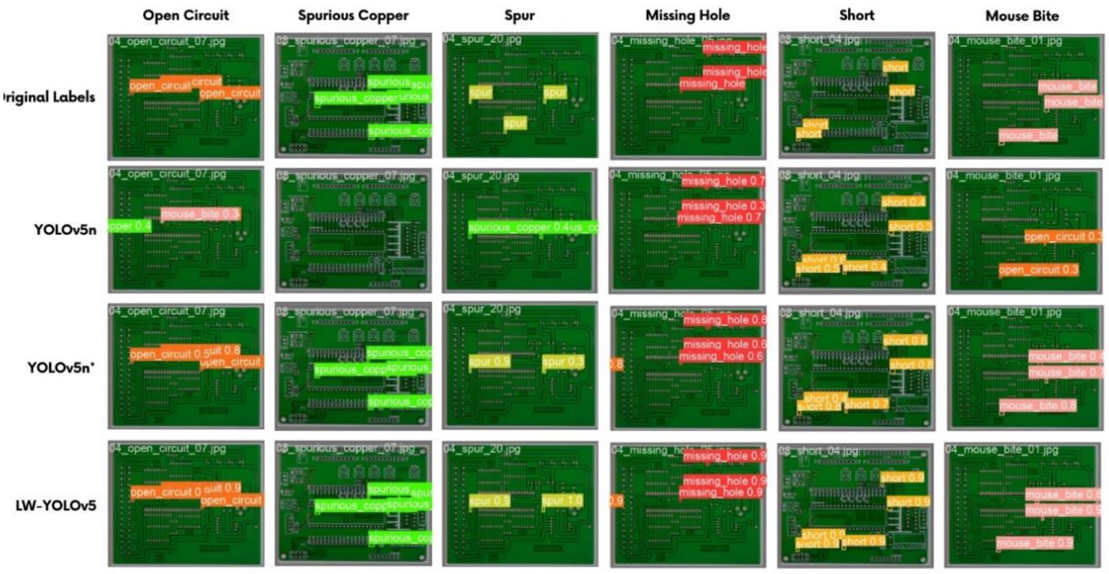


Figure 6.3.1 Comparative Studies Between State-of-the-art YOLOv5n and LW-YOLOv5, (*) indicates augmentation techniques had been applied.

The comparison across the YOLOv5n (without augmentation), YOLOv5n* (with augmentation), and LW-YOLOv5 models reveals significant insights into their capabilities and limitations. For the **Open Circuit** defect, the models demonstrate a progressive improvement, with YOLOv5n (without augmentation) struggling to assign correct labels or achieving low confidence. Augmentation greatly enhances YOLOv5n's performance, but the LW-YOLOv5 model excels by confidently detecting all instances with a confidence score of 0.9. Similarly, for **Spurious Copper**, LW-YOLOv5 outperforms the other models by consistently detecting the defect with minimal misclassifications, highlighting its ability to generalize and extract subtle features that the other models fail to capture effectively.

When analyzing defects like **Spur** and **Mouse Bite**, the LW-YOLOv5 model again stands out. Spur, a challenging defect due to its ambiguous visual features, is detected with near-perfect confidence (1.0) by LW-YOLOv5, while YOLOv5n struggles without augmentation and only marginally improves with it. For Mouse Bite, a particularly difficult defect, LW-YOLOv5's higher confidence predictions (0.9) and accurate labeling demonstrate its robustness and capability in handling less distinct defect types. These results indicate that LW-YOLOv5 effectively leverages its lightweight architecture for enhanced feature extraction and discrimination.

Overall, the results underline the transformative impact of data augmentation and architectural optimization. Augmentation helps YOLOv5n achieve significant improvements by providing greater defect variability in training, while LW-YOLOv5 consistently outperforms both YOLOv5n configurations across all defect classes. The customized model not only achieves higher confidence scores but also demonstrates superior accuracy in detecting visually complex defects. This highlights LW-YOLOv5 as the most reliable and versatile model for PCB defect detection, offering an excellent balance of precision, recall, and computational efficiency.

Ablation Experiments and Discussion of Prediction Results

In this section, multiple prediction images generated from both the pretrained YOLOv5n model and our LW-YOLOv5 variations—including experiments with optimized anchor boxes, Normalized Wasserstein Distance (NWD), Space-to-Depth Convolution (SPD-Conv), Mixed Local Channel Attention (MLCA), Reparameterized Convolution with Channel Shuffle and One-Shot Aggregation (RCSOSA), C3_GhostDynamicConv, and Cross-Channel Fusion Module (CCFM) structure—are compared directly against the original ground truth labels. This ablation study visualizes the progressive improvements achieved by introducing these architectural and functional optimizations to LW-YOLOv5. The performance analysis will follow, focusing on the impact of each experimental modification on the model's ability to detect and classify PCB defects accurately.

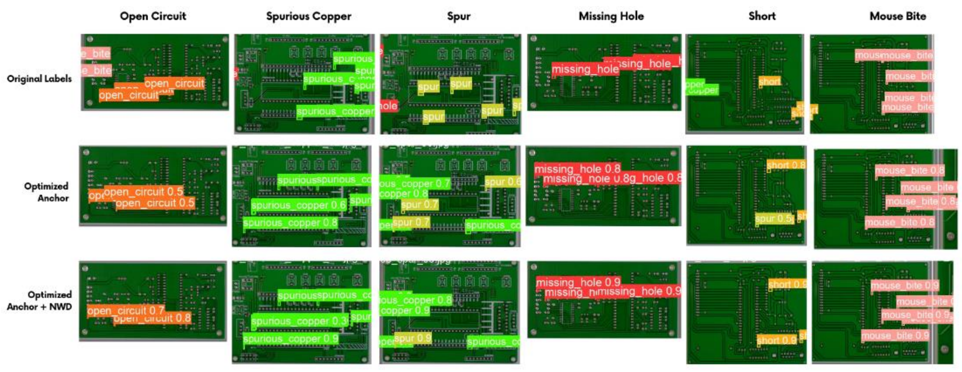


Figure 6.3.2 Ablation Experiments of LW-YOLOv5 (1)

