Die Defect Detection for Integrated Circuit using Deep Learning Object Detection Techniques

WONG TACK HWA

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Science (Honours) Software Engineering

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

SEPTEMBER 2023

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : 

Name : Wong Tack Hwa

ID No. : 1901610

Date : 05/10/2023

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"Die Defect Detection for Integrated Circuit using Deep Learning Object Detection Techniques"** was prepared by **WONG TACK HWA** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Science (Honours) Software Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature : 

Supervisor : IR. DR. THAM MAU LUEN
ASSISTANT PROFESSOR
LEE KONG CHIAN FACULTY OF ENGINEERING AND SCIENCE
UNIVERSITI TUNKU ABDUL RAHMAN

Date : 05/10/2023

Signature : kckhor

Co-Supervisor : Khor Kok Chin

Date : 05/10/2023

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisors, Dr. Tham Mau Luen and Dr. Khor Kok Chin for their invaluable advice, guidance and their enormous patience throughout the development of the research. In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me encouragement to complete my final year project.

ABSTRACT

Due to advances in semiconductor technology, the complexity of integrated circuit design continues to increase, resulting in ever-smaller defects appearing on these circuits. While some companies still rely on manual inspection for defect detection, these small and hard-to-see defects often lead to high false detection rates due to the human eye's limitations. This study aims to replace manual inspection with an approach that uses object detection to identify subtle defects, which are die rotation and die cracks. The YOLOv5n model is trained to capture ROI and strengthened by incorporating the SAM model to enhance segmentation performance. To address the issue of limited defect images, the StyleGANv2 model is trained to generate extra defect images. The YOLOv7-tiny model has been trained for object detection, with several enhancements made to the network architecture and loss function, pruning is also applied to decrease computational demands. The final model boosts a 3% increase in mAP@0.5 and 2.5% increase in mAP@0.5:0.95, while reducing parameters by 65.34% and GFLOPS by 33.84% compared to the original YOLOv7-tiny model. This study demonstrates that object detection can be an effective method for detecting defects in integrated circuits. The proposed method is able to achieve high accuracy and efficiency.

TABLE OF CONTENTS

LIST OF TABLES **10**

LIST OF FIGURES **12**

LIST OF SYMBOLS / ABBREVIATIONS **17**

LIST OF APPENDICES **20**

CHAPTER

1 INTRODUCTION **21**

1.1 General Introduction 21

1.2 Importance of the Study 22

1.3 Problem Statement 22

1.3.1 Low Accuracy in Human Defect Detection 23

1.3.2 Limitations of human eyes in detecting small defects 23

1.3.3 Limitations of traditional image processing 23

1.3.4 Challenges of over-rejection by AOI 23

1.4 Aim and Objectives 24

1.5 Research Questions 24

1.6 Research Hypothesis 25

1.7 Scope and Limitation of the Study 25

1.8 Proposed Solution 26

2 LITERATURE REVIEW **27**

2.1 Traditional machine learning 27

2.2 Overview of deep learning 28

2.3 Overview of the method used in defect detection 35

2.3.1 Defect detection using image processing technique 35

2.3.2 Defect detection with deep learning: classification approach 40

2.3.3 Defect detection with deep learning: object detection approach 58
2.4 Comparison 71
2.5 Overview of the object detection model 74
2.5.1 R-CNN 74
2.5.2 Fast R-CNN 75
2.5.3 Faster R-CNN 76
2.5.4 SSD 78
2.5.5 YOLO 79
2.6 Conclusion 81
3 METHODOLOGY AND WORK PLAN 83
3.1 Development Tool 83
3.1.1 PyTorch 83
3.1.2 MMAGIC 83
3.1.3 OpenCV 83
3.1.4 Visual Studio Code 84
3.1.5 Anaconda 84
3.1.6 Label Studio 84
3.1.7 Albumentations 84
3.2 Evaluation metric in object detection 84
3.2.1 IoU 85
3.2.2 Precision and recall 86
3.2.3 Precision-Recall curve 86
3.2.4 mAP 87
3.3 Workflow of model training 89
3.3.1 Data preparation 90
3.3.2 Data preprocessing 92
3.3.3 Data augmentation 104
3.3.4 Data annotation 105
3.3.5 Model Training 106
3.3.6 Model Improvements 107
3.3.7 Model Pruning 111
3.3.8 Performance Evaluation 114
3.4 Work Breakdown Structure (WBS) 114

3.5 Gantt Chart 116
4 RESULTS AND DISCUSSION 118
4.1 Comparison of OpenCV, YOLOv5, and SAM in Capturing ROI 118
4.1.1 OpenCV 118
4.1.2 YOLOv5 Segmentation Model 119
4.1.3 SAM 121
4.1.4 Discussion 123
4.2 Comparison of StyleGANv2, StyleGANv2, and Stable Diffusion 123
4.3 Comparison of YOLOv7-tiny, Modified YOLOv7-tiny, and Pruned YOLOv7-tiny 125
4.3.1 Baseline 126
4.3.2 Setting-1 Model 126
4.3.3 Setting-2 Model 127
4.3.4 Setting-3 Model 127
4.3.5 Setting-4 Model 128
4.3.6 Setting-5 Model 128
4.3.7 Setting-6 Model 128
4.3.8 Setting-7 Model 129
4.3.9 Discussion 129
5 CONCLUSION AND RECOMMENDATIONS 131
REFERENCES 133
APPENDICES 139

LIST OF TABLES

Table 2.1:	CNN configuration	40
Table 2.2:	Accuracy and time comparison of CNN and other models on the testing dataset	43
Table 2.3:	k-NN test result for DATASET-NN	43
Table 2.4:	SP&A-Net architecture	46
Table 2.5:	Dataset Description with Feature Description, Defect Types, and Example Images	47
Table 2.6:	Comparison of SP&A-Net and Resnet-50 with ablation study in different blocks	47
Table 2.7:	Analysis of the effect of composition ratio on the performance of SP&A-NET	47
Table 2.8:	Dimensions of the multi-level feature maps extracted at each scale	51
Table 2.9:	Results of performance comparison among various inspection methods	57
Table 2.10:	Distribution of datasets	58
Table 2.11:	Detection and Classification Accuracy of R-CNN	61
Table 2.12:	Defect types in the dataset and number of images per defect type	63
Table 2.13:	Comparison of performance in the test set	63
Table 2.14:	Performance comparison between models with and without SELayer	64
Table 2.15:	Distribution of datasets	66
Table 2.16:	Selected hyperparameter for experiment with default and modified value	66
Table 2.17:	Results of model performance with different model hyperparameters on test images	67

Table 2.18:	Results of model performance with different data augmentation parameters on test images	67
Table 2.19:	Results of ensemble model	68
Table 3.1:	Description of evaluation metrics in object detection	85
Table 3.2:	Table of IC defect types and details	90
Table 4.1:	Metrics of YOLOv5 Segmentation Model	119
Table 4.2:	Computational Costs of YOLOv5 Segmentation Model	119
Table 4.3:	Metrics and Computational cost of YOLOv5n Object Detection Model	121
Table 4.4:	FPS Performance of Various SAM Models	121
Table 4.5:	FID Scores Comparison: StyleGANv2 vs. StyleGANv3	123
Table 4.6:	Metrics and Computational Cost of Various YOLOv7-tiny Models	125

LIST OF FIGURES

Figure 2.1:	Difference between traditional Computer Vision workflow and deep learning workflow	28
Figure 2.2:	Illustration of the concept of perceptron	28
Figure 2.3:	Illustration of perceptron in modern machine learning	29
Figure 2.4:	Single-layer linear regression neural network	30
Figure 2.5:	An MLP with one hidden layer of 5 hidden units	30
Figure 2.6:	Illustration of the visual cortex in the human vision system	32
Figure 2.7:	A simple CNN architecture	32
Figure 2.8:	A simple illustration of convolution operation	33
Figure 2.9:	Max-pooling operation with window shape of $2 * 2$	34
Figure 2.10:	Convolution operation with padding	34
Figure 2.11:	Mask image	35
Figure 2.12:	Reference template	36
Figure 2.13:	(a) Reference image (b) Test image (c) Difference image	36
Figure 2.14:	Wavelet energy in the image	37
Figure 2.15:	20 sub-images of a wafer die	37
Figure 2.16:	(a) Original scratch image on die. (b) Images after applying median and Sobel filters	38
Figure 2.17:	CNN architecture	40
Figure 2.18:	Illustration of defect image cluster analysis results by identified class	41
Figure 2.19:	Five defects in Dataset-TT	42
Figure 2.20:	Misclassified “unknown” defect vs ring-shaped particle image	42

Figure 2.21:	Self-proliferation process	44
Figure 2.22:	Self-attention block	45
Figure 2.23:	SP&A Block	46
Figure 2.24:	Performance evaluation of AEI defect pattern	48
Figure 2.25:	Performance evaluation of ADI defect pattern	48
Figure 2.26:	Performance evaluation of API defect pattern	48
Figure 2.27:	Inspection framework's pipeline	50
Figure 2.28:	Architecture of the MST-GAN	50
Figure 2.29:	Structure of MSCE	51
Figure 2.30:	Structure of CSFF	51
Figure 2.31:	Structure of swin transformer decoder	52
Figure 2.32:	Structure of patch expanding	52
Figure 2.33:	Structure of MSTG	53
Figure 2.34:	Flowchart of the inspection process	54
Figure 2.35:	The visual output obtained from various GAN models	55
Figure 2.36:	Pixel value distribution after applying transposed convolution	56
Figure 2.37:	Pixel value distribution after applying pixel shuffle	56
Figure 2.38:	Performance of the multi-scale weight mask Inspection Framework	57
Figure 2.39:	Workflow of the system	58
Figure 2.40:	Example image of defective semiconductor unit	59
Figure 2.41:	The R-CNN’s classification output reveals the position of the defect(s), defect type, and the level of confidence in the classification. The images (a-d) depict examples of die crack, pinhole, blob, and underfill	60
Figure 2.42:	Modified YOLOv5x architecture	62

Figure 2.43:	SELayer architecture	62
Figure 2.44:	Visualisations of test results for the model with SELayer in the test set	64
Figure 2.45:	Illustration of ensemble model	68
Figure 2.46:	Example of NMS and WBF	68
Figure 2.47:	Overview of R-CNN	74
Figure 2.48:	Overview of Fast R-CNN	75
Figure 2.49:	Overview of Faster R-CNN	76
Figure 2.50:	Overview of RPN	76
Figure 2.51:	Network architecture of SSD	78
Figure 3.1:	Illustration of IoU	86
Figure 3.2:	Precision x Recall curve	87
Figure 3.3:	Workflow for training YOLOv7-tiny object detection model	89
Figure 3.4:	Die defects	91
Figure 3.5:	Flowchart for Capturing ROI of IC Chip Using OpenCV	92
Figure 3.6:	Effects of Smoothed Polygon	92
Figure 3.7:	Workflow for training YOLOv5 segmentation model	94
Figure 3.8:	Flowchart for Inference Process of YOLOv5 Segmentation Model in Capturing the ROI of IC Chip	95
Figure 3.9:	Masks and Bounding Box of IC Chip Obtained from SAM	97
Figure 3.10:	Exported Dataset in JSON Format	97
Figure 3.11:	Exported Dataset in COCO Format	98
Figure 3.12:	Exported Datasets in YOLO Text File Format	98

Figure 3.13:	Hyperparameters Configuration for YOLOv5 Segmentation Model	100
Figure 3.14:	Inference Result with Mosaic Enabled	100
Figure 3.15:	Flowchart for Inference Process of Ensemble Model in Capturing the ROI of IC Chip	102
Figure 3.16:	Masks and Bounding Boxes of Die Rotation Obtained from SAM	105
Figure 3.17:	Hyperparameters Configuration for YOLOv7-tiny Object Detection Model	106
Figure 3.18:	Convolutional Layer vs. CoordConv Layer: A Comparison	108
Figure 3.19:	Structure of GSConv	109
Figure 3.20:	Structure of GS Bottleneck Module and VoV-GSCSP Module	109
Figure 3.21:	Combined Loss Function Code Incorporating CIOU and NWD	110
Figure 3.22:	Workflow for Network Slimming	111
Figure 3.23:	Workflow for LAMP Pruning	112
Figure 3.24:	Gantt chart for Preliminary Planning and Project Planning from 30/1/2023 to 26/3/2023	116
Figure 3.25:	Gantt chart for Project Planning from 27/3/2023 to 18/4/2023	116
Figure 3.26:	Gantt chart for Model Developing from 19/4/2023 to 28/5/2023	117
Figure 3.27:	Gantt chart for Model Developing from 29/5/2023 to 9/7/2023	117
Figure 3.28:	Gantt chart for Model Developing from 10/7/2023 to 20/8/2023	117

Figure 3.29:	Gantt chart for Model Deployment from 21/8/2023 to 2/9/2023	117
Figure 4.1:	Results of OpenCV in Capturing ROI	118
Figure 4.2:	Results of YOLOv5n-seg in Capturing ROI	119
Figure 4.3:	Results of YOLOv5s-seg in Capturing ROI	120
Figure 4.4:	Predicted Masks by YOLOv5n-seg and YOLOv5s-seg for Rotated IC Chip	120
Figure 4.5:	Predicted Masks by Various SAM Models	122
Figure 4.6:	Predicted Masks by SAM_VIT_b and SAM_VIT_l for Rotated IC Chip	122
Figure 4.7:	Generated Die Rotation Images	124
Figure 4.8:	Loss and mAP Graphs for Baseline Model	126
Figure 4.9:	mAP Graph for Setting-1 Model	126
Figure 4.10:	Graph of Ordering of Batch Normalization Parameters for Setting-3 Model	127
Figure 4.11:	Graph of Ordering of Batch Normalization Parameters for Setting-6 Model	128

LIST OF SYMBOLS / ABBREVIATIONS

1-D WT	One-dimensional wavelet transform
2-D WT	Two-dimension Wavelet Transform
ADA	Adaptive Discriminator Augmentation
AdamW	Adam with decoupled weight decay
ADI	After Development Inspection
AEI	After Etch Inspection
AI	Artificial Intelligence
AOI	Automatic Optical Inspection
AP	Average Precision
API	After Polish Inspection
CIoU	Complete Intersection over Union
CNN	Convolutional Neural Network
CSFF	Cross-scale feature fusion
CSP	Cross Stage Partial
DSC	Depth-Wise Separable Convolutions
DT	Destructive Method
DWConv	Depthwise Convolution
E-ELAN	Extended Efficient Layer Aggregation Network
ELAN	Efficient Layer Aggregation Network
FID	Fréchet inception distance
FPN	Feature pyramid network
GC	Global-Context
GFLOPS	Giga Floating Point Operations Per Second
IC	Integrated Circuit
IDE	Integrated Development Environment
IOU	interception over union
IoU	Intersection over union
k-NN	K-nearest neighbours
LoRA	Low Rank Adaptation
mAP	mean Average Precision
MLP	Multilayer Perceptron
MSCE	Multi-scale CNN encoder
MSTG	Multi-scale template generation

MST-GAN	Multi-scale GAN with transformer
NAG	Nesterov Accelerated Gradient
NDT	Non-Destructive Method
NL	Non-Local
NMS	Non-Maximum Suppression
NWD	Normalized Wasserstein Distance
PAN	Path Aggregation Network
PANet	Path Aggregation Network
PE	Patch expanding
RCNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
RLE	Run-length encoding
ROI	Region of interest
RPN	Region Proposal Network
RPN	Region Proposal Network
SAE	Stacked autoencoder
SAM	Segment Anything Model
SC	Standard Convolutions
SE	Squeeze-and-Excitation
SNL	Simplified-Non-Local
SOTA	State-of the art
SP&A-Net	Self-proliferation-and-attention neural network
SPP	Spatial Pyramid Pooling
SPPF	Spatial Pyramid Pooling-Fast
SPPnet	Spatial Pyramid Pooling network
SSD	Single Shot MultiBox Detector
STB	Swin transformer blocks
STD	Swin transformer decoder
SVM	Support Vector Machine
SVM-rbf	Support vector machine with the radial basis kernel
VRAM	Video Random Access Memory
WBF	weighted box fusion
WME	Wafer Map Editor
WTM	Modulus of the wavelet transform

WTMS	Modulus of the wavelet transform sum
YOLO	You Only Look Once
ZF	Zeiler and Fergus

LIST OF APPENDICES

Appendix A:	Comparison of Channel Graphs between Setting-2 and Setting-3 Models	138
Appendix B:	Comparison of Channel Graphs between Setting-2 and Setting-4 Models	138
Appendix C:	Comparison of Channel Graphs between Setting-5 and Setting-6 Models	138
Appendix D:	Comparison of Channel Graphs between Setting-5 and Setting-7 Models	139
Appendix E:	Configuration File for StyleGANv2	140
Appendix F:	Configuration File for StyleGANv3	141
Appendix G:	Configuration File for Stable Diffusion Fine-Tune with LoRA	142
Appendix H:	Model Configuration File for Original YOLOv7-tiny Neck	143
Appendix I:	Model Configuration File for Modified Neck with CoordConv	144
Appendix J:	Model Configuration File for Modified Neck with VoVGSCSP and GSConvs	145
Appendix K:	CoordConv Implementation in YOLOv7-tiny Neck: Code Snippet	146
Appendix L:	GSConv, GSConvs, and VoV-GSCSP Implementation in YOLOv7-tiny Neck: Code Snippet	147

CHAPTER 1

INTRODUCTION

1.1 General Introduction

Thousands to millions of transistors, resistors, and capacitors are interconnected and layered on a thin semiconductor substrate to form an integrated circuit (IC) (Arena Solutions, 2023).

The primary manufacturing processes for integrated circuits can be categorized into three main parts, which are the formation of the silicon wafer, wafer fabrication, and assembly or testing. During the formation stage, silicon crystals are generated and then sliced into thin sections, creating silicon wafers. These wafers serve as the foundational material for constructing integrated circuits (ICs). During the fabrication stage, semiconductor materials undergo a series of complex steps to produce the individual parts of the IC. During the assembly stage, Individual parts of the IC are removed from the wafer and assembled into a package. This typically involves attaching the IC to a substrate or lead frame, enclosing it in ceramic or plastic, and incorporating wire bonds to connect the IC to the leads. Lastly, testing will be performed to ensure each IC performs as planned (Alam and Kehtarnavaz, 2022).

Currently, there are two commonly used inspection procedures for ICs, which can be broadly categorized into DT and NDT. DT involves inspecting the IC without damaging it, while NDT involves inspecting the IC by disassembling the assembled IC. The majority of the industry uses one of the NDT methods, specifically AOI, for inspection. AOI consists of both hardware and software components. The hardware includes a set of image sensors and illumination devices used to capture images of the IC. The software utilizes an inspection algorithm to detect defects from captured images. These inspection algorithms can be classified into two main categories, which are traditional image processing techniques and deep learning techniques (Batool et al., 2021).

Defect detection is extremely important in two critical steps of IC packaging, which are the die attachment stage and the wire bonding stage. Numerous defects such as misplaced or misaligned die, too much or too little epoxy, and missing solder bumps, are found in the wire bonding stage, resulting

in compromised mechanical reliability and affecting the thermal or electrical efficiency of IC. Defect detection is also utilised during the wire bonding phase to identify defects such as broken, missing, or sagging wires, which could disrupt the intended electrical signal transmission from the IC (Alam & Kehtarnavaz, 2022).

1.2 Importance of the Study

One of the standards in the semiconductor manufacturing industry is to ensure the long-term reliability of semiconductors. This not only translates into an enhanced user experience but also extends the lifespan of the products they power. Particularly, semiconductors find applications in critical domains such as medical and military equipment, where sudden failure or breakdown can cause serious safety and reliability issues. Defect detection then becomes an important safeguard to thoroughly screen out defective semiconductors before they enter the market. Premature failure of equipment can also damage a company's reputation and incur warranty costs. Defect detection also plays a crucial role in process improvement and optimization. In the case of a significant number of IC chips exhibiting die rotation problems, it serves as a valuable indicator of potential issues during the die soldering process or with the die soldering machine itself. Defect detection helps to identify the root cause of defects and thus provides an indicator for process improvement. Maintaining a high yield enhances the company's reputation and provides it with a significant competitive advantage. AOI machine is costly, but if object detection can replace the AOI machine's algorithms, the only required equipment is a microscope for capturing IC images and a computer running the object detection software. This could result in significant cost savings. By automating the defect detection process, the company can redirect human resources from defect detection to other critical tasks, ultimately enhancing operational efficiency within the industry.

1.3 Problem Statement

Although there are existing methods for semiconductor defect detection, such as human inspection, AOI machines, and traditional image processing, but these methods still have their own disadvantages and limitations.

1.3.1 Low Accuracy in Human Defect Detection

Companies often use manual methods for defect detection, but training an employee to perform defect detection takes a significant amount of time, or roughly 6 to 9 months of training, to achieve 90% accuracy. However, within 15 months of training, for a variety of reasons, such as increased difficulty due to product evolution, decreased motivation due to stress, or process advances, the accuracy of the manual inspection drops dramatically to about 70% to 85% (Mat Jizat et al., 2021).

1.3.2 Limitations of human eyes in detecting small defects

The human eye cannot detect small defects, and manual inspection is inappropriate when some inspection settings are detrimental to human health (Jin and Chen, 2022). Creating more difficult manufacturing procedures is taken medicine in order to create smaller devices, and the demand for smaller and more complex integrated circuits is also increasing, which leads to an increase in the rate of defects. These defects are usually so miniscule that they are difficult for the human eye to discern (Aryan, Sampath and Sohn, 2018). For example, misalignment of the die by as little as ten micrometres or rotation of the die by less than one degree.

1.3.3 Limitations of traditional image processing

As mentioned, inspection algorithms in AOI can be classified into traditional image processing and deep learning techniques. However, the traditional image processing technique struggles to handle backgrounds with complex textures, noise, or varying lighting conditions (Bhatt et al., 2021). Traditional image processing techniques are highly dependent on feature engineering and require experienced engineers to pre-process the dataset, such as feature selection, noise reduction, feature extraction and selection algorithms. However, this procedure may lead to information distortion or loss, which reduces the accuracy of pattern recognition. (Batool et al., 2021).

1.3.4 Challenges of over-rejection by AOI

Over-reject is a very common problem in AOI if the AOI system is set to be too sensitive or has incorrect programming. The AOI system could classify a lot of

false positives or perfectly functional items as faulty, leading to needless rework or waste, which results in higher expenses, lower efficiency, and lower yield in the IC manufacturing industry.

1.4 Aim and Objectives

This study aims to develop a comprehensive deep learning visual-based inspection approach that based on object detection techniques for detecting die rotation and die crack defects in IC, with the goal of minimising false positive and false negative rates.

Objective:

- Apply data augmentation and StyleGANv2-generated images to enhance segmentation dataset robustness and object detection data diversity. Compare the performance between StyleGANv2, StyleGANv3, and Stable Diffusion.
- Develop a YOLOv5n segmentation model for precise ROI localization and explore an ensemble approach with the YOLOv5n object detection model and Segment Anything Model (SAM) to enhance segmentation accuracy. Compare the performance of capturing ROI between YOLOv5, ensemble model of YOLOv5n and SAM, and OpenCV.
- Train a YOLOv7-tiny model for die crack and rotation detection, incorporating loss function improvements, network architecture improvements, and pruning.

1.5 Research Questions

- What technique can be used to solve the problem of insufficient defect images, and how to increase the robustness of datasets?
- What is the suitable technique to capture the ROI, and how to increase the segmentation accuracy?
- How to improve the accuracy of the object detection model? How to further decrease the computational cost of the model?

1.6 Research Hypothesis

- With datasets containing fewer than 500 images, StyleGANv2 demonstrates superior performance compared to StyleGANv3 and Stable Diffusion.
- SAM outperforms YOLOv5, followed by OpenCV, in effectively capturing and segmenting the ROI of the IC chip.
- Modifying the original YOLOv7-tiny model can lead to an increase in mAP, while pruning can significantly reduce computational costs.

1.7 Scope and Limitation of the Study

The project's scope is to develop a comprehensive deep learning visual-based inspection approach using deep learning techniques to detect die defects in wireless earphone IC from a well-known manufacturer, which include die crack and die rotation. To capture the ROI of the IC chip, the YOLOv5n segmentation model was trained, and an ensemble model based on YOLOv5n object detection and SAM was developed. The object detection model based on YOLOv7-tiny was trained, and several improvements were performed to increase the accuracy. To further decrease the computation cost of YOLOv7-tiny, pruning was performed.

This study only focuses on detecting the two defects in the die during the IC packaging phase, which are die defect and die rotation. It does not consider other defects in other components, such as defects in PCB, LED, and wires and bonds. It does not consider other die defects such as misaligned and missing die. It also does not consider the epoxy defect and foreign molecules on the die. Moreover, this study does not implement image processing techniques such as super-resolution and image reconstruction. Thus, the proposed system might not be able to work as expected when the input image is corrupted, or the image is very blurred. Most importantly, this study examines potential methods for defect detection, but it does not specifically discuss how such methods may be put into practice in real-world production settings. This study's primary objectives are exploring and evaluating these techniques rather than their implementation in the industry.

In terms of development, the limitation of this study is the computational resources available The dataset are provided by the industry

partner, ASPL Malaysia Sdn Bhd. This dataset are confidential and all the deep learning model were trained on local machine to prevent information leakage. The local machine is equipped with RTX3060 mobile and AMD Ryzen 7 5800H CPU. Due to the limited 6GB VRAM available, some complicated networks cannot be trained, such as fine-tuning the stable diffusion model with textual inversion or detail-preserving visual conditions. Some complicated blocks such as attention or transformer modules, can also not be implemented into the original YOLOv7-tiny model. Nevertheless, the pruning repository used in this study does not supply those modules with complicated operations, such as modules that involve shuffle or slicing operations and modules that involve weight sharing.

1.8 Proposed Solution

The proposed solution in this study is to use object detection to detect die rotation and die crack defects. Lim et al. (2023) proposed a PCB defect detection model based on YOLOv5. They proposed a multi-scale FPN based on the original YOLOv5 and modified the original CIOU loss function to increase the performance of YOLOv5 in detecting small defects. As a result, the modified YOLOv5 attained a mAP@0.5:0.95 of 81.20%, marking a 3.65% increase over the original YOLOv5 model. Additionally, Lu et al. (2022) proposed a neural network for IC defect detection that incorporates a SELayer into the original backbone of the YOLO5x model. This modification resulted in a significant increase in accuracy, with a mAP@0.5 of 95.4.

The two studies mentioned above have shown that changing the YOLO model's components could significantly enhance the model's performance, leading to a high accuracy rate in defect detection systems. The additional aspect that needs to be addressed in this study is to find a suitable way to capture the ROI of IC, explore new data augmentation methodologies, find a solution to solve the problem of insufficient images and reduce the computation cost of YOLO by pruning.

CHAPTER 2

LITERATURE REVIEW

2.1 Traditional machine learning

For a task like image classification, one of the huge differences between machine learning and deep learning is that the feature that can represent the picture informative in a discrete area needs to be extracted manually through feature extraction to represent the definitions of each class, which are also known as “bag of words”. An image is classified as containing a specific item if the image contains a sufficient number of features in another image. This process is done by looking up the “bag of words” in another image during the inference process (Mahony et al., 2019).

In the traditional machine learning process, filtering the unimportant features and extracting those features that best describe the characteristic of an object in the image is necessary. This process often requires an experienced computer vision engineer, and it takes a very long time as engineers are needed to fine-tune the parameters in order to extract those important features. This is one of the main challenges of traditional machine learning (Mahony et al., 2019).

In a neural network, no feature extraction process is required as the neural network can identify the underlying patterns of each class and automatically extract those important features. In short, the challenge of traditional machine learning has been solved by deep learning as the neural network now carries out the feature extraction process, and no experienced computer vision engineer is needed anymore. Figure 2.1 shows how deep learning has eliminated the process of feature extraction.

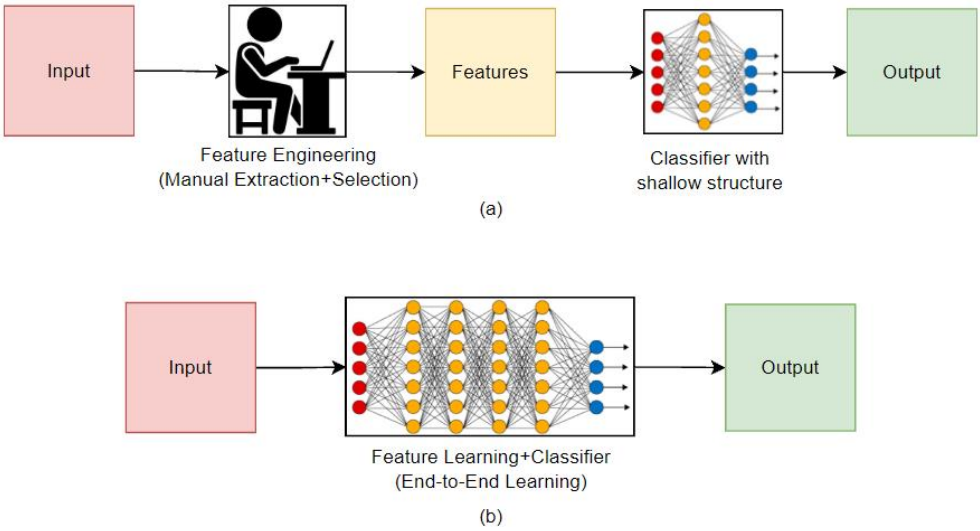


Figure 2.1: Difference between traditional Computer Vision workflow and deep learning workflow (adopted from Mahony et al. (2019))

2.2 Overview of deep learning

Perception, which is the origin model of the neural network, was introduced by Frank Rosenblatt (1957). The idea of perception is based on the biological neuron. Figure 2.2 illustrates the concept of the perceptron, which takes inputs from other neurons, processes them, and then emits an output signal. The retina unit sends its data to the projection unit, which then transmits it to the association unit. The signal will be fired if the total signal is equal to or greater than the association unit threshold (Wang & Raj, 2017).