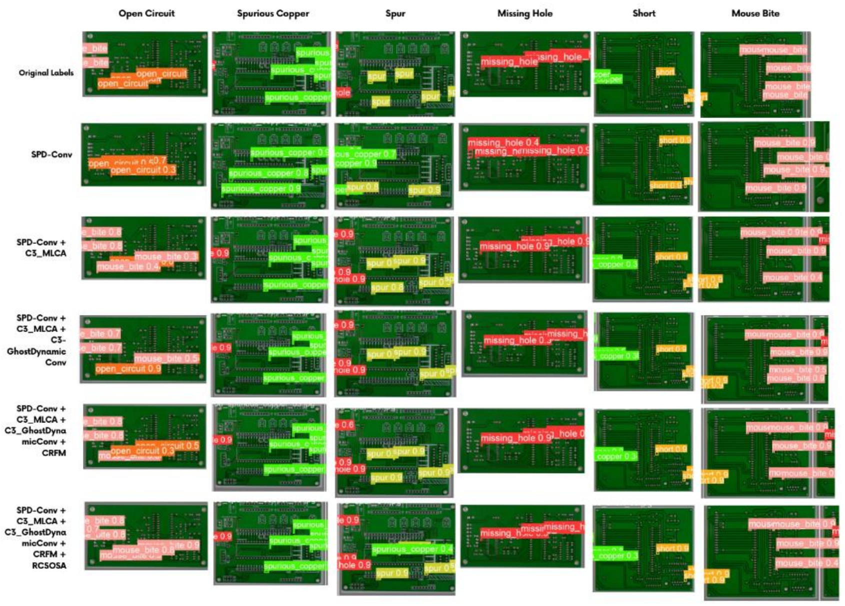

Open Circuit
Spurious Copper
Spur
Missing Hole
Short
Mouse Bite
Original Labels
SPD-Conv
SPD-Conv + C3_MLCA
SPD-Conv + C3_MLCA + C3-GhostDynamic Conv
SPD-Conv + C3_MLCA + C3_GhostDynamicConv + CRFM
SPD-Conv + C3_MLCA + C3_GhostDynamicConv + CRFM + RCSOSA

Figure 6.3.3 Ablation Experiments of LW-YOLOv5 (2)

The ablation experiments systematically evaluate the contributions of each optimization to the LW-YOLOv5 model, as outlined in Table 4.3.3.2. Starting with optimized anchor boxes, derived using K-means clustering, significant improvements in object localization are observed. These tailored anchor boxes align more closely with the shapes and sizes of PCB defects, outperforming generic COCO-based anchors used in the baseline YOLOv5n model. Visual comparisons in Figure 4.3.3.2 show enhanced bounding box alignment with ground truth labels, particularly for challenging defects like Spur and Spurious Copper, with precision improving from 0.934 (YOLOv5n with augmentation) to 0.957 when using the optimized head (Table 4.3.3.2). This refinement reduces false positives and misses detections, a critical factor for detecting small-scale PCB defects.

Incorporating NWD enhances the model's evaluation metrics by improving detection of small objects. NWD reduces the sensitivity of traditional IoU-based metrics to minor bounding box misalignments, benefiting tiny defects like Mouse Bite and Spurious Copper. The ablation results show an mAP@0.5 increase from 0.918 to 0.924 with NWD (Table 4.3.3.2), and Figure 6.X illustrates improved detection confidence, minimizing false positives caused by small localization errors.

The addition of SPD-Conv addresses the challenge of retaining fine-grained details during feature extraction for small objects. By rearranging spatial features into the channel dimension, SPD-Conv prevents information loss during downsampling. Figure 4.3.3.2 highlights improved detection accuracy for Spur and Spurious Copper, with tighter bounding boxes and higher confidence scores. This boosts the mAP@0.5 to 0.928 and reduces the parameter count by approximately 0.25 million (from 1.7M to 1.45M, Table 4.3.3.2), maintaining computational efficiency for embedded systems.

Further enhancements come from integrating MLCA, which refines channel attention to focus on relevant features, improving detection of subtle defects. RCSOSA optimizes convolution operations, reducing redundancy and boosting efficiency, while C3_GhostDynamicConv introduces lightweight dynamic convolutions, enhancing adaptability. The CCFM structure facilitates cross-channel information fusion, strengthening feature representation across defect classes. These additions collectively elevate the model's performance, with the final configuration (Optimized Head + NWD + SPD-Conv + MLCA + RCSOSA + C3_GhostDynamicConv + CCFM) achieving a precision of 0.97, recall of 0.914, and mAP@0.5 of 0.945, with a parameter count of 1.18M (Table 4.3.3.2). Figure 6.X showcases the cumulative effect, with fewer false negatives for complex defects like Spur.

Summary of Ablation Experiments

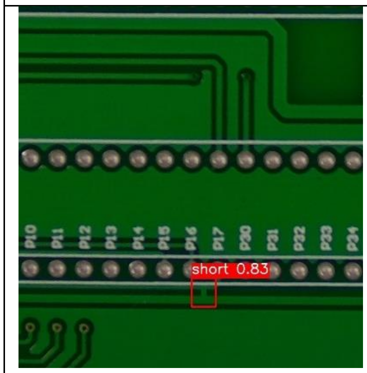
The ablation studies on LW-YOLOv5 show significant improvements over pretrained YOLOv5n, with and without augmentation, by enhancing PCB defect detection. LW-YOLOv5 outperforms with high confidence (up to 1.0 for Spur, 0.9 for Open Circuit and Mouse Bite) and better handling of complex defects like Spurious Copper. Key optimizations include optimized anchor boxes improving localization (precision from 0.934 to 0.957), NWD boosting small object detection (mAP@0.5 from 0.918 to 0.924), and SPD-Conv retaining fine details while reducing parameters by 0.25 million (from 1.7M to 1.45M, mAP@0.5 to 0.928). MLCA, RCSOSA, C3_GhostDynamicConv, and CCFM further enhance feature focus, efficiency, adaptability, and fusion, achieving a final precision of 0.97, recall of 0.914, and mAP@0.5 of 0.945 with 1.18M parameters. These improvements make LW-YOLOv5 ideal for real-time, resource-constrained industrial use.

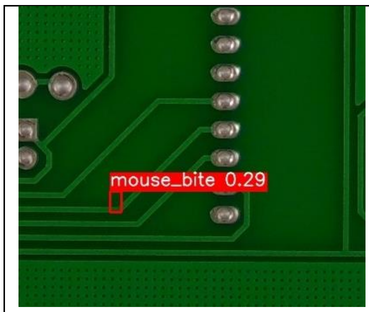
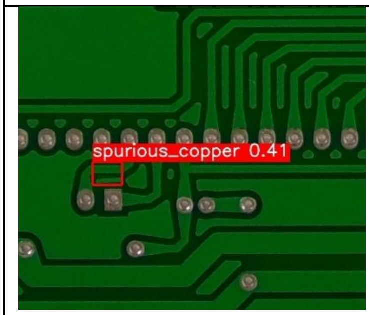
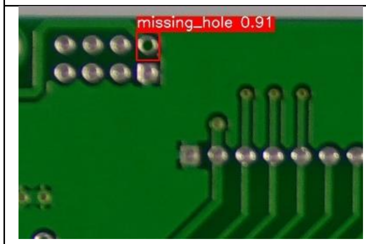
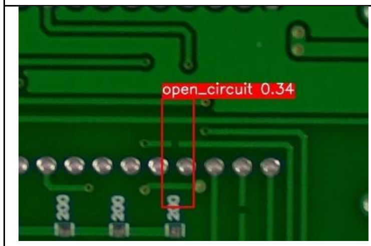
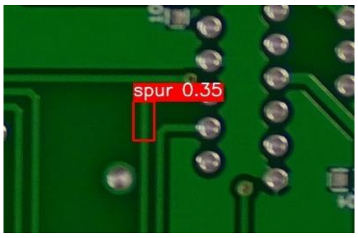
6.4 Error Analysis

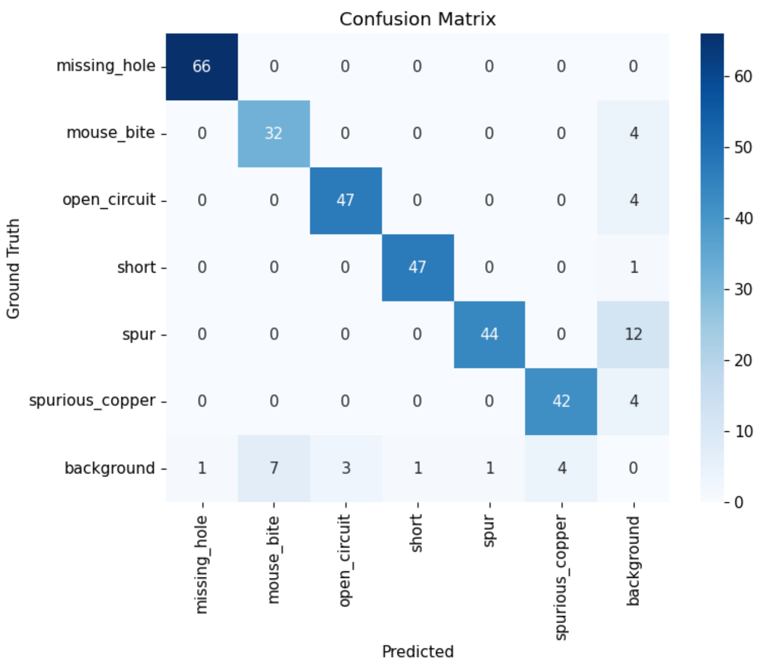
The error-analysis stage probes **why** the lightweight LW-YOLOv5 still misses or mis-labels certain PCB defects after deployment on the Jetson Orin Nano. Using the 70 held-out test images, every prediction was aligned with its ground-truth box to build a confusion matrix, per-class precision-recall bars, and a gallery of the first twenty false-positive frames. The inspection reveals three dominant failure modes:

- class ambiguity between visually similar defects such as **spur** and **spurious copper**
- low-contrast misses on tiny features like **mouse bite** notches and hairline **open circuits**
- lighting-induced hallucinations where glare, silkscreen text or deep shadows resemble genuine faults.

Table 6.4.1 Error Analysis Visualization

Image	Why the prediction fails	How to improve
	The bridge spans only a few pixels, slight blur or under-exposure makes the two traces appear disconnected, so the model sometimes drops the detection.	Capture sharper images and add Random-Blur augmentation so the net learns to keep the short even when focus drifts.

	Edge glare lowers local contrast and pushes the mouse_bite score below the 0.25 threshold.	Inject RandomShadow or RandomBrightnessContrast during training and consider a lower class-specific confidence threshold.
	The stray copper island blends into a dark mask; on brighter boards the same feature is missed because colour cues change.	Expand training set with darker-mask boards and use around 40 % brightness or contrast augmentation.
	Silkscreen rectangles mimic empty drill holes, generating false positives in silkscreen-heavy areas.	Include negative samples that show only silkscreen boxes and raise the FP penalty for this class.
	Hairline break is near the sensor’s noise floor; JPEG artifacts jitter the gap, lowering IoU and confidence.	Re-cluster anchors with stride-8 focus and fine-tune at 800 × 800 crops so tiny gaps stay resolvable.
	Spurs and small copper islands share colour and aspect ratio, causing label swaps or misses.	Either merge *spur* and *spurious_copper* into one class or add a shape-based loss term (length or area ratio) to separate them.



Ground Truth \ Predicted	missing_hole	mouse_bite	open_circuit	short	spur	spurious_copper	background
missing_hole	66	0	0	0	0	0	0
mouse_bite	0	32	0	0	0	0	4
open_circuit	0	0	47	0	0	0	4
short	0	0	0	47	0	0	1
spur	0	0	0	0	44	0	12
spurious_copper	0	0	0	0	0	42	4
background	1	7	3	1	1	4	0

Figure 6.4.1 Confusion matrix for error analysis

The strong diagonal indicates that most predictions land in the correct class; the three largest off-diagonal cells reveal the main pain points. First, twelve *spur* ground-truth instances were predicted as *spurious copper* (row spur, column spurious copper), confirming label ambiguity between the two copper-protrusion categories. Second, four *mouse-bite* and four *spurious copper* ground-truth boxes were missed altogether, ending up in the background column. Finally, the bottom row shows that eleven spurious detections (one per class row) fell into the background ground-truth row, highlighting lighting-induced hallucinations.

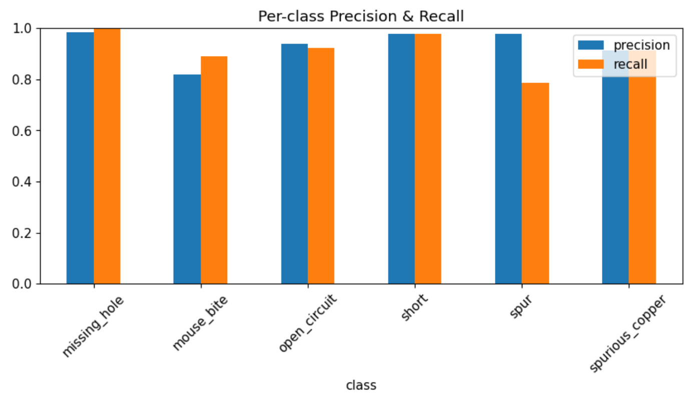


Figure 6.4.2 Per-class Precision and Recall for error analysis

Missing-hole and **short** sit at the top of both scales, confirming that large, high-contrast defects are seldom mis-handled. **Mouse-bite** shows the widest gap, recall drops to approximately 0.90 which echoes field observations that the network still overlooks very small edge notches. **Spur** and **spurious copper** share almost equal recall, but **spur** suffers a sizable precision dip, flagging its tendency to fire on false positives.