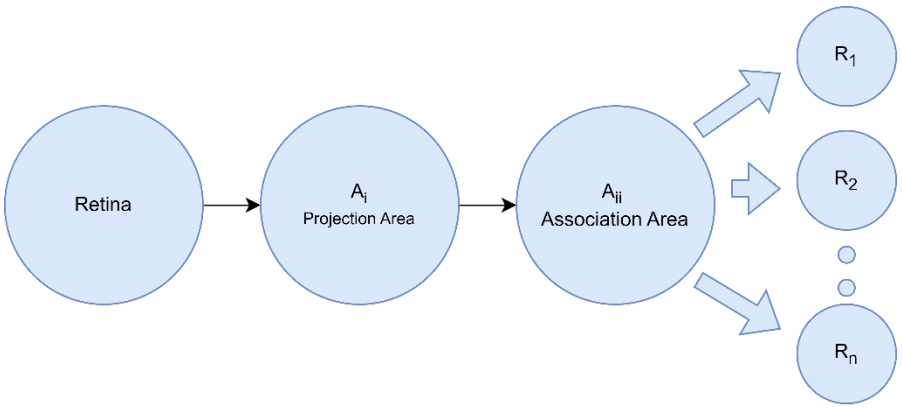


Figure 2.2: Illustration of the concept of perceptron (adopted from Wang & Raj (2017))

In most of the modern neural network, the association unit is often to be ignored, as shown in Figure 2.3 (Wang & Raj, 2017).

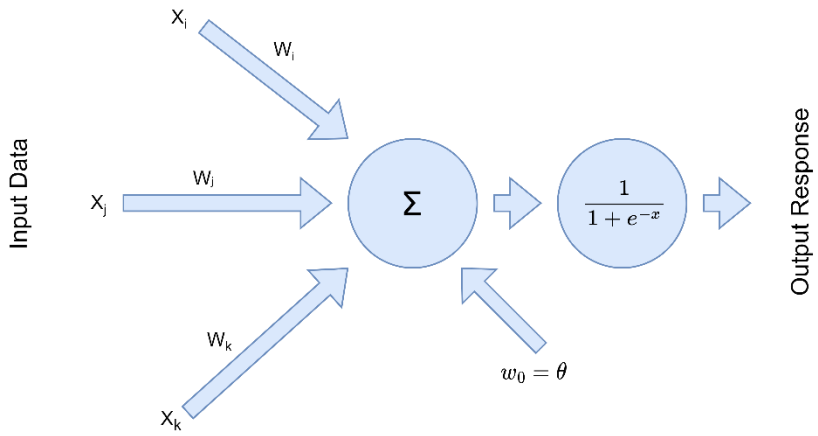


Figure 2.3: Illustration of perceptron in modern machine learning (adopted from Wang & Raj (2017))

Perceptron nowadays also refers to a single-layer neural network, which can be illustrated by linear regression and is widely used for solving regression problems. Based on the assumption of linear regression, a linear relationship is assumed to be happening between the dependent variable y and the dependent variable x. Thus, y can be represented by the weighted sum of feature x with some noise, which is assumed to be normally distributed. The linear regression can be expressed in mathematical form: $\hat{y} = w_1 x_1 + \cdots + w_d x_d + b$. Where w_d is weight and b is bias. Weights define the importance of each feature and affect the output value. The bias will be the output value when all the features have a value of 0. The concept behind linear regression and perceptron is almost the same. The only difference is linear regression does not consider the threshold function.

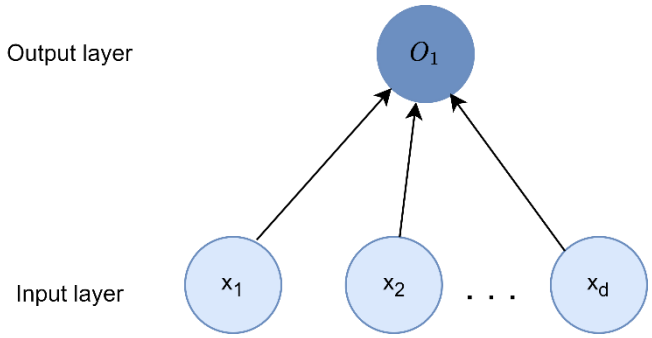


Figure 2.4: Single-layer linear regression neural network (adopted from Zhang et al. (2021))

Ignoring the weights and biases, Figure 2.4 illustrates the structure of the linear regression. $x_1, \ldots, x_d$ represent the number of features. Without considering the input layer, there is only one layer in the linear regression model. This led to linear regression also being referred to as a single-layer fully connected neural network (Zhang et al., 2021).

Due to the characteristic of a linear function, the perceptron can only solve the linear problem in the real world as the decision boundary of the perceptron is linear, which can only represent logical operations such as AND or OR. Minski and Papert (1969) pointed out that the non-linear function, such as XOR function, was unable to be represented by perceptron. At the time, this became an obstacle to the development of neural networks (Shrestha and Mahmood, 2019).

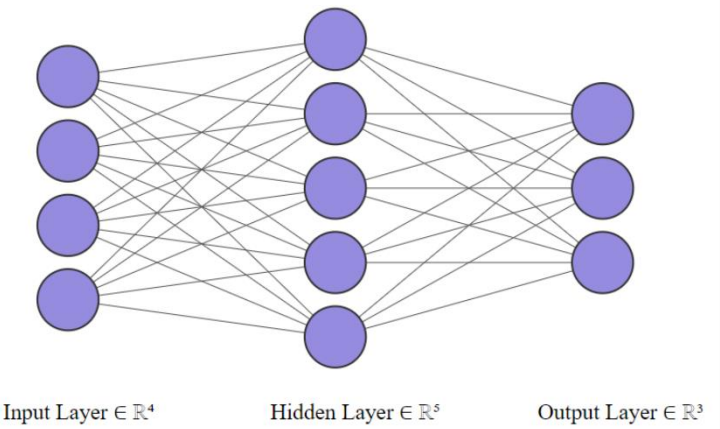


Figure 2.5: An MLP with one hidden layer of 5 hidden units (adopted from Zhang et al. (2021))

The concept of universal approximation property has been introduced. It stated the multilayer perceptron (MLP) can be formed by stacking a one-layer neural network as a hidden layer into the original single one-layer neural network, as shown in Figure 2.5. MLP can deal with the non-linear function now as Boolean function and continuous function can be represented by MLP. However, there was no proper way to guide the training process of MLP. The neuron weight must be updated during the training process to generate the desired output. A mechanism for quantifying the difference between the generated and expected outputs is needed. When the number of layers increases, it is getting harder to quantify the contribution of the output of each neuron to the error (Shrestha and Mahmood, 2019).

The challenge faced by MLP had been solved with the introduction of backpropagation, "Backpropagation first propagates the error term at output layer back to the layer at which parameters need to be updated, then uses standard gradient descent to update parameters with respect to the propagated error." (Wang and Raj, 2017). With the help of backpropagation, MLP can now adjust the neuron's weight to reduce the error. In order to fully use the multilayer perceptron, a non-linear activation function needs to be implemented in each hidden neuron to introduce the non-linearity in MLP (Zhang et al., 2021).

However, there are two limitations of MLP when dealing with computer vision tasks. The huge number of weights or dimensions made the MLP extremely computationally expensive to train. Let's assume an image with a size of $256 * 256$, resulting in $256 * 256 * 3 = 196{,}608$ input dimension for an image as the image normally contains three channels, which are red, green and blue (RGB). Since the number of hidden layers is often bigger than the input layer, the number of weights will surpass $196{,}608^2$ dimension even for a shallow network. Secondly, according to research, nearby image pixels are statistically related. However, MLP cannot capture the relationship between nearby image pixels and thus cannot capture the spatial information as it does not consider the image's local structure (Prince, 2023).

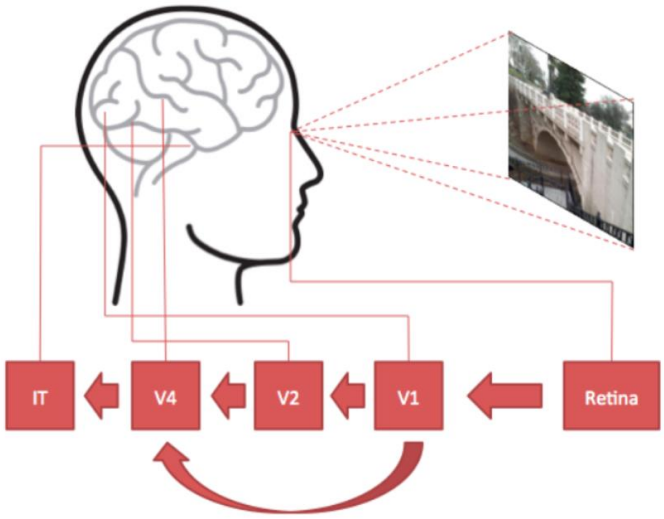


Figure 2.6: Illustration of the visual cortex in the human vision system (adopted from Wang & Raj (2017))

In order to solve the limitation of MP in computer vision tasks, Convolutional Neural Network (CNN) has been introduced. The idea of CNN comes from the visual cortex in the human brain. Figure 2.6 illustrates the information processing process when the brain receives image signals. The primary visual cortex (V1) receives the image pixel from the retina as a signal and extracts low-level detail such as edge. The secondary visual cortex (V2) will then receive the signal from V1 and extract the mid-level features such as orientation, spatial frequency and colour. The signal will then be sent to V4, and high-level features such as geometric shapes will be extracted. Lastly, the Inferior temporal gyrus (IT) identifies the object based on the feature extracted before (Wang & Raj, 2017).

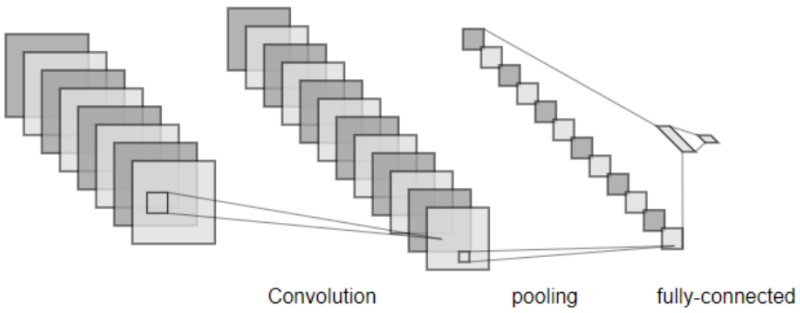


Figure 2.7: A simple CNN architecture (adopted from O'Shea & Nash (2015))

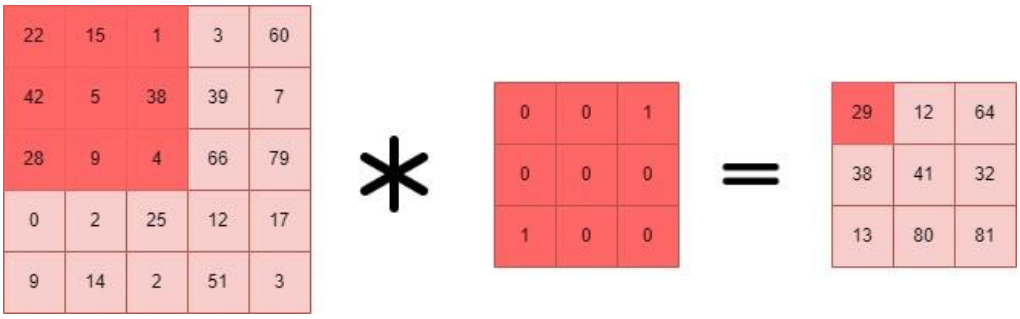


Figure 2.8: A simple illustration of convolution operation (adopted from Wang & Raj (2017))

As shown in Figure 2.7, CNN typically consists of three components, which are the convolutional layer, pooling layer and fully connected layer. The convolutional layer will compute the dot product between the weight of local regions in the input feature map and kernel, to extract the feature in the input feature map and produce the output feature map. Non-linearity will be introduced by applying the activation function, such as rectified linear unit, and it only allows the active feature to pass through to the following layer (O'Shea and Nash, 2015).

For example, as illustrated in Figure 2.8, the input feature map is the leftmost matrix, the kernel is located in the middle, and the output feature map is the rightmost matrix. The target matrix of the input feature map is the top-left 3x3 submatrix, as the size of the kernel is $3 * 3$. The dot product will be performed between the target matrix and the kernel and produce a result of 29. When the stride is equal to one, the target matrix will slide one column to the right to continue to perform the dot product. As a result, every $3 * 3$ target matrix in the input feature map will be convoluted into one digit (Wang and Raj, 2017).

The objective of the pooling layer is to reduce the complexity of the model via pooling operation. A fixed shape window will slide through the input regions based on the stride, and each pooling window will output a digit to summarise the features present in a region to decrease the dimensionality and number of parameters. There are two types of pooling operation, which are maximum pooling that computes the maximum value in the pooling window and average pooling that computes the average value in the pooling window (Zhang et al., 2021).

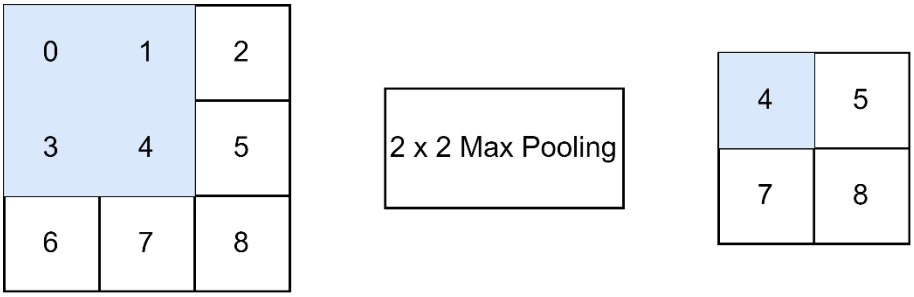


Figure 2.9: Max-pooling operation with window shape of $2 * 2$ (adopted from Zhang et al. (2021))

For example, as illustrated in Figure 2.9, the top-left 2x2 submatrix in the input feature map will be treated as a pooling window, a maximum value between 0,1,3,4 will be computed, and the output will be 4. The pooling window will continue to slide through the input feature map from top to bottom and left to right (Zhang et al., 2021).

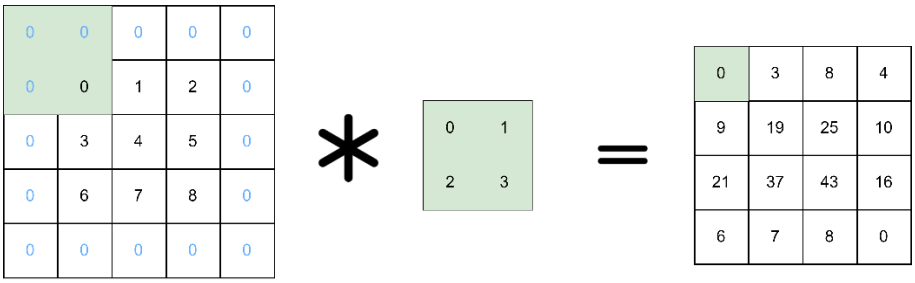


Figure 2.10: Convolution operation with padding (adopted from Zhang et al. (2021))

The fully connected layer in CNN, which is an MLP network, will treat the feature map extracted by the convolutional layer and downsampling by the pooling layer as input and perform the classification task. When the number of convolutional layers increases, the receptive field of the output feature map will become larger, thus increasing severe loss of edge pixels. This had become one of the problems of CNN, which caused some of the useful information in the image to be lost. To solve the problem, padding has been implemented. Padding will add extra zero filler pixels around the edge of the image to increase the input image's effective size, enlarge the output size, and achieve the goal of preserving the boundary information. For example, as illustrated in Figure 2.10,

$3 * 3$ input is padded to $5 * 5$, producing a $4 * 4$ output matrix (Zhang et al., 2021).