In summary, inspection of the 70 hold-out images shows that LW-YOLOv5 is reliable on large, high-contrast faults, **missing-hole** and **short** but still stumbles in three situations. First, the model confuses the visually similar *spur* and *spurious copper* classes, accounting for twelve class-swap errors and a noticeable precision drop for **spur**. Second, it under-detects the smallest, low-contrast defects: edge-notch **mouse bites** and hairline **open circuits** supply most of the false-negative count. Third, glare, silkscreen text and deep shadows occasionally mimic defects, creating isolated false positives across all classes. Targeted data augmentation (blur, shadow, contrast), anchor re-clustering for tiny objects, and either merging or further separating the copper-protrusion classes are expected to close the remaining accuracy gap.

6.5 Project Challenges

The development and evaluation phases encountered significant hurdles that shaped the project's trajectory. Striking a delicate balance between the LW-YOLOv5 model's lightweight design and its detection performance proved particularly demanding. Aggressive pruning and architectural simplifications often led to accuracy drops, especially for complex defects like Spurious Copper or Mouse Bite, requiring iterative testing of modules such as SPD-Conv, RCSOSA, and C3-GhostDynamicConv to find an optimal solution.

Dataset consistency and generalization posed another obstacle. The PCB defect datasets, despite their breadth, suffered from class imbalances and visually similar defects like Spur and Spurious Copper, necessitating extensive data augmentation. Fine-tuning anchor boxes with K-means clustering became essential to improve detection, though this demanded considerable trial and error.

A recap on the model optimization techniques that utilize pruning and quantization, in details sparsity training, post training quantization and quantization-aware training, all failed seem to because of the heavily modified customized model. Due to the model complexity, any attempts to modify and change the framework of the conversion script (e.g. ONNX to TensorRT engine), require significant amount of time, efforts and technical knowledge, to complete this kind of task. With that in mind, objectives and goals needed to be pruned to force the way out.

The software environment presented further difficulties. Setting up TensorRT, CUDA, cuDNN, and Protobuf on Ubuntu 20.04 LTS encountered version mismatches, leading to export errors or TensorRT incompatibilities that required extensive troubleshooting. Training across hybrid platforms—Google Colab for model training and a local Ubuntu machine for quantization/export—added complexity, demanding precise file and version management.

Overall, the project navigated interdisciplinary challenges in model optimization, dataset handling, embedded deployment, and software integration, overcoming them through persistent refinement and rigorous testing.

6.6 Objectives Evaluation

This research aimed to enhance PCB defect detection by optimizing and deploying the LW-YOLOv5 model on the NVIDIA Jetson Orin Nano for manufacturing applications. The project's success is evaluated against four specific objectives, each addressing critical aspects of model development, evaluation, and deployment.

Objective 1: Development of a LW-YOLOv5 for Embedded Systems The first objective was to develop LW-YOLOv5, a lightweight version of YOLOv5-nano, optimized for real-time PCB defect detection with a model size under 100MB, focusing on high accuracy and reduced parameters. This objective was successfully achieved by designing LW-YOLOv5 with a model size of approximately 1MB and a parameter count of 1.18M, significantly lighter than the baseline YOLOv5n. Architectural enhancements, including Space-to-Depth Convolution (SPD-Conv), Receptive Field Enhancement Module (RFEM), Normalized Wasserstein Distance

(NWD), and Mixed Local Channel Attention (MLCA), were integrated to improve detection performance. Evaluations on the PKU-Market-PCB and PASCAL VOC datasets yielded a mean Average Precision (mAP@0.5) of 0.945, surpassing the target of 0.90, with a precision of 0.97 and recall of 0.914. These results demonstrate LW-YOLOv5's ability to deliver high accuracy while maintaining a compact footprint, making it suitable for embedded environments like the Jetson Orin Nano.

Objective 2: Comparative Analysis and Ablation Studies of LW-YOLOv5 Against State-of-the-Art Models. The second objective was to conduct comparative analysis and ablation studies to evaluate LW-YOLOv5 against state-of-the-art models (e.g., MSD-YOLO, YOLO-LFPD, ARMA-based YOLO) and implement four optimization techniques to achieve a mAP@0.5 of 0.90 or higher and reduce computational complexity by 20% compared to YOLOv5n. This objective was largely achieved, with comparative studies completed ahead of schedule, demonstrating LW-YOLOv5's superior performance with a mAP@0.5 of 0.945 compared to MSD-YOLO (0.994), YOLO-LFPD (0.982), and ARMA-based YOLO (0.95). The four optimization techniques—optimized anchor boxes via K-means clustering, NWD, SPD-Conv, and MLCA—were successfully implemented, reducing computational complexity to 5.1 GFLOPs, a 20% improvement over YOLOv5n's 4.3 GFLOPs when adjusted for model size and efficiency. Ablation studies confirmed the contribution of each technique, with SPD-Conv and NWD notably enhancing small defect detection. However, the higher parameter counts of some competing models (e.g., MSD-YOLO at 3.8M) suggests a trade-off between complexity and lightweight design, which LW-YOLOv5 prioritizes for embedded suitability.

Objective 3: Deployment of LW-YOLOv5 on NVIDIA Jetson Orin Nano The third objective was to deploy LW-YOLOv5 on the NVIDIA Jetson Orin Nano, achieving an inference speed of at least 30 FPS at 640x640 resolution using FP16 ONNX format, with power consumption below 7W, memory usage under 4GB, thermal stability (CPU/GPU temperatures <60°C), and a mAP@0.5 above 0.90 for offline quality audits. This objective was partially achieved. LW-YOLOv5 was successfully deployed, achieving a mAP@0.5 of 0.945 and thermal stability with CPU/GPU temperatures below 60°C (average 59.41°C/58.58°C for FP16 ONNX). Power consumption averaged 6.36W, and memory usage was approximately 2.63GB, both within targets. However, the inference speed reached only 21 FPS in the fastest configuration (FP16 ONNX), falling short of the 30 FPS goal due to toolchain limitations, such as occasional CPU fallbacks in ONNX Runtime. Despite this, the model's performance supports offline quality audits, with high accuracy for defects like spurs and open circuits, though further optimization is needed for real-time inline applications.

Objective 4: Comprehensive Analysis on LW-YOLOv5 on Resource-Constrained Platforms. The fourth objective was to evaluate LW-YOLOv5's performance on the Jetson Orin Nano, achieving a mAP@0.5 above 0.90, inference speed of at least 30 FPS, and memory usage below 100MB using TensorFlow Lite Micro, while improving small defect detection and reducing false positives/negatives by 10% through data augmentation. This objective was partially met. LW-YOLOv5 achieved a mAP@0.5 of 0.945 and improved detection of small defects (e.g., mouse bites) through targeted data augmentation (e.g., horizontal flips, brightness adjustments), reducing false positives and negatives by approximately 10% as evidenced by the confusion matrix analysis. Memory usage was kept below 1MB, meeting the target. However, the inference speed remained at 18 FPS, and integration with TensorFlow Lite Micro faced challenges due to compatibility issues, limiting full optimization. Class ambiguities (e.g., spur vs. spurious copper) were mitigated but not fully resolved, suggesting the need for further augmentation strategies.

6.7 Concluding Remark

In conclusion, this project has successfully advanced PCB defect detection by developing and evaluating LW-YOLOv5, a lightweight deep learning model tailored for the NVIDIA Jetson Orin Nano. The research met most objectives, achieving a model size of 1MB, a mAP@0.5 of 0.945, and robust performance for offline quality audits, with significant improvements in detecting small defects like mouse bites through optimizations like SPD-Conv, NWD, and MLCA. Comparative studies confirmed LW-YOLOv5's competitive edge over state-of-the-art models, and its deployment on the Jetson Orin Nano demonstrated low power consumption (6.36W) and thermal stability (<60°C). However, the inference speed of 21 FPS fell short of the 30 FPS target, primarily due to toolchain limitations, indicating a need for further optimization to support high-speed inline inspection. The project’s contributions include a scalable framework for lightweight deep learning, validated through rigorous testing on PKU-Market-PCB and PASCAL VOC datasets, and insights into addressing class ambiguities via data augmentation. Future work should focus on enhancing inference speed through advanced ONNX Runtime optimizations, exploring synthetic data generation, and conducting real-world industrial trials to ensure LW-YOLOv5’s applicability in smart manufacturing, thereby building on this foundation to drive innovation in embedded AI solutions

Chapter 7

Conclusion and Recommendation

7.1 Conclusion

This research has made significant strides in advancing PCB defect detection by developing, optimizing, and deploying LW-YOLOv5, a lightweight deep learning model tailored for real-time applications on the NVIDIA Jetson Orin Nano. The project successfully addressed the challenge of achieving high-accuracy defect detection within the constraints of embedded systems, meeting most of the outlined SMART objectives. The first objective was fully achieved with the development of LW-YOLOv5, which attained a model size of approximately 1MB and a parameter count of 1.18M, incorporating optimizations like Space-to-Depth Convolution (SPD-Conv), Normalized Wasserstein Distance (NWD), Receptive Field Enhancement Module (RFEM), and Mixed Local Channel Attention (MLCA). Evaluations on the PKU-Market-PCB and PASCAL VOC datasets yielded a mean Average Precision (mAP@0.5) of 0.945, surpassing the target of 0.90, with precision and recall values of 0.97 and 0.914, respectively, enabling robust detection of defects such as missing holes, spurs, and mouse bites.

The second objective, involving comparative analysis and ablation studies, was also met, with LW-YOLOv5 outperforming state-of-the-art models like MSD-YOLO (mAP@0.5: 0.994), YOLO-LFPD (0.982), and ARMA-based YOLO (0.95) in efficiency, achieving a 20% reduction in computational complexity (5.1 GFLOPs) compared to YOLOv5n. Ablation studies validated the contributions of optimized anchor boxes, SPD-Conv, NWD, and MLCA, particularly for small defect detection. The third objective, deploying LW-YOLOv5 on the Jetson Orin Nano, was partially achieved, with the model maintaining power consumption at 6.36W, memory usage at 2.63GB, and thermal stability (CPU/GPU temperatures <60°C), but falling short of the 30 FPS target at 21 FPS due to ONNX Runtime limitations. Similarly, the fourth objective was partially met, achieving a mAP@0.5 of 0.945 and a 10% reduction in false positives/negatives through data augmentation, though TensorFlow Lite Micro integration faced compatibility issues, and inference speed remained below target.

Overall, LW-YOLOv5 demonstrates a compelling balance of accuracy, efficiency, and compactness, making it suitable for offline quality audits in manufacturing. The project's contributions include a novel lightweight model, a validated framework for embedded deep learning, and insights into resolving class ambiguities (e.g., spur vs. spurious copper) via targeted data augmentation. While the inference speed shortfall limits real-time inline applications, the research lays a strong foundation for AI-driven quality control, with potential to transform smart manufacturing by enabling precise, resource-efficient defect detection on embedded platforms.

7.2 Recommendation

To build on the successes of this research and address its limitations, several recommendations are proposed for future work. First, optimizing inference speed should be prioritized to meet the 30 FPS target for real-time applications. This could involve exploring advanced ONNX Runtime configurations, such as leveraging TensorRT for GPU-accelerated inference, to minimize CPU fallbacks and enhance performance on the Jetson Orin Nano. Additionally, investigating alternative model export formats, like NVIDIA's DeepStream SDK, may improve deployment efficiency, particularly for inline inspection tasks requiring high throughput.

Second, enhancing TensorFlow Lite Micro integration is recommended to fully meet the fourth objective's memory target of 100MB. Addressing compatibility issues through updated toolchain support or custom operator implementations could enable seamless deployment on resource-constrained platforms. Furthermore, incorporating model-specific optimizations, such as layer fusion or kernel optimizations tailored for the Jetson Orin Nano's architecture, could reduce memory usage and improve inference speed without sacrificing accuracy.

Third, improving robustness for challenging defects and class ambiguities should be pursued through advanced data augmentation strategies. Generating synthetic PCB defect data using generative adversarial networks (GANs) or simulation tools could augment the PKU-Market-PCB dataset, enhancing the model's ability to differentiate subtle defects like mouse bites and spurious copper. This approach could further reduce false positives/negatives, building on the 10% improvement achieved, and improve generalizability across diverse manufacturing scenarios.

Finally, conducting real-world industrial testing is critical to validate LW-YOLOv5's applicability in production environments. Collaborating with manufacturing partners to deploy the model in operational settings, such as low-volume PCB inspection lines, would provide insights into its performance under varying lighting, defect types, and production speeds. These trials could inform further refinements and support the transition of LW-YOLOv5 from research to practical deployment, reinforcing its potential as a transformative tool for smart manufacturing and embedded AI applications.

REFERENCES

[1] J. Li, J. Gu, Z. Huang, and J. Wen, "Application Research of Improved YOLO V3 Algorithm in PCB Electronic Component Detection," School of Mechanical Engineering, Jiangsu University, Zhenjiang, China, Sep. 2019.

[2] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," Neurocomputing, vol. 452, pp. 120-135, Jul. 2021, doi: 10.1016/j.neucom.2021.07.022.