In short, CNN is able to extract the spatial information vis convolutional layer, and computing resources are also decreased via pooling operation, which solves the problem of spatial information and computing resources faced by MLP.

2.3 Overview of the method used in defect detection

The following section will provide a comprehensive review of various defect detection methods, including those based on traditional image processing techniques, as well as deep learning methods such as classification and object detection.

2.3.1 Defect detection using image processing technique

The following section will review defect detection methods that utilise image processing techniques.

2.3.1.1 Template-based systems for wafer die surface inspection.

In 2005, a template-based vision system for the inspection of wafer die surfaces had been developed by Shankar and Zhong. The system is able to detect the defect as small as two-thousandths of an inch in a wafer that up to 8 inches.

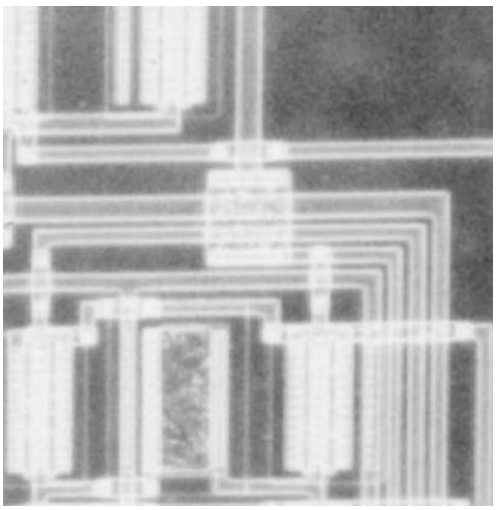


Figure 2.11: Mask image (adopted from Shankar & Zhong (2005))

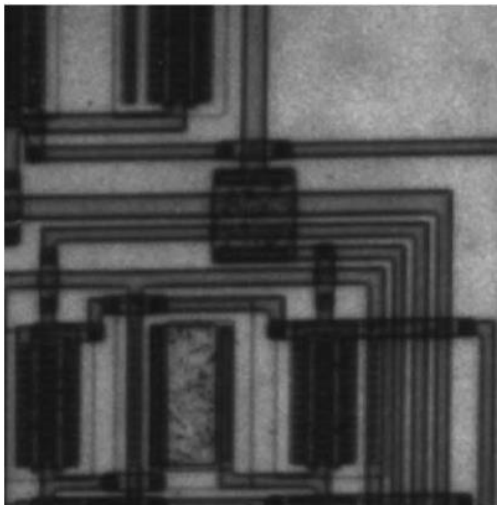


Figure 2.12: Reference template (adopted from Shankar & Zhong (2005))

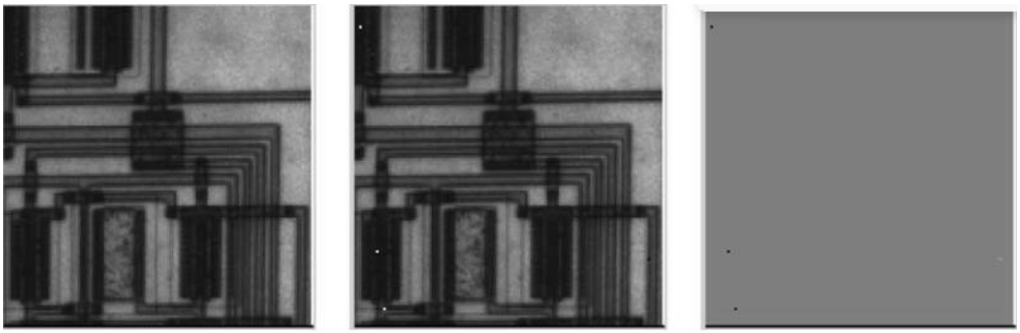


Figure 2.13: (a) Reference image (b) Test image (c) Difference image (adopted from Shankar & Zhong (2005))

The following is a description of the system's step-by-step defect identification procedure. Edge detection is performed on the reference image to produce the mask image, as shown in Figure 2.11. Figure 2.12 illustrates the reference template. For each die in the wafer, a subtraction operation is performed between the test die image and reference image to produce an absolute difference image, as shown in Figure 2.13. The pixel differences between the two images are depicted in this absolute difference image. The mask image is then multiplied with the difference image to reduce the potential pixel inconsistencies arising from various factors. A rule-based defect specification system will applied to the pixel difference discovered in the difference image to determine whether the observed defect is tolerable or not based on the rule set by the manufacturer.

As the system is based on a reference approach, a high-quality reference image is a prerequisite to generate a perfect difference image that avoids false detection and a high-quality reference image is needed to act as a "golden" sample. However, high-quality reference is not always possible, which has become one of the disadvantages of the system. Furthermore, unknown defects or new defects might not be able to be detected using the reference approach. Some of the defects that are hard to express in a rule-based system might also produce false detection. Rule-based defect specification system is the advantage of the system, as it can further classify the detected defect into tolerable and critical defects.

2.3.1.2 Two-dimension wavelet transform (2-D WT) approach for semiconductor wafer die surface inspection

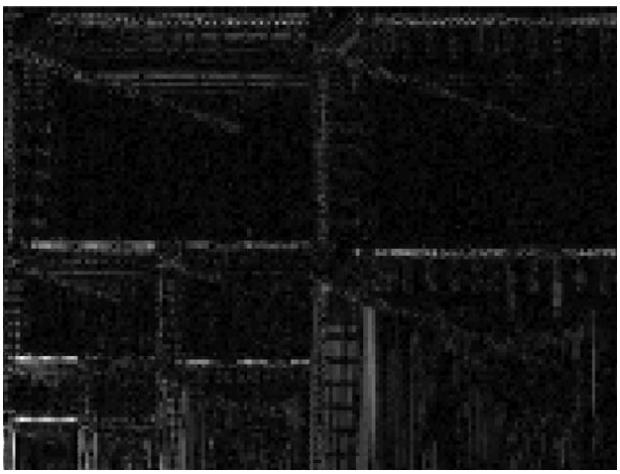


Figure 2.14: Wavelet energy in the image (adopted from Yeh et al. (2010))

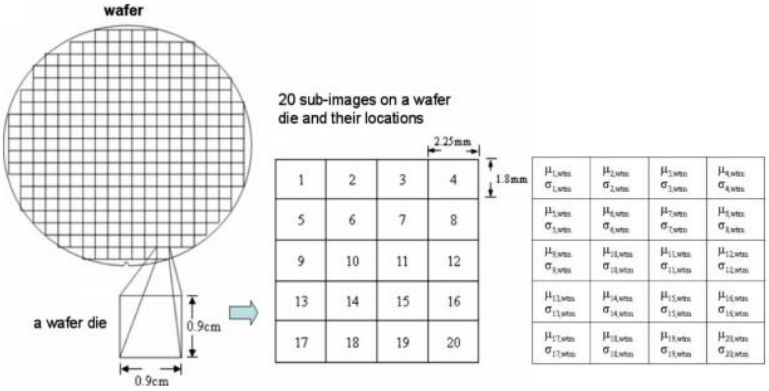


Figure 2.15: 20 sub-images of a wafer die (adopted from Yeh et al. (2010))

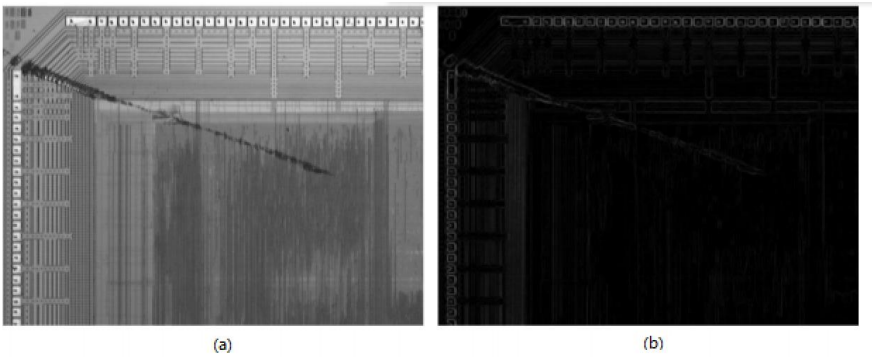


Figure 2.16: (a) Original scratch image on die. (b) Images after applying median and Sobel filters (adopted from Yeh et al. (2010))

Yeh et al (2010) implemented a two-dimension wavelet transform (2-D WT) approach for detecting the visual defects on semiconductor wafer die. The wavelet transform is a mathematical technique for breaking down signals and data into their component frequencies. A total of 4 types of 2-D wavelets will be produced. Each set of 2-D wavelets can capture different aspects of the input signal, including the smooth part, vertical detail part, horizontal detail part, and diagonal detail part. This means that the 2-D WT is able to decompose an input 2-D signal, such as an image, into 2-D wavelet coefficient matrices that represent smooth and detailed parts of the signal. “Wavelet energy” is used to represent the squares of a pixel’s coefficient. An image’s objects retain more wavelet energy than its backgrounds. Compared to pixels in smooth sections of the object, pixels at corners, noisy clusters, or jagged edges retain substantially more wavelet energy. WTMS describes the “clustering wavelet energy”, which considers the local clustering of wavelet coefficients at a particular location in the image by considering the wavelet energy of the neighbourhood around an image pixel. Figure 2.10 illustrates the wavelet energy in an image, where the whiteness of pixels indicates greater wavelet energy.

The following is a description of the system’s step-by-step defect detection procedure, In step 1, to achieve the goal of high-resolution inspection, an image is taken for each wafer die and then split into 20 smaller sub-images as shown in Figure 2.11. In step 2, the median filter and Sobel filter is applied to preprocess the sub-image, as shown in Figure 2.12. To reduce the computational complexity, pixels within each sub-image will be preselected,

and WTMS will be calculated for these preselected pixels only. Step 3, wavelet energy in each pixel can be expressed by WTM for each pixel in each sub-image with a different scale, which is calculated by taking the absolute value of the wavelet coefficients. A pixel will advance to the subsequent stage and be selected for the computation of WTMS if the WTM of a pixel surpasses the population mean and standard deviation derived from the golden image. In step 4, the calculation of WTMS will be executed for the chosen pixel at varying scales.In step 5, the interscale ratio of the selected pixel will be calculated. A pixel's clustered wavelet energy at one scale is significantly greater than its energy at another scale if the interscale ratio is less than zero. This indicates that the pixel is more likely to be a defect at this scale than at other scales. Such defects might manifest as irregular edges, sharp corners, or clustered noisy areas in an image of a wafer die. The pixel with an interscale ratio of more than 0 is classified as a non-defect pixel. The pixel with interscale ratio less than 0 will enter the next step. The non-defective pixels such as the corner and edges of water in golden image might also have a value of interscale that is less than 0. To avoid false detection, the selected pixel with an interscale ratio of less than zero is compared to those pixels with an interscale ratio of less than zero in the golden image. The selected pixel is considered as non-defective if the detected defect pixel belongs to the non-defective pixel in the golden image, else the pixel is consider as defective pixel. Steps 2 to 6 will be repeated for the remaining sub-images.

The advantage of the approach is it is suitable for a diverse array of product categories as the comparison is made indirectly between the WTM in the test image and the population mean and standard deviation in the golden image. Instead of performing precise pixel-by-pixel matching between the golden and test images, the approach is considered a non-reference method based on the statistical comparison methodology. Most importantly, this method eliminates the need for training and can directly execute inference.

The disadvantage of the approach is that the golden template is still needed for comparison, and proper parameters such as wavelet basis or number of decomposition levels must be chosen wisely to avoid false detection. It may be necessary to experiment with different parameters to determine the optimal

parameters that vary on different datasets. Moreover, the approach cannot categorise defect pixels into particular types of defects.

2.3.2 Defect detection with deep learning: classification approach

The following section will review defect detection methods that utilise the classification approach.

2.3.2.1 Automatic defect classification for wafer surface damage using CNN and k-NN

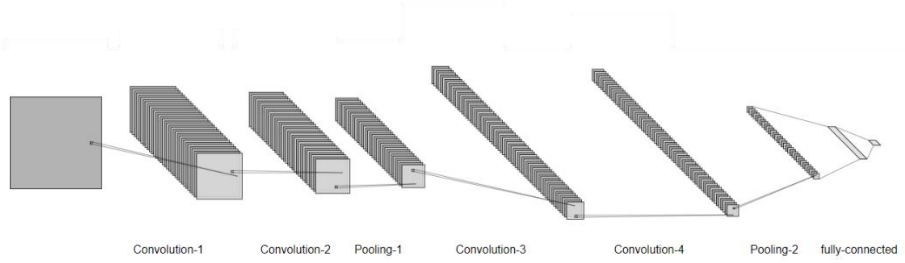


Figure 2.17: CNN architecture (adopted from Cheon et al. (2019))

Table 2.1: CNN configuration (adopted from Cheon et al. (2019))

CNN parameter	Value
Number of convolution layers	4
Filter size	3 × 3
Number of pooling layers	2 max-pooling
Pooling filter size	2 × 2
Number of feature maps	32/32/64/64
Number of fully connected layers	1
Number of nodes in fully connected layer	512
Activation function	ReLU
Regularization method	Dropout
Classification function of the output layer	Softmax
Loss function	Categorical cross entropy

A deep learning-based automatic defect classification method was proposed by Cheon et al (2019). The main objective is to classify different wafer surface damage. The method is developed based on the CNN and k-nearest neighbours algorithm (k-NN). The valid feature is extracted by CNN, while those defect types that were not encountered during the training phase will be recognised by

k-NN. Figure 2.17 shows the CNN architecture, and the configuration of CNN is shown in Table 2.1. ReLU activation function is applied to all the layers except for the output layer. Dropout is utilised in CNN as a regularisation technique to mitigate overfitting.

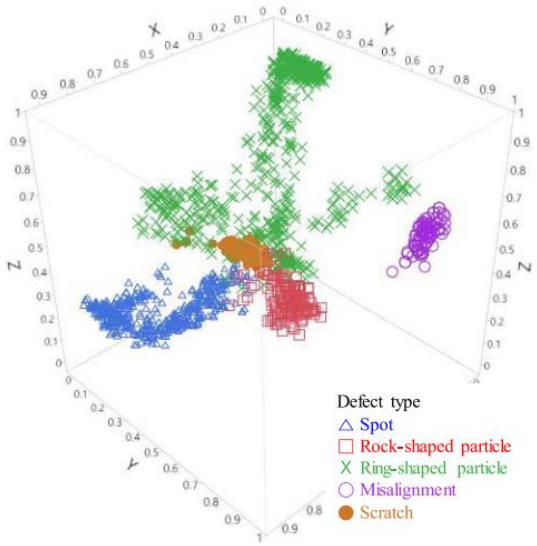


Figure 2.18: Illustration of defect image cluster analysis results by identified class (adopted from Cheon et al. (2019))

The algorithm for detecting surface defects on semiconductor wafers can be conceptually divided into two main phases. Clustering and threshold configuration are carried out in the initial phase, while membership testing is carried out in the subsequent phase. If the image exceeds the threshold, which means the image is too far from other clusters, then no label will be assigned to the image. This causes the k-NN to be slightly different from the regular k-NN.

The algorithm's initial phase filters away the CNN training image that was incorrectly labelled. The structure of CNN is then modified by applying the sigmoid function as an activation function to the fully connected layer. This process aims to normalise the feature vector within the range of zero to one. The training image that was properly labelled is fed into the modified CNN, and the feature vector will be output by the output layer and act as input for k-NN. The k-NN classifier is then constructed for each cluster. The k number is set to one, and Euclidean distance is used in calculating the distance. For each image in a particular cluster, the total squared distance is computed between the image and

its single neighbouring image within the same class. The confidence limit which can express as 100(1 − α) % will be involved in the threshold calculation. In order to solve the problem of overlapping distance distributions due to the presence of similar defect classes, α or the classification error rate is permitted. The final threshold will be 90% percentile of the empirical distribution of the total squared distance within each cluster as α is set to 0.1. The output of the clustering is visualised in Figure 2.18.

In the second part of the algorithm, the inference image is passed to the modified CNN and generates the feature vector. The feature vector generated is then passed to k-NN for calculating the total squared distance between the inference image and each cluster. The inference image will be classified as an "unknown" defect if it exceeds the threshold for all clusters. Otherwise, the inference image will be sent to the unmodified CNN to perform classification.

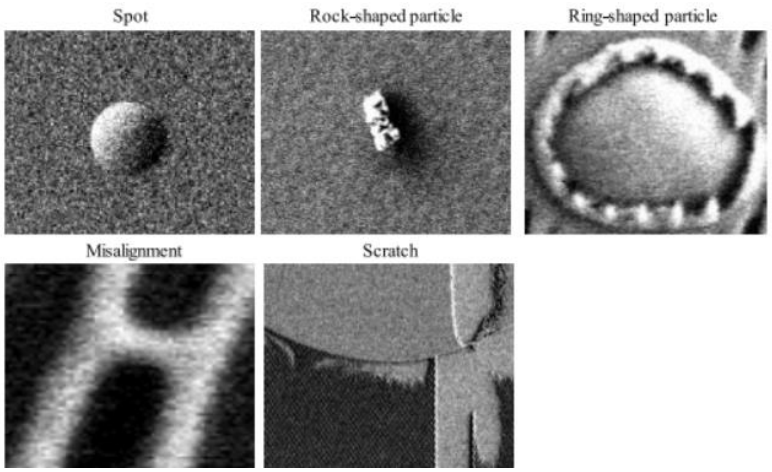


Figure 2.19: Five defects in Dataset-TT (adopted from Cheon et al. (2019))

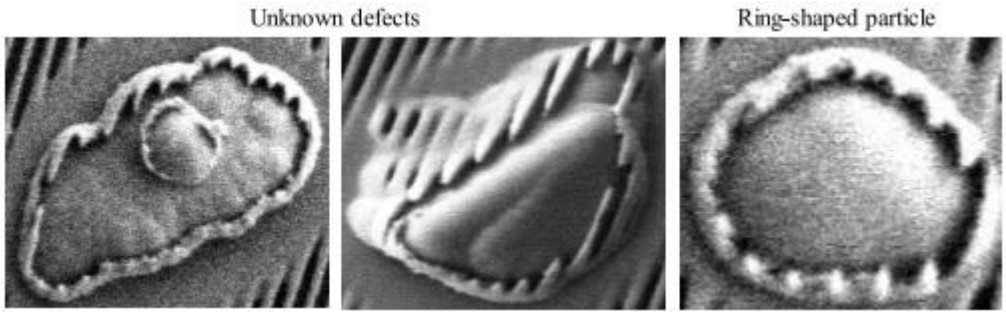


Figure 2.20: Misclassified "unknown" defect vs ring-shaped particle image (adopted from Cheon et al. (2019))

Table 2.2: Accuracy and time comparison of CNN and other models on the testing dataset (adopted from Cheon et al. (2019))

Classifier	Train accuracy	Valid accuracy	Test accuracy	CPU times in training	CPU times in testing
CNN	99.4%	98.7%	96.2%	42856	1.813
SAE	99.1%	94.0%	91.8%	49471	0.781
MP	99.9%	93.4%	92.8%	22757	1.313
SVM-rbf	100%	94.3%	92.5%	26485	125.516
MP (extracted feature)	53.1%	56.3%	55.2%	3970	0.016
SVM-rbf (extracted feature)	66.8%	62.9%	62.4%	145	0.203

Table 2.3: k-NN test result for DATASET-NN (adopted from Cheon et al. (2019))

Defect class	Threshold value	Number of images that exceeds threshold
Spot	6.133	30
Rock-shaped particle	34.083	30
Ring-shaped particle	23.812	28
Misalignment	1.167	30
Scratch	43.700	30

Two datasets, dataset-TT and dataset-UN were used, dataset-TT was used to train and test the unmodified CNN. This dataset contains 2,133 images with five defective categories, as shown in Figure 2.19. Dataset-UN was used to evaluate the ability of k-NN to detect instances of unknown defect categories. This dataset contains 30 “unknown” defect images. As shown in Table 2.2, CNN achieved the highest accuracy of 96.2% in the testing dataset while comparing with other networks such as MP, SAE, SVM-rbf, MP and SVM-rbf with edge detection algorithm. Table 2.3 shows the threshold for each cluster and the result of using k-NN to detect unknown defects in Dataset-UN. Out of the thirty unknown defects, k-NN failed to identify two images with “unknown defects”.

k-NN identified these two unknown defects as ring particle defects as these defects look very similar, as shown in Figure 2.20.

The advantage of the method is it can identify the unknown defect. Over time, equipment ageing and environmental changes can occur, and the production process can change slightly. This leads to the emergence of new and unknown defects. The existing trained CNN may make incorrect assessments of these defects. The CNN needs to be retrained to detect these defects, but this will face the problem that there might not be a sufficient number of images available to train the CNN during the initial stages of the appearance of new defects. The long training time and inference time had become disadvantages of this method. As shown in Table 2.2, the training time required to train the CNN is approximately 11.9 hours, and the inference time needed is 1.813 seconds for an image, making it unsuitable for real-time inference.

2.3.2.2 Semiconductor defect pattern classification using SP&A Net

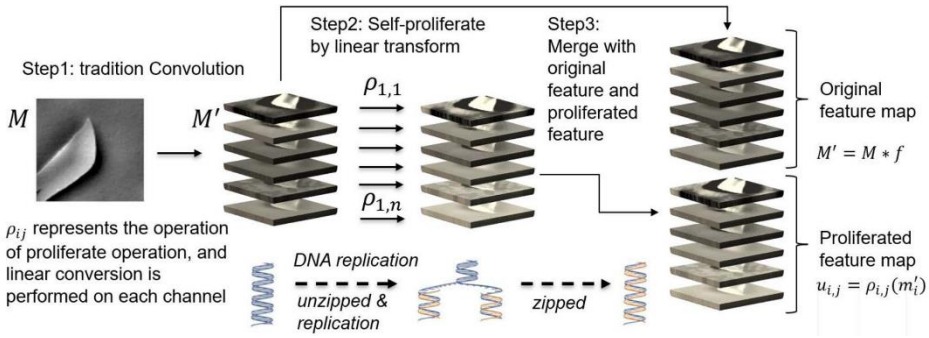


Figure 2.21: Self-proliferation process (adopted from Yuanfu Yang & Sun (2022))

In 2022, YuanFu Yang and Sun proposed an architecture for semiconductor defect pattern classification named SP&A-Net. A series of linear transformations are applied in the process of self-proliferation to generate extra feature maps. Channel-wise and spatial-wise attention mechanisms are applied in the self-attention process to capture the feature map's long-range relationship. The primary objective of SP&A-Net is to reduce computational complexity while upholding high accuracy in defect inspection tasks. There are two main

components in SP&A-Net, which are the self-proliferation block and the self-attention block.

Self-proliferation block Is inspired by the DNA replication process, which generates an extra feature map akin to the replication process in DNA to increase the accuracy. This block performs a standard convolutional operation and outputs a series of feature maps in the initial step. The subsequent step involves applying a linear transformation through a depth-wise operation to each channel of the feature map acquired from the previous convolutional stage, resulting in a set of extra maps that are equivalent in quantity to the channels present in the original feature map. The original feature map is then merged with the extra feature map generated. The process of self-proliferation is illustrated in Figure 2.21.

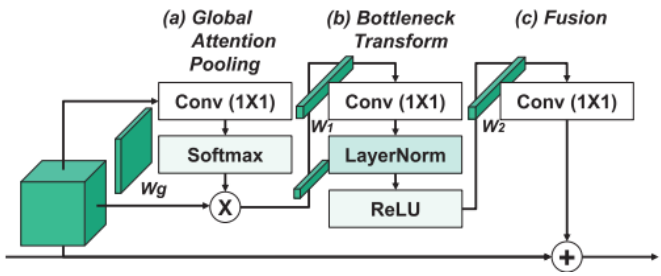


Figure 2.22: Self-attention block (adopted from Yuanfu Yang & Sun (2022))

The self-attention block aims to incorporate the information from other positions in the sequence to enhance the attributes of specific query positions. The self-attention block can be divided into sub-components, which are global attention pooling, bottleneck transform, and fusion, as shown in Figure 2.22. By considering the pairwise relationship between each location and the query location, an attention map that illustrates the significance of each spatial location within the input feature map is created by global attention pooling. Elementwise multiplication is applied between the attention map and the input feature map to generate the output feature map. The bottleneck transform captures the channel-wise dependencies by emphasising the most significant channels in the feature map and suppressing the unnecessary ones. In the fusion function, the global context feature is aggregated into the input feature map by applying

broadcasting element-wise addition. The structure of the self-attention block is illustrated in Figure 2.22.

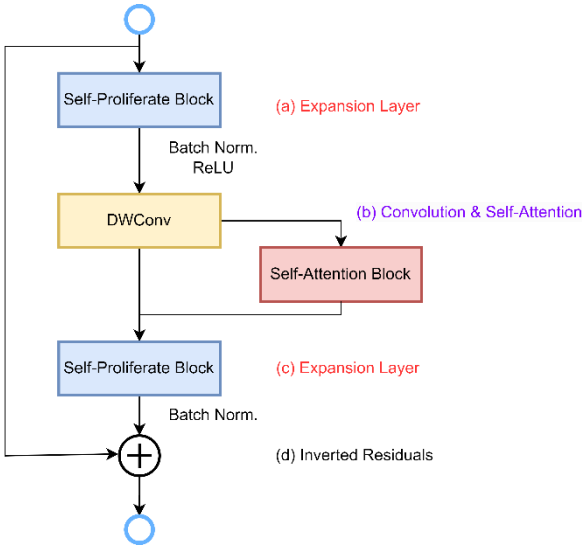


Figure 2.23: SP&A Block (adopted from Yuanfu Yang & Sun (2022))

Table 2.4: SP&A-Net architecture (adopted from Yuanfu Yang & Sun (2022))

Input	Block	Output	Expansion	Self-Attention
512X512X3	Conv2D 3X3	16	-	-
256X256X16	SP&A Block	16	16	1
256X256X16	SP&A Block	24	48	1
128X128X24	SP&A Block	24	72	1
128X128X24	SP&A Block	48	72	1
64X64X48	SP&A Block	48	120	1
64X64X48	SP&A Block	96	240	1
32X32X96	SP&A Block	96	200	1
32X32X96	SP&A Block	96	184	1
32X32X96	SP&A Block	96	184	1
32X32X96	SP&A Block	144	480	1
32X32X144	SP&A Block	144	672	1
32X32X144	SP&A Block	192	672	1
16X16X192	SP&A Block	192	960	1
16X16X192	SP&A Block	192	960	1
16X16X192	SP&A Block	192	960	1
16X16X192	SP&A Block	192	960	1
16X16X192	Conv2D 1X1	960	-	-
16X16X960	AvgPool 16X16	-	-	-
16X16X960	Conv2D 1X1	1280	-	-
1X1X1280	FC	1000	-	-

Table 2.5: Dataset Description with Feature Description, Defect Types, and Example Images (adopted from Yuanfu Yang & Sun (2022))

Defect Type	Image1	Image2	Image3	Feature Description	Defect Type	Image1	Image2	Image3	Feature Description
Remain				Flat irregular shape, usually on the edge of the wafer	Hump				Irregular bumps, edges fused to surface
Silk				Discontinuous linear protrusion	Flask				Flaky, undulated top surface
Multi-dots				Random dots raised (more than three points)	Fallon				Large irregular particles
Scratch				Linear depression	Oval				Large particles made of spheres
Small-Particle				Small irregular particles	Color Mark				No obvious undulations on the surface
Ball				Round particles					

Table 2.6: Comparison of SP&A-Net and Resnet-50 with ablation study in different blocks (adopted from Yuanfu Yang & Sun (2022))

Model	#Params (K)	*Refinement*									
		Regularization			Flatten		Optimizer			Learning Rate	
		Batch Normalization	Layer Normalization	Group Normalization	Global Avg. Pooling	Conv1X1	SGD	Adam	NAG	Reduce LR On Plateau	Cosine Annealing
ResNet50+SE Block	2673	97.23%	97.23%	97.10%	97.07%	97.25%	97.03%	97.23%	97.12%	97.31%	97.39%
ResNet50+NL Block	4125	97.44%	97.41%	97.51%	97.28%	97.58%	97.44%	97.54%	97.59%	97.35%	97.38%
ResNet50+SNL Block	3742	97.46%	97.57%	97.61%	97.62%	97.34%	97.21%	97.35%	97.38%	97.52%	97.25%
ResNet50+GC Block	3353	97.45%	97.19%	97.34%	97.08%	97.44%	97.42%	97.31%	97.51%	97.24%	97.66%
SPNet (without attention)	2120	97.58%	97.64%	97.59%	97.49%	97.67%	97.37%	97.39%	97.49%	97.54%	97.57%
SP&A-Net (with self-attention)	2135	97.94%	98.15%	97.93%	97.91%	97.94%	97.66%	97.69%	97.87%	97.97%	**98.27%**

Table 2.7: Analysis of the effect of composition ratio on the performance of SP&A-NET (adopted from Yuanfu Yang & Sun (2022))

r	Accuracy	precision	Recall	F1- Score	#Params (M)
0.06	98.45%	97.45%	99.16%	98.30%	5.10
0.13	98.44%	97.43%	99.15%	98.28%	3.40
0.25	98.40%	97.37%	99.05%	98.20%	2.98
0.50	**98.38%**	**97.32%**	**99.03%**	**98.17%**	**2.60**
0.63	97.81%	96.75%	98.44%	97.58%	2.47
0.83	97.21%	96.38%	97.48%	96.93%	2.35
1.00	95.83%	94.24%	96.62%	95.42%	2.29

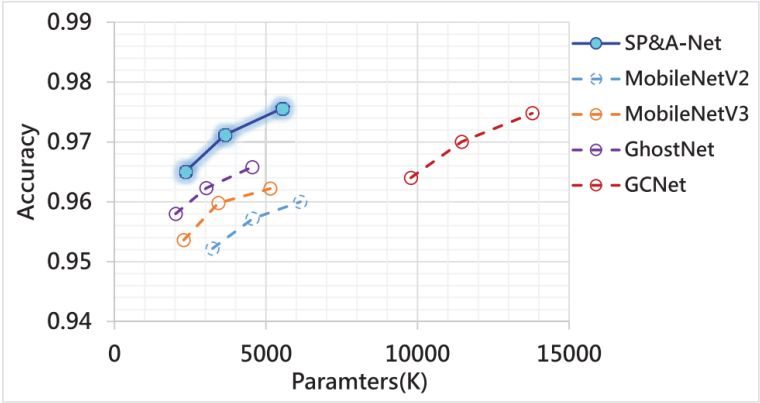


Figure 2.24: Performance evaluation of AEI defect pattern (adopted from Yuanfu Yang & Sun (2022))