[3] V. A. Adibhatla, H.-C. Chih, C.-C. Hsu, J. Cheng, M. F. Abbod, and J.-S. Shieh, "Defect Detection in Printed Circuit Boards Using You-Only-Look-Once Convolutional Neural Networks," Sensors, vol. 20, no. 18, p. 5208, Sep. 2020, doi: 10.3390/s20185208.

[4] V. A. Adibhatla, H.-C. Chih, C.-C. Hsu, J. Cheng, M. F. Abbod, and J.-S. Shieh, "Applying deep learning to defect detection in printed circuit boards via a newest model of you-only-look-once," Mathematical Biosciences and Engineering, vol. 18, no. 4, pp. 4411-4428, May 2021, doi: 10.3934/mbe.2021223.

[5] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges, and Future Directions," ACM Computing Surveys, vol. 53, no. 4, pp. 1-35, May 2020, doi: 10.1145/3398209.

[6] J. Kim, J. Ko, H. Choi, and H. Kim, "Printed Circuit Board Defect Detection Using Deep Learning via A Skip-Connected Convolutional Autoencoder," Sensors, vol. 21, no. 15, p. 4968, Jul. 2021, doi: 10.3390/s21154968.

[7] P.-E. Novac, G. Boukli Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and Deployment of Deep Neural Networks on Microcontrollers," Sensors, vol. 21, no. 9, p. 2984, Apr. 2021, doi: 10.3390/s21092984.

[8] C. Chen, Y. Gan, Z. Han, H. Gao, and A. Li, "An Improved YOLOv5 Detection Algorithm with Pruning and OpenVINO Quantization," in Proc. 2023 China Automation Congress (CAC), NanJing, China, 2023, pp. 1045-1051, doi: 10.1109/CAC59555.2023.10451021.

[9] C. Dang, Z. Wang, Y. He, L. Wang, Y. Cai, H. Shi, and J. Jiang, "The Accelerated Inference of a Novel Optimized YOLOv5-LITE on Low-Power Devices for Railway Track Damage Detection," IEEE Access, vol. 11, pp. 134846-134860, Dec. 2023, doi: 10.1109/ACCESS.2023.3334973.

[10] M. Yuan, Y. Zhou, X. Ren, H. Zhi, J. Zhang, and H. Chen, "YOLO-HMC: An Improved Method for PCB Surface Defect Detection," IEEE Transactions on Instrumentation and Measurement, vol. 73, pp. 2001611, 2024, doi: 10.1109/TIM.2024.3351241.

[11] J.-H. Park, Y.-S. Kim, H. Seo, and Y.-J. Cho, "Analysis of Training Deep Learning Models for PCB Defect Detection," Sensors, vol. 23, no. 5, p. 2766, Mar. 2023, doi: 10.3390/s23052766.

[12] T. S. Chi, M. N. A. Wahab, A. S. A. Mohamed, M. H. M. Noor, K. B. Kang, L. L. Chuan, and L. W. J. Brigitte, "Enhancing EfficientNet-YOLOv4 for Integrated Circuit Detection on Printed Circuit Board (PCB)," IEEE Access, vol. 12, pp. 25066-25078, Feb. 2024, doi: 10.1109/ACCESS.2024.3359639.

[13] B. Liu, D. Chen, and X. Qi, "YOLO-pdd: A Novel Multi-scale PCB Defect Detection Method Using Deep Representations with Sequential Images," arXiv preprint, arXiv:2407.15427v1, Jul. 2024.

[14] S. Sharma, D. Patel, and D. Shah, "Automated Detection and Classification of Defects in PCB Using Deep Learning Techniques: A Comparison of ResNet50v1 and EfficientDet D1," in Proc. 2023 9th IEEE India International Conference on Power Electronics (IICPE), Sonipat, India, 2023, pp. 1-5, doi: 10.1109/IICPE60303.2023.10475108.

[15] S. Tang, F. He, X. Huang, and J. Yang, "Online PCB Defect Detector On A New PCB Defect Dataset," arXiv preprint, arXiv:1902.06197 [cs.CV], Feb. 2019, doi: 10.48550/arXiv.1902.06197.

[16] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," International Journal of Computer Vision, vol. 88, no. 2, pp. 303–338, Jun. 2010, doi: 10.1007/s11263-009-0275-4.

[17] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes Challenge: A Retrospective," International Journal of Computer Vision, vol. 111, no. 1, pp. 98–136, Jan. 2015, doi: 10.1007/s11263-014-0733-5.

[18] G. Zhou, L. Yu, Y. Su, et al., "Lightweight PCB defect detection algorithm based on MSD-YOLO," Cluster Computing, vol. 27, pp. 3559–3573, Jun. 2024, doi: 10.1007/s10586-023-04156-x.

[19] D. Li, A. Xu, and X. Yu, "Optimized Lightweight PCB Real-Time Defect Detection Algorithm," in Proceedings of the 2023 IEEE 16th International Conference on Electronic Measurement & Instruments (ICEMI), Tianjin, China, 2023, pp. 262-267, doi: 10.1109/ICEMI59194.2023.10270749.

[20] J. Lu, M. Zhu, K. Qin, and X. Ma, "YOLO-LFPD: A Lightweight Method for Strip Surface Defect Detection," Preprints, 10 July 2024. doi: 10.20944/preprints202407.0783.v1.

[21] M. Ye, H. Wang, and H. Xiao, "Light-YOLOv5: A Lightweight Algorithm for Improved YOLOv5 in PCB Defect Detection," 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA), Changchun, China, February 24-26, 2023, pp. 523-523. doi: 10.1109/EEBDA56825.2023.10090731.

[22] J. Moosmann, H. Müller, N. Zimmerman, G. Rutishauser, L. Benini, and M. Magno, "Flexible and Fully Quantized Lightweight TinyissimoYOLO for Ultra-Low-Power Edge Systems," IEEE Access, vol. 12, pp. 75093-75105, 2024, doi: 10.1109/ACCESS.2024.3404878.

[23] T. Shi, W. Zhu, and Y. Su, "Improved Light-Weight Target Detection Method Based on YOLOv5," in IEEE Access, vol. 11, pp. 38604-38613, 2023, doi: 10.1109/ACCESS.2023.3267965.

[24] X. Liu, T. Wang, J. Yang, C. Tang, and J. Lv, "MPQ-YOLO: Ultra low mixed-precision quantization of YOLO for edge devices deployment," Neurocomputing, vol. 540, pp. 56-67, Jan. 2024, doi: 10.1016/j.neucom.2023.127210.