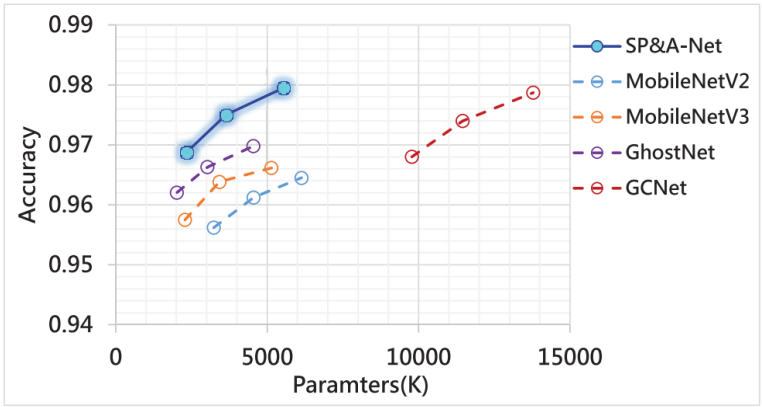


Figure 2.25: Performance evaluation of ADI defect pattern (adopted from Yuanfu Yang & Sun (2022))

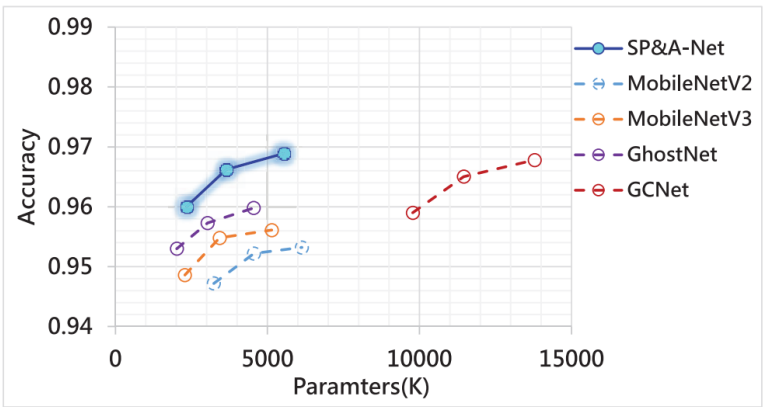


Figure 2.26: Performance evaluation of API defect pattern (adopted from Yuanfu Yang & Sun (2022))

The self-proliferation-and-attention block is built by combining the self-attention block and self-proliferation block. Self-proliferation-and-

attention block is built by applying the concept of inverted residual block, in which a shortcut is implemented to mitigate the issue of gradient vanishing. In this block, the dimension of the input channel is expanded, and the feature is extracted, then the output will be projected back to the original smaller input channel dimension. This block can be divided into four parts, as illustrated in Figure 2.23, which are the expansion layer, convolution and self-attention layer, compression layer, and inverted residuals. Additional feature maps are generated in the expansion layer to enlarge the input dimension. In the convolutional and self-attention layer, depth-wise convolution is applied to extract the feature from the expanded feature map, and the long-range dependencies are captured by the self-attention block. The output dimension is then reduced by the compression layer to the same dimension as the input dimension. The network architecture of SP&A-Net is shown in Table 2.4.

Three datasets, the AEI dataset, ADI dataset and the API dataset, were used to evaluate the performance of SP&A-Net. Table 2.5 shows the 11 defect categories of the dataset. An optimal strategy for SP&A-Net was found through an ablation study, as shown in Table 2.6. In this ablation study, the SP&A-Net was also compared to Resnet-50 with different blocks, such as SE blocks, NL blocks, SNL blocks, and GC blocks. In addition, by taking into account the trade-off between the f1-score and the number of parameters, an optimal composition ratio (r) of 0.5 is selected for the self-proliferation within the network through the ablation study, as shown in Table 2.7. As shown in Figures 2.24, 2.25, and 2.26, SP&A-Net outperforms the compared networks in terms of accuracy with lower parameters across three different datasets.

The key advantage of the network is that the large kernel is no longer necessary to capture the large receptive fields of the features since the network can capture the spatial relationships in the feature maps, which leads to a decrease in the computational cost of the network. Moreover, due to the extra feature maps generated by the self-proliferation block, the accuracy of the network has been further enhanced. The SP&A-Net attains impressive accuracy levels while demanding fewer parameters and FLOPs than other baseline networks.

2.3.2.3 Multi-scale inspection framework for surface defect detection using MST-GAN

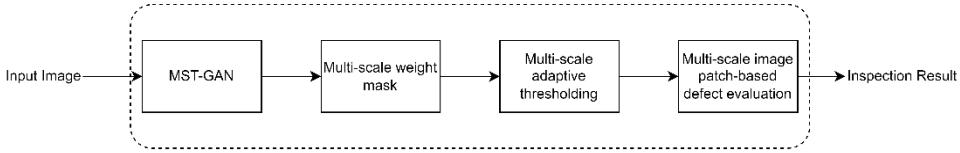


Figure 2.27: Inspection framework's pipeline (adopted from Chen et al., 2023)

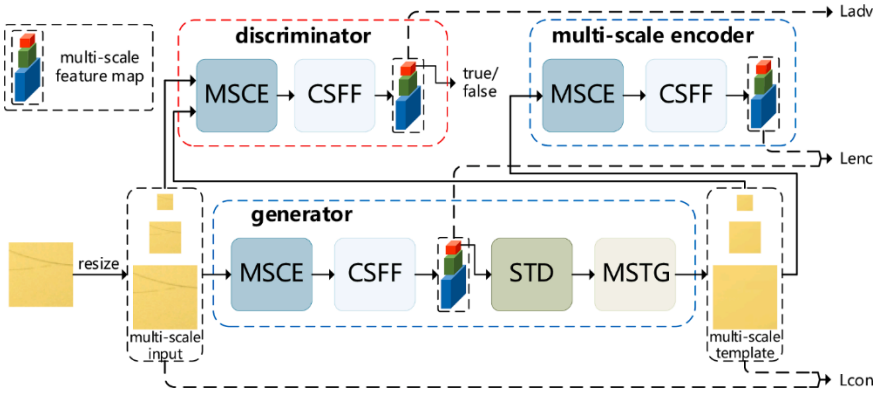


Figure 2.28: Architecture of the MST-GAN (adopted from Chen et al. (2023))

A multi-scale inspection framework for surface defect inspection of IC metal packages was proposed by Chen et al. (2023). The framework can be divided into main parts, as shown in Figure 2.27.

MST-GAN consists of three components, which are a generator, discriminator, and multi-scale encoder. MST-GAN learns the quality pattern present in the image through training with 768 defect-free samples. MST-GAN is able to simulate the input photo under defect-free conditions by “describing” the intrinsic quality pattern of the input image and output a multi-scale defect-free templates. The concept of generator and discriminator corresponds to the min-max two player game, in which the generator will learn the data distribution of real images through the training process and try to generate images that closely resemble real ones while the discriminator will try to differentiate those generated images and realistic image to assisting the generator in refining its output. The multi-scale template is generated by the generator in MST-GAN and the discriminator in MST-GAN differentiates between the real and generated templates by leveraging fused high-level feature maps. To improve

the performance of the generator to generate more realistic images, the multi-scale encoder extracts pyramid feature maps from the generated templates and applies the information from these feature maps in the computation of the loss function. MSCE and CSFF form the discriminator and multi-scale encoder. On the other hand, the generator consists of MSCE, CSFF, STD and MSTG.

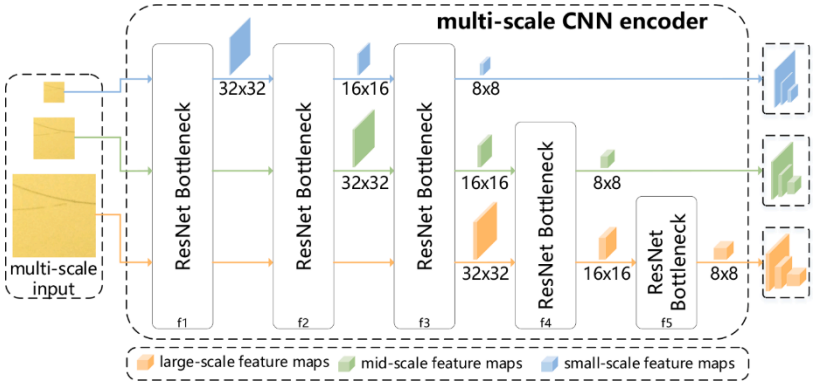


Figure 2.29: Structure of MSCE (adopted from Chen et al. (2023))

Table 2.8: Dimensions of the multi-level feature maps extracted at each scale (adopted from Chen et al. (2023))

	high-level	mid-level	low-level
large-scale feature maps	8 × 8 × 256	16 × 16 × 128	32 × 32 × 64
mid-scale feature maps	8 × 8 × 128	16 × 16 × 64	32 × 32 × 32
small-scale feature maps	8 × 8 × 64	16 × 16 × 32	32 × 32 × 16

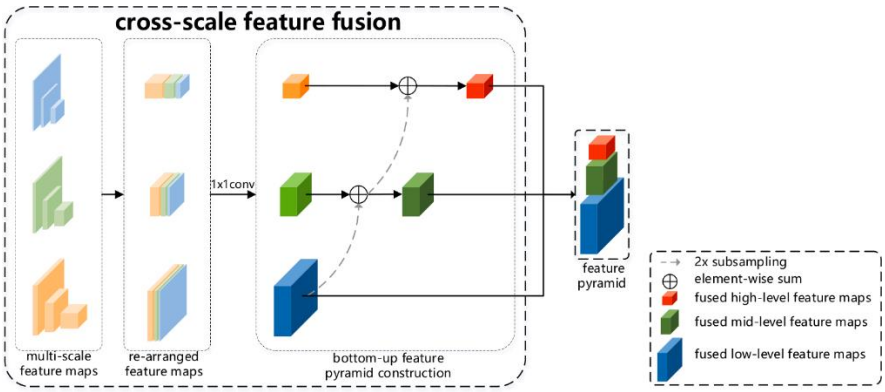


Figure 2.30: Structure of CSFF (adopted from Chen et al. (2023))

Figure 2.29 illustrates the structure of MSCE, which extracts multi-scale features from multi-scale input images through multiple ResNet blocks in

MSCE. A series of feature maps of different sizes are obtained, which include feature maps of small, medium, and large scales that capture low-level, mid-level and high-level features of each input scale. Table 2.8 shows the dimension of the extracted feature map for each input scale. Figure 2.30 illustrates the structure of CSFF. Multi-scale feature map generated by MSCE is reconstructed by CSFF via the bottom-up approach, and a feature pyramid will be produced. Feature maps with different scales at the same level will be rearranged via concatenation by MSCE. This rearranged feature map subsequently undergoes a 1x1 convolutional layer. Subsampling and element-wise summation are applied to the lower-level feature map. The lower-level feature map is now added to the higher-level feature map. This suggests that some of the valuable features present in high-level representations may also be represented in low-level representations. Integration of MSCE and CSFF allows intrinsic features in the input image to be efficiently captured in the pyramid feature map.

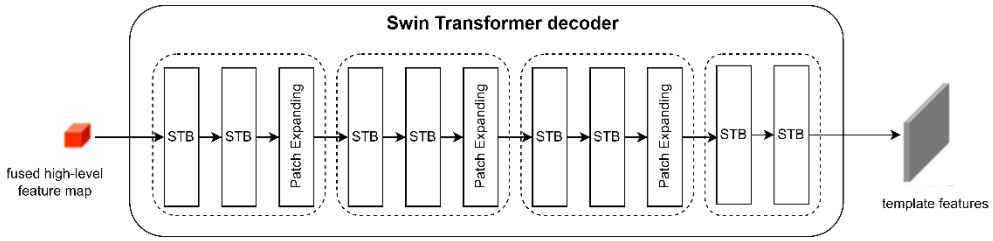


Figure 2.31: Structure of swin transformer decoder (adopted from Chen et al. (2023))

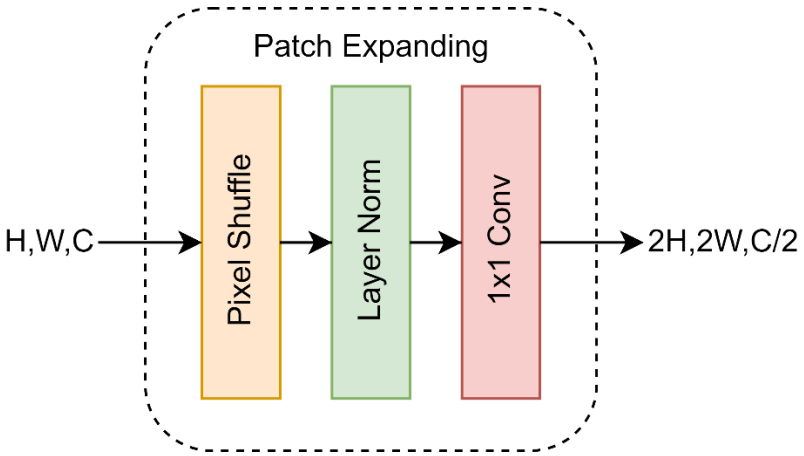


Figure 2.32: Structure of patch expanding (adopted from Chen et al. (2023))

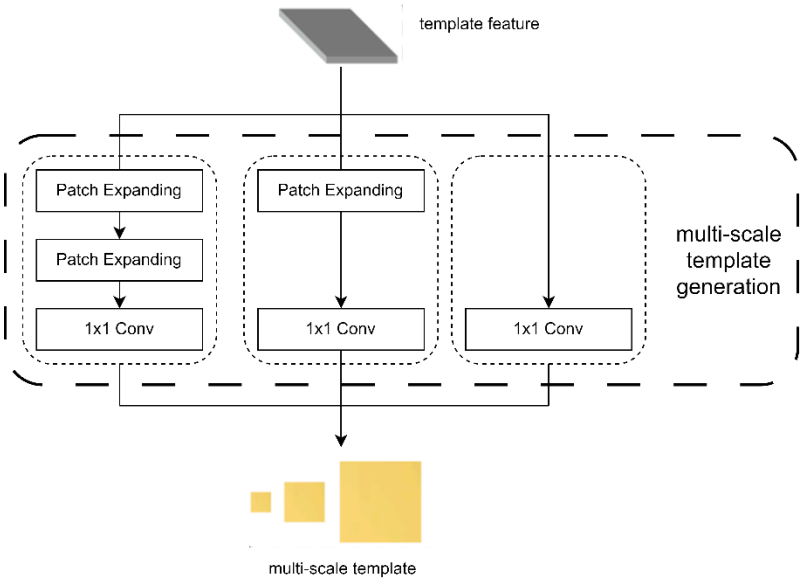


Figure 2.33: Structure of MSTG (adopted from Chen et al. (2023))

The structure of STD is shown in Figure 2.31. STD is built by three submodules that contain two STBs and one PE, followed by another submodule that contains only two STBs. The main objective of STD is reconstructing the template feature by leveraging the fused high-level feature map to augment MST-GAN's modelling capability in capturing the intrinsic patterns of qualified samples. The structure of PE is shown in Figure 2.32. Instead of transposed convolution, pixel shuffling was chosen as an upsampling technique in PE, as the zero padding in transposed convolution causes interference pixels to appear in the rebuilt image, which hinders the accuracy of the subsequent inspection process. Pixel shuffling effectively suppresses interfering pixels in the reconstructed image. Template feature produced by STD is reconstructed via three pathways in MSTG, as shown in Figure 2.33, resulting in a multi-scale defect-free template.

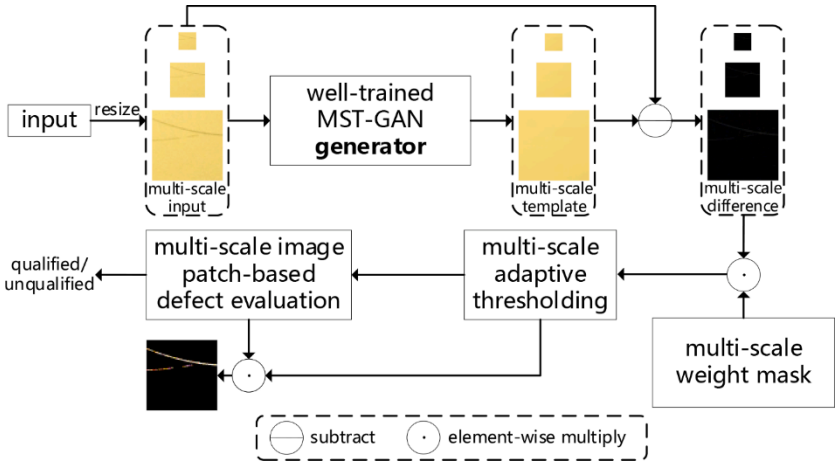


Figure 2.34: Flowchart of the inspection process (adopted from Chen et al. (2023))

Figure 2.34 illustrates the inspection process. The multi-scale input is subtracted from the multi-scale defect template produced by well-trained MST-GAN, resulting in a multi-scale difference image. The multi-scale average feature map for the three distinct scales is derived by averaging the multi-scale difference images with a collection of qualified training dataset images. Subsequently, the multi-scale averaged feature map is inverted and normalised. The process involves upsampling the averaged feature maps from the lower scale and subsampling those from the higher scale. This is to fuse the averaged feature maps of each scale with the average feature maps of the other scales, resulting in the creation of the multi-scale weight mask. In particular, a mid-scale weight mask is generated by combining the average feature map at the mid-scale level with the subsampled large-scale average feature map and the upsampled small-scale average feature map. Similarly, a large-scale weight mask is generated by combining the large-scale average feature map with the upsampled small-scale average feature map and the upsampled mid-scale average feature map. The average feature map at the low-scale level is merged with the subsampled large-scale feature map and the subsampled mid-scale average feature map to yield a small-scale weight mask. In order to generate a weighted difference image with small reconstruction errors and interfering pixels, element-wise multiplication is applied between multi-scale weighted mask and multi-scale difference image. The multi-scale adaptive thresholding process involves the application of varying threshold values to the weighted

difference image across different scales and produces a thresholded multi-scale difference image. This image highlights potential defects that might be present in the weighted difference image. The thresholding value is computed using the weighted difference image's local means and standard deviations across different scales. The multi-scale image patch-based defect evaluation process involves the calculation of the defect probability of individual image patches by applying the sliding windows strategy to the thresholded multi-scale difference image. A threshold value is applied to calculate defect probability, and a binary defect probability map for each image patch is generated. The applied threshold is determined by taking into account the dimensions of the image, the size of the sliding window, and a sensitivity factor of 0.0001. By computing scores for all patches within the defect probability map at the three distinct scales, the defect evaluation score is obtained. To determine whether the inspected sample is defective, the defect evaluation score obtained is compared with the highest defect assessment score derived from the training dataset.

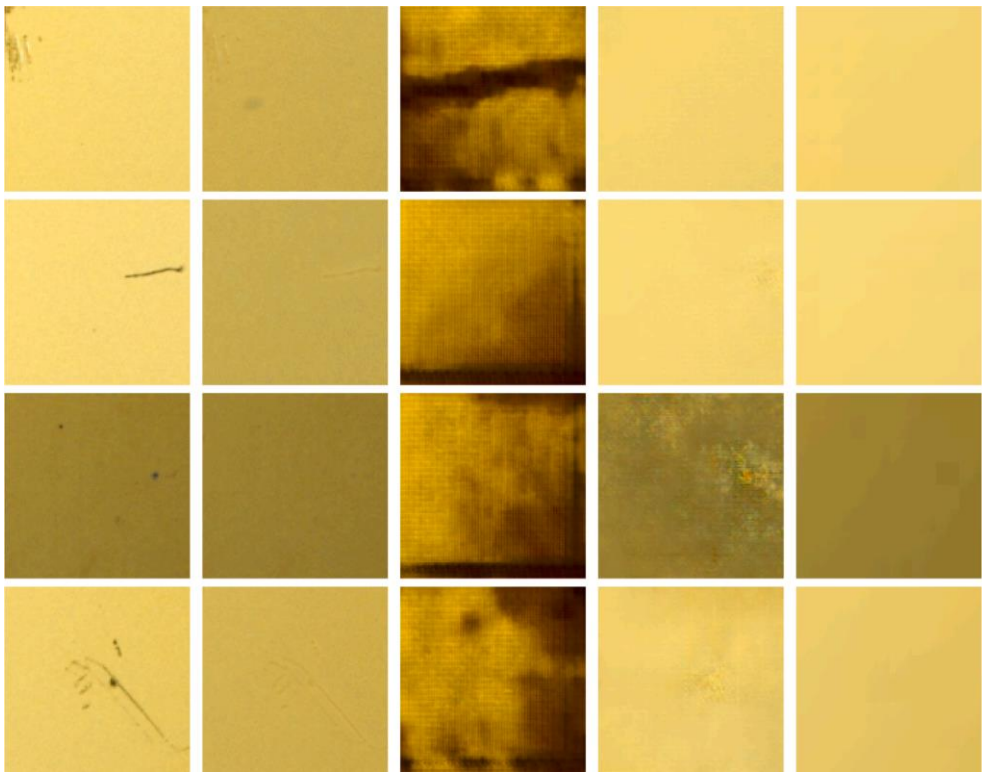


Figure 2.35: The visual output obtained from various GAN models (Chen et al. (2023))

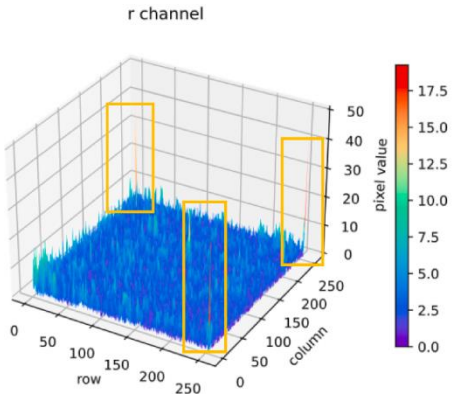


Figure 2.36: Pixel value distribution after applying transposed convolution (adopted from Chen et al. (2023))

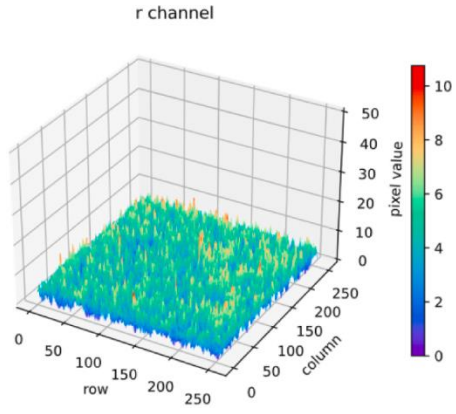


Figure 2.37: Pixel value distribution after applying pixel shuffle (adopted from Chen et al. (2023))

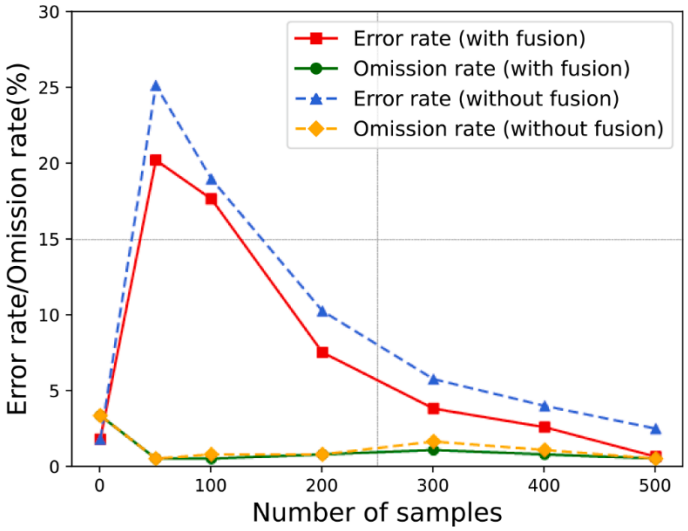


Figure 2.38: Performance of the multi-scale weight mask Inspection Framework (adopted from Chen et al. (2023))

Table 2.9: Results of performance comparison among various inspection methods (adopted from Chen et al. (2023))

Methods	Error rate (%)	Omission rate (%)	Accuracy (%)	P (%)	R (%)	F	FPS
CycleGAN	6.25	25.5	82.7	85.2	93.8	0.893	20.4
DiscoGAN	1.14	98.0	75.9	76.5	98.9	0.863	19.6
GANomaly	57.4	69.6	39.6	66.3	42.5	0.518	**186**
Skip-GANomaly	9.68	92.6	70.6	75.9	90.3	0.825	124
DifferNet	5.98	6.53	93.8	97.9	94.0	0.959	48.0
GAN-based template	28.9	2.84	77.2	98.8	71.0	0.826	22.2
MST-GAN	**0.70**	**0.57**	**99.3**	**99.8**	**99.3**	**0.996**	70.9

Figure 2.35 shows the defect-free templates generated by different GAN models. The first column shows the original sample, followed from left to right by the output of CycleGAN, DiscoGAN, the GAN-based template, and the MST-GAN. Compared to the other models, MST-GAN shows the strongest ability to extract the desired intrinsic patterns as the other models have their own drawbacks. CycleGAN's outputs come with different contrast levels, DiscoGAN fails to extract the desired intrinsic patterns, and the GAN-based templates also fail at some point. Figure 2.36 and Figure 2.37 show the performance of transposed convolution and pixel shuffle. As shown by the rectangular boxes in Figure 2.36, transposed convolution introduces some

interfering pixels in the corners of the differential image, while pixel shuffle does not. The impact of the multi-scale weight mask on inspection performance is illustrated in Figure 2.38. With weighted masks, there is a lower error rate and omission rate compared to the framework without weighted masks. Compared to other SOTA inspection methods, the framework achieved the best performance of 0.996 f1 score, as shown in Table 2.9.

The advantage of this framework is that defect-free templates can be automatically generated by capturing the intrinsic quality patterns of the input image in order to achieve extremely high accuracy. However, the drawback of requiring a relatively high amount of computation and memory to run the model is also apparent, which leads to the need for a more powerful machine to run the model.

2.3.3 Defect detection with deep learning: object detection approach

The following section will review defect detection methods that utilise defect detection approaches.

2.3.3.1 Die-level defect detection or classification system using R-CNN

Table 2.10: Distribution of datasets (adopted from You et al. (2022))

Defect Type	Training Set	Validation Set	Testing Set
Defect-less	0	0	4
Blob	5	5	12
Die crack	5	3	4
Pin hole	5	5	5
Underfill	5	5	10

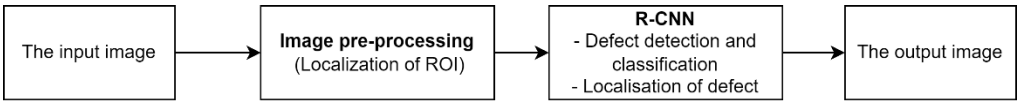


Figure 2.39: Workflow of the system (adopted from You et al. (2022))

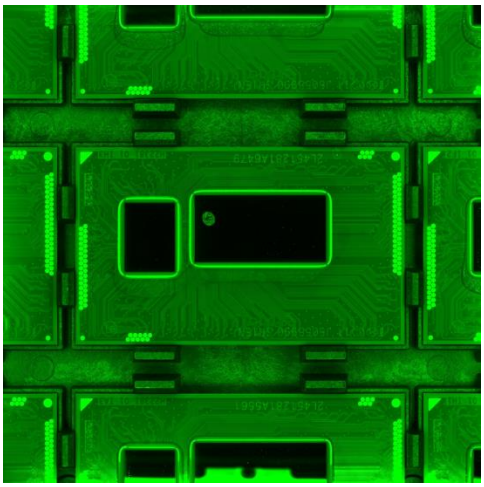


Figure 2.40: Example image of defective semiconductor unit (adopted from You et al. (2022))

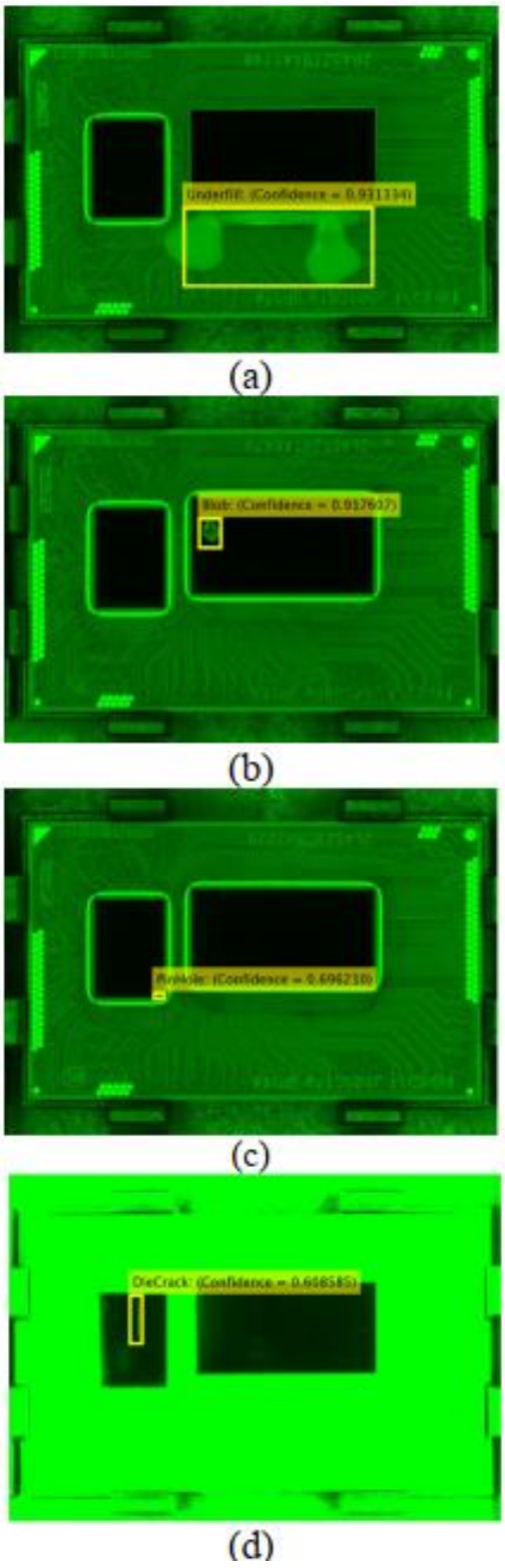


Figure 2.41: The R-CNN's classification output reveals the position of the defect(s), defect type, and the level of confidence in the classification. The images (a-d) depict examples of die crack, pinhole, blob, and underfill (adopted from You et al. (2022))

Table 2.11: Detection and Classification Accuracy of R-CNN (adopted from You et al. (2022))

Detection	R-CNN
Average accuracy	**88.5%**
Classification	
No defect	75.0%
Blob defect	66.7%
Die crack defect	25.0%
Pin hole defect	50.0%
Underfill defect	100.00%
Average accuracy	**71.4%**

You et al. (2022) proposed a die-level defect detection or classification system using the R-CNN. Table 2.10 presents the dataset's available images per defect type. Figure 2.39 shows the workflow of the system. The input images are gone through the image pre-processing step to localise the region of interest (ROI). Although the input image may contain multiple semiconductors, the region of interest (ROI) being analysed is a single semiconductor unit, as shown in Figure 2.40. Some of the ROI is localised manually due to high exposure in the image. The pre-processed image is passed to the R-CNN to perform defect detection and classification. The pre-trained AlexNet model is used as the RCNN's basis network for transfer learning. To suit the dataset, AlexNet's SoftMax layer was modified. The semiconductor is detected as "defect-less" or "defect", and the defect in the semiconductor is further classified, as shown in Figure 2.41. Table 2.11 shows the result of detection and classification, where 31 out of 35 were successfully detected as "defect" or "defect-less", achieving 88.5% of average detection accuracy.