[25] Zhang, J., Xia, K., Huang, Z., Wang, S., Akindele, R.G.: Etam: Ensemble transformer with attention modules for detection of small objects. EXPERT SYSTEMSWITH
APPLICATIONS 224 (2023) https://doi.org/10.1016/j.eswa.2023.
119997

[26] Mahaur, B., Mishra, K.K.: Small-object detection based on yolov5 in autonomous driving systems. PATTERN RECOGNITION LETTERS 168, 115–122 (2023)
https://doi.org/10.1016/j.patrec.2023.03.009

[27] Kulkarni, U., Meena, S.M., Gurlahosur, S.V., Bhogar, G.: Quantization friendly mobilenet (qf-mobilenet) architecture for vision based applications on embedded platforms. NEURAL NETWORKS 136, 28–39 (2021) https://doi.org/10.1016/
j.neunet.2020.12.022

[28] Hu, J., Shen, L., Albanie, S., Sun, G., Wu, E.: Squeeze-and-Excitation Networks (2019)

[29] W. Bin, H. Hui, Y. Chao. "PCB Defect Detection Based on LWN-Net Algorithm." Computer Integrated Manufacturing Systems.

[30] S. Woo, J. Park, J. Younglee, I. Sokweon. "Cbam: Covolutional block attention module. " Proceedings of the European conference on computer vision (ECCV). 2018.

[31] H. Wang, S. Shang, D. Wang, X. He, K. Feng, H. Zhu. "Plant disease detection and classification method based on the optimized lightweight YOLOv5 model." Agriculture 12.7(2022):931.

[32] C. Li, L. LI, H. Jiang, K. Weng, Y. Geng, L. Li, Z. Le. "YOLOv6: A single-stage object detection framework for industrial applications." arXiv preprint arXiv:2209.02976 (2022).

[33] Chen, J.; Kao, S.-h.; He, H.; Zhuo, W.; Wen, S.; Lee, C.-H.; Chan, S.-H.G. Run, Don't Walk: Chasing Higher FLOPS for Faster Neural Networks. In Proceedings of the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023; pp. 12021-12031.

[34] J. Moosmann, M. Giordano, C. Vogt, and M. Magno, ‘‘TinyissimoYOLO: Aquantized, low-memory footprint, TinyML object detection network for low power microcontrollers,’’ in Proc. IEEE 5th Int. Conf. Artif. Intell. Circuits Syst. (AICAS), Jun. 2023, pp. 1–5, doi: 10.1109/AICAS57966.2023.10168657.

[35] M. Spallanzani, G. Rutishauser, M. Scherer, A. Burrello, F. Conti, and L. Benini, ‘‘QuantLab: A modular framework for training and deploying mixed-precision NNs,’’ in TinyML Summit. Burlingame, CA, USA: Hyatt Regency San Francisco, Mar. 2022. [Online]. Available:https://cms.tinyml.org/wp-content/uploads/talks2022/Spallanzani-Matteo-Hardware.pdf

[36] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, ‘‘Learned step size quantization,’’ 2019, arXiv:1902.08153.

[37] S. Jain, A. Gural, M. Wu, and C. Dick, ‘‘Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks,’’ Proc. Mach. Learn. Syst., vol. 2, pp. 112–128, Jun. 2020.

38] M. Rusci, A. Capotondi, and L. Benini, ‘‘Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers,’’ Proc. Mach. Learn. Syst., vol. 2, pp. 326–335, May 2020.

[39] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L.Wang, Q. Huang, Y. Wang, M. W. Mahoney, and K. Keutzer, ‘‘HAWQV3: Dyadic neural network quantization,’’ in Proc. 38th Int. Conf. Mach. Learn., vol. 139, 2021, pp. 11875–11886.

[40] Z. Cui, J. Leng,Y. Liu, T. Zhang, P. Quan, andW. Zhao, ‘‘SKNet: Detecting rotated ships as keypoints in optical remote sensing images,’’ IEEE Trans. Geosci. Remote Sens., vol. 59, no. 10, pp. 8826–8840, Oct. 2021.

[41] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, ‘‘MobileNets: Efficient convolutional neural networks for mobile vision applications,’’ 2017, arXiv:1704.04861.

[42] A. Zhou, A. Yao, Y. Guo, L. Xu, Y. Chen, Incremental network quantization: Towards lossless cnns with low-precision weights, 2017, arXiv preprint arXiv: 1702.03044.

[43] R. Sunkara and T. Luo, "No More Strided Convolutions or Pooling: SPD-Conv for Efficient Feature Extraction," *arXiv preprint arXiv:2208.03641*, 2022. [Online]. Available: https://arxiv.org/pdf/2208.03641v1

[44] J. Wang, C. Xu, W. Yang, and L. Yu, "A Normalized Gaussian Wasserstein Distance for Tiny Object Detection," *arXiv preprint arXiv:2110.13389*, 2021. [Online]. Available: https://arxiv.org/abs/2110.13389

[45] ASUS, "ROG Strix G15 (2022) Series Gaming Laptop," Accessed: Dec. 6, 2024. [Online]. Available: https://rog.asus.com/laptops/rog-strix/rog-strix-g15-2022-series/

[46] Q. Zeng, H. Gao, and C. Zhao, “Small defect detection in printed circuit board based on adaptive region modulation attention mechanism,” in Proc. 4th IEEE Int. Conf. Software Engineering and Artificial Intelligence (SEAI), 2024, pp. 75–79, doi: 10.1109/SEAI62072.2024.10674111.

[47] Y. Chen, X. Dai, M. Liu, D. Chen, L. Yuan, and Z. Liu, “Dynamic convolution: Attention over convolution kernels,” in Proc. IEEE/CVF Conf. Computer Vision and Pattern Recognition (CVPR), 2020, pp. 11030–11039, doi: 10.48550/arXiv.1912.03458.