As a two-stage classifier, which is often slower and has lower frame rates, the RCNN approach is less suited for real-time inspection applications. Furthermore, the RCNN in the system has low accuracy (71.4 %), further limiting its effectiveness in defect detection. Another disadvantage of this system is that it requires manual capture of ROI.

2.3.3.2 Deep convolutional network based on the YOLOv5 for IC defect detection.

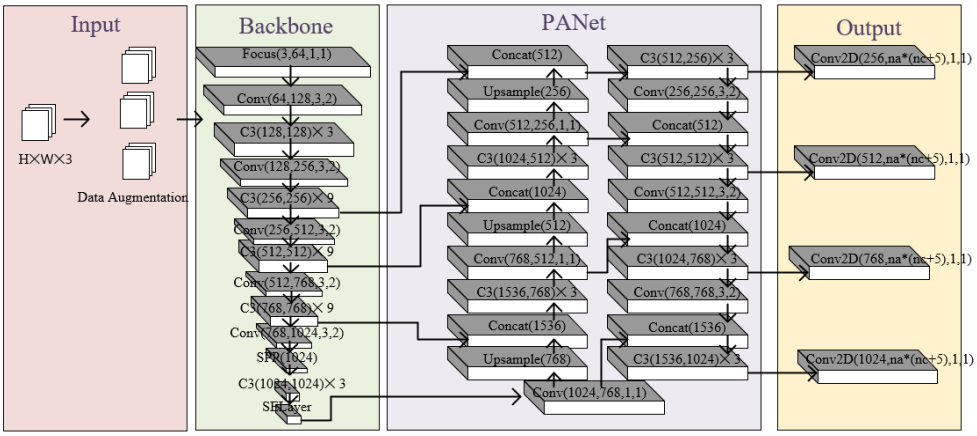


Figure 2.42: Modified YOLOv5x architecture (adopted from Lu et al. (2022))

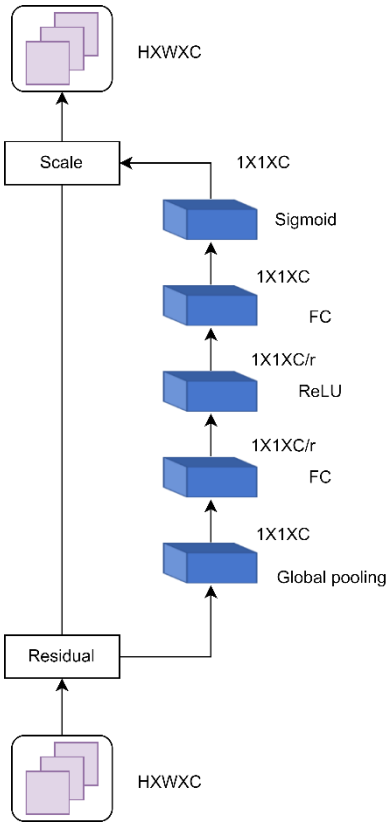


Figure 2.43: SELayer architecture (adopted from Lu et al. (2022))

Table 2.12: Defect types in the dataset and number of images per defect type (adopted from Lu et al. (2022))

Defect Type	number
Chipping	953
Gold layer scratches	762
Excess glue	786
Gold layer defect	1008
Graphic defect	994
Gold layer particles	965
Heterochromatic	826
Crack	983
Bridge deformation	990
Side slot offset	1020
Side groove different	987
Edge groove gold layer	966
Graphic scratches	1000
Different colours of gold	989

Table 2.13: Comparison of performance in the test set (adopted from Lu et al. (2022))

Methods	mAP@0.5	mAP@0.1
SSD	4.02%	27.40%
Faster rcnn	49.38%	67.75%
Efficientdet	52.58%	54.89%
Yolov5	94.90%	97.77%
Yolov5 + SE	95.40%	97.79%

Table 2.14: Performance comparison between models with and without SELayer (adopted from Lu et al. (2022))

Defect type	P		R		mAP@0.5	
	1	2	1	2	1	2
Chipping	0.973	0.97	0.959	0.958	0.974	0.965
Gold layer scratches	0.99	1	0.996	1	0.995	0.995
Excess glue	0.681	0.681	0.598	0.618	0.608	0.622
Gold layer defect	0.997	0.996	1	1	0.995	0.995
Graphic defect	0.986	0.964	0.949	0.967	0.983	0.982
Gold layer particles	0.993	0.987	0.991	0.987	0.995	0.995
Heterochromatic	0.81	0.818	0.854	0.884	0.855	0.876
Crack	0.916	0.909	0.961	0.965	0.965	0.977
Bridge deformation	0.995	0.995	1	1	0.995	0.995
Side slot offset	1	1	0.956	0.995	0.967	0.994
Side groove different	0.998	0.998	1	1	0.995	0.995
Edge groove gold layer	0.997	0.998	0.969	0.972	0.977	0.977
Graphic scratches	0.982	0.978	0.991	0.992	0.995	0.995
Different colours of gold	0.998	0.996	0.992	0.992	0.995	0.995
Average					0.949	**0.954**

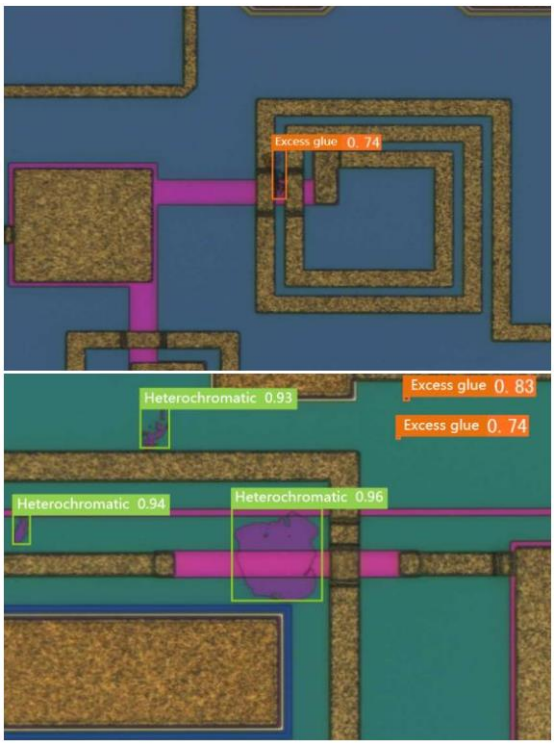


Figure 2.44: Visualisations of test results for the model with SELayer in the test set (adopted from Lu et al. (2022))

In 2023, Lu et al. proposed a deep convolutional network based on the YOLOv5 for IC defect detection. The network is built upon the YOLOv5x architecture incorporating SELayers following the feature layer of the last scale in the backbone and expanding PANet from the original three scales to four scales. The addition of the fourth scale in the PANet results in the creation of four detection heads instead of the three detection heads present in the original YOLOv5x. Figure 2.42 shows the network architecture of the modified YOLOv5x. The main purpose of the SELayer module is to explicitly model channel interdependencies or to capture the channel relationship, as shown in Figure 2.43. Global average pooling is performed on the input feature map to squeeze the feature map to the size of $1 \times 1 \times C$, then fed into two fully connected layers and sigmoid function, learning weight to explicitly model channel association and to produce the channel-wise weight factor, excitation the feature map. The channel weight is applied to the original feature layer by element-wise multiplication, reweighting the feature map. The dataset contains 13329 images. Table 2.12 show the defect type and the number of images. The dataset was split into three sets for training, validation, and testing. Table 2.13 shows the modified model performance in the test dataset's result compared to other models. YOLOv5x with SELayer had the highest accuracy, 95.40 of mAP@0.5, 0.5% better than the YOLOv5x without SELayer. Table 2.14 presents the detailed accuracy results for each defect type, comparing the YOLOv5x with and without the SELayer, the label "1" indicates the model without the SELayer. Figure 2.44 depicts some of the output images from the inference process on the testing dataset.

The advantage of the modified model is that it is substantially more accurate. Being based on YOLOv5x, the biggest model in the YOLOv5 family and the one with the highest computational cost, which leads to longer training and inference periods, is one of its drawbacks.

2.3.3.3 Optimising YOLOv7 for Semiconductor Line Space Pattern Defect Detection

Table 2.15: Distribution of datasets (adopted from Dehaerne et al. (2023b))

Sample counts	Train	Validation	Test
Line collapse	550	66	76
Bridge	238	19	17
Microbridge	380	47	78
Gap	1046	156	174
Probable Gap	315	49	54
Total instances	**2529**	**337**	**399**
Total images	**1053**	**117**	**154**

Table 2.16: Selected hyperparameter for experiment with default and modified value (adopted from Dehaerne et al. (2023b))

Type	**Hyperparameter**	**Default**	**Modified (1)**	**Modified (2)**
Weight & learning	Anchor threshold	4	9	13
	Number of anchors	3	9	13
	IOU threshold	0.2	0.5	0.75
	Object loss gain	0.7	0.25	0.5
	Class loss gain	0.3	0.1	0.5
	Box loss gain	0.05	0.1	0.25
	Focal-loss gamma	0.0	0.1	1.5
	Freeze backbone layers	First layer only	First 25 layers	All 50 layers
	Model size	Base	Tiny	Base-X
Data augmentation	Vertical Flipping (probability)	0.0	0.5	-
	Horizontal Flipping (probability)	0.5	0.0	-
	Mosaic	1.0	0.0	0.5
	Scale (+/- gain)	0.5	0.25	0.75
	Translation (+/- fraction)	0.2	0.0	0.5
	Angle (+/- degrees)	0	45	90
	Shear (+/- degrees)	0	15	30
	HSV (fraction)	0.015/0.7/0.4 (h/s/v)	0.0 (all)	1.0 (all)

Table 2.17: Results of model performance with different model hyperparameters on test images (adopted from Dehaerne et al. (2023b))

Hyperparameter	**Value**	**AP@0.5**					
		microbridge	**gap**	**bridge**	**Line collapse**	**p-gap**	**mAP**
Default		**0.873**	**0.967**	0.602	**1.000**	0.508	**0.790**
Anchor threshold	9	0.806	0.950	0.639	1.000	**0.529**	0.785
	13	0.792	0.958	0.537	1.000	0.238	0.705
Anchors	9	0.726	0.948	0.587	1.000	0.167	0.686
	13	0.766	0.948	0.477	0.000	0.103	0.574
IOU threshold	0.1	0.737	0.950	0.590	1.000	0.150	0.685
	0.75	0.807	0.959	0.609	1.000	0.163	0.708
Object loss gain	0.25	0.754	0.949	0.581	1.000	0.274	0.712
	0.5	0.800	0.959	0.750	1.000	0.275	0.757
Class loss gain	0.1	0.737	0.950	0.590	1.000	0.150	0.685
	0.5	0.803	0.958	0.583	1.000	0.457	0.760
Box lose gain	0.1	0.762	0.959	0.562	1.000	0.106	0.678
	0.5	0.800	0.959	0.750	1.000	0.275	0.757
Focal-loss gamma	1.0	0.635	0.890	0.652	0.980	0.000	0.631
	1.5	0.581	0.851	0.505	1.000	0.000	0.587
Freeze layers	25	0.712	0.919	0.584	1.000	0.247	0.693
	50	0.745	0.949	0.579	1.000	0.139	0.682
Model size	Tiny	0.746	0.960	**0.819**	1.000	0.281	0.761
	Base-X	0.821	0.960	0.515	1.000	0.191	0.697

Table 2.18: Results of model performance with different data augmentation parameters on test images (adopted from Dehaerne et al. (2023b))

Hyperparameter	**Value**	**AP@0.5**					
		microbridge	**gap**	**bridge**	**Line collapse**	**p-gap**	**mAP**
Default		**0.873**	0.967	0.602	**1.000**	0.508	0.790
Vertical Flipping	0.5	0.709	0.960	0.790	1.000	**0.604**	**0.812**
Horizontal Flipping	0.0	0.722	0.959	0.718	1.000	0.507	0.781
Mosaic	0.0	0.647	0.952	0.581	1.000	0.030	0.642
	0.5	0.780	0.949	0.589	1.000	0.277	0.719
Scale	0.25	0.822	0.949	0.437	1.000	0.288	0.699
	0.75	0.758	0.939	0.634	1.000	0.133	0.693
Translation	0.0	0.784	**0.968**	0.540	1.000	0.107	0.680
	0.5	0.808	0.940	0.457	1.000	0.195	0.680
Angle	45	0.633	0.959	**0.912**	1.000	0.268	0.754
	90	0.597	0.899	0.745	1.000	0.055	0.659
Shear	15	0.779	0.967	0.548	1.000	0.277	0.714
	30	0.785	**0.968**	0.575	1.000	0.346	0.735
HSV	0.0	0.781	0.949	0.586	1.000	0.326	0.729
	1.0	0.677	0.949	0.584	1.000	0.197	0.681

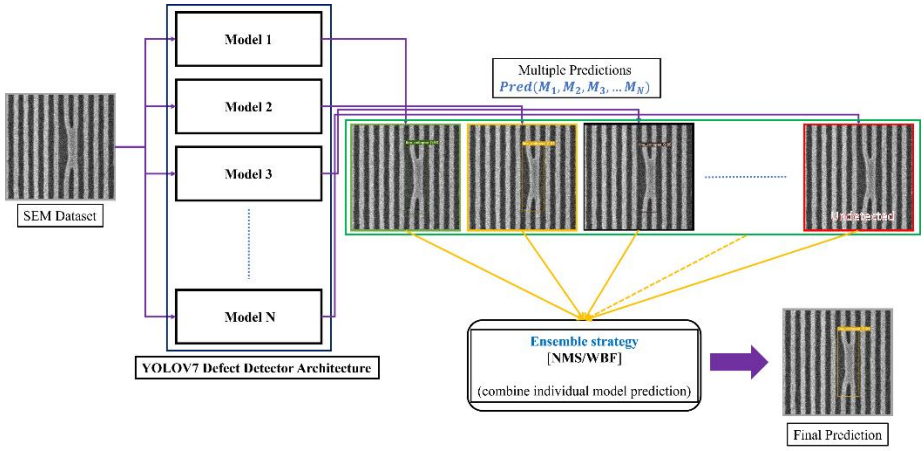