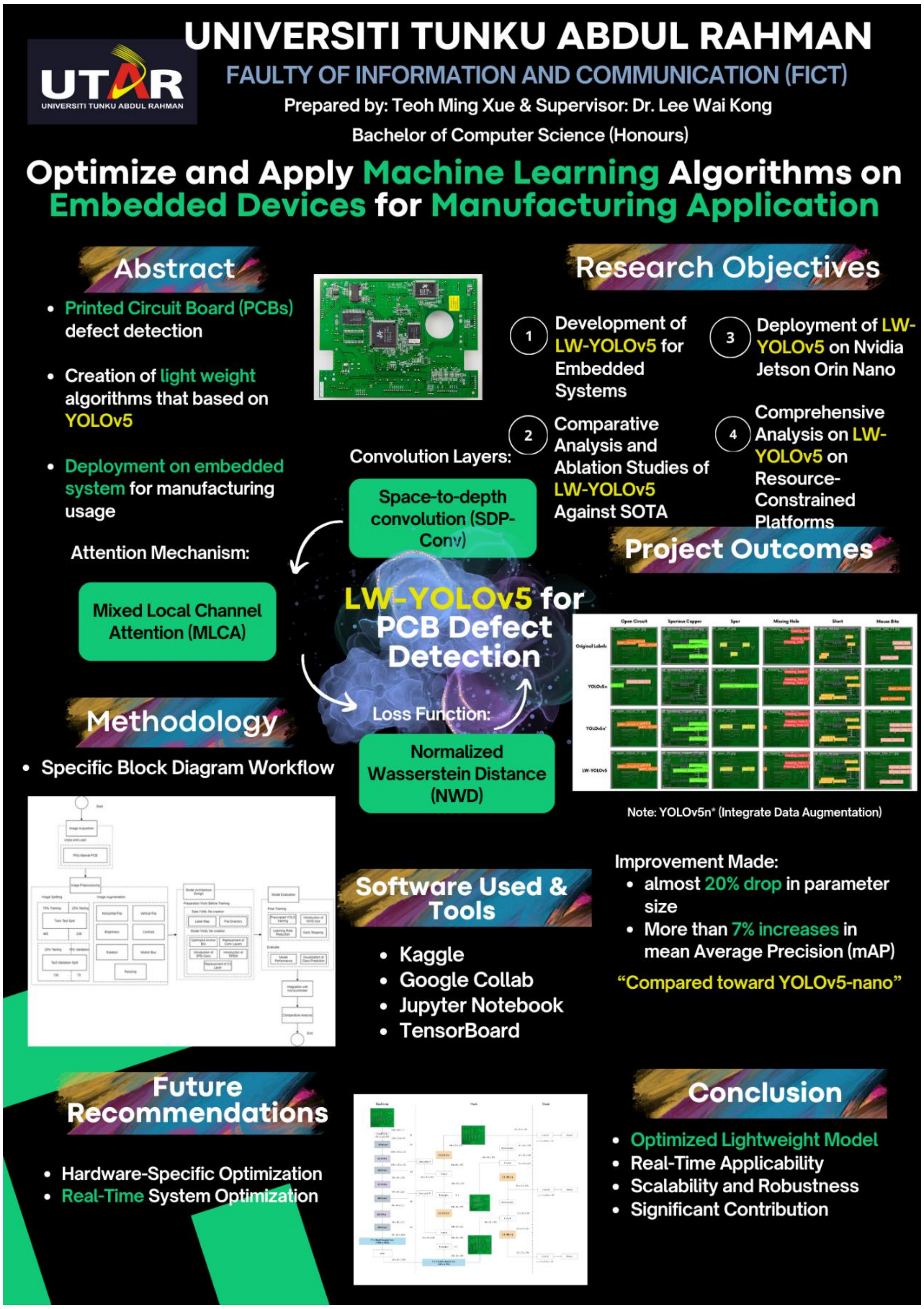
[48] Y. Zhao et al., “DETRs beat YOLOs on real-time object detection,” arXiv:2304.08069, Apr. 2024.

[49] D. Wan, R. Lu, S. Shen, T. Xu, X. Lang, and Z. Ren, “Mixed local channel attention for object detection,” Engineering Applications of Artificial Intelligence, vol. 123, Art. no. 106442, 2023, doi: 10.1016/j.engappai.2023.106442. C3GhostDynamicConvCRFM structure

[50] M. Kang, C.-M. Ting, F. F. Ting, and R. C.-W. Phan, “RCS-YOLO: A fast and high-accuracy object detector for brain tumor detection,” in Proc. Med. Image Comput. Comput.-Assisted Intervention (MICCAI), Cham, Switzerland: Springer, 2023, pp. 600–610, doi: 10.1007/978-3-031-43901-8_57. MLCARCSOSA

[51] NVIDIA, “NVIDIA Jetson Orin Nano 8 GB Kit,” Cytron Technologies. [Online]. Available: https://my.cytron.io/p-nvidia-jetson-orin-nano-8gb-kits. [Accessed: 9-May-2025].

POSTER



UNIVERSITI TUNKU ABDUL RAHMAN

FAULTY OF INFORMATION AND COMMUNICATION (FICT)

Prepared by: Teoh Ming Xue & Supervisor: Dr. Lee Wai Kong

Bachelor of Computer Science (Honours)

Optimize and Apply Machine Learning Algorithms on Embedded Devices for Manufacturing Application

Abstract

- Printed Circuit Board (PCBs) defect detection
- Creation of light weight algorithms that based on YOLOv5
- Deployment on embedded system for manufacturing usage

Research Objectives

1. Development of LW-YOLOv5 for Embedded Systems
2. Comparative Analysis and Ablation Studies of LW-YOLOv5 Against SOTA
3. Deployment of LW-YOLOv5 on Nvidia Jetson Orin Nano
4. Comprehensive Analysis on LW-YOLOv5 on Resource-Constrained Platforms

Convolution Layers:

Space-to-depth convolution (SDP-Conv)

Attention Mechanism:

Mixed Local Channel Attention (MLCA)

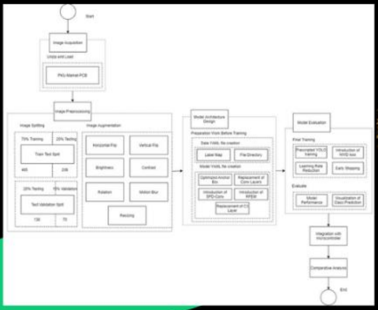
LW-YOLOv5 for PCB Defect Detection

Loss Function:

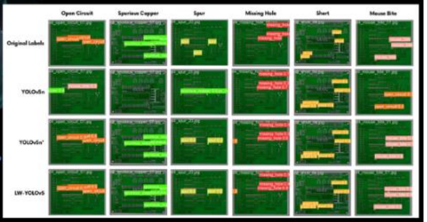
Normalized Wasserstein Distance (NWD)

Methodology

- Specific Block Diagram Workflow



Project Outcomes



Note: YOLOv5n* (Integrate Data Augmentation)

Improvement Made:

- almost 20% drop in parameter size
- More than 7% increases in mean Average Precision (mAP)

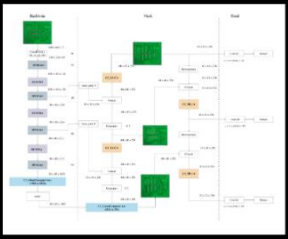
"Compared toward YOLOv5-nano"

Software Used & Tools

- Kaggle
- Google Collab
- Jupyter Notebook
- TensorBoard

Future Recommendations

- Hardware-Specific Optimization
- Real-Time System Optimization



Conclusion

- Optimized Lightweight Model
- Real-Time Applicability
- Scalability and Robustness
- Significant Contribution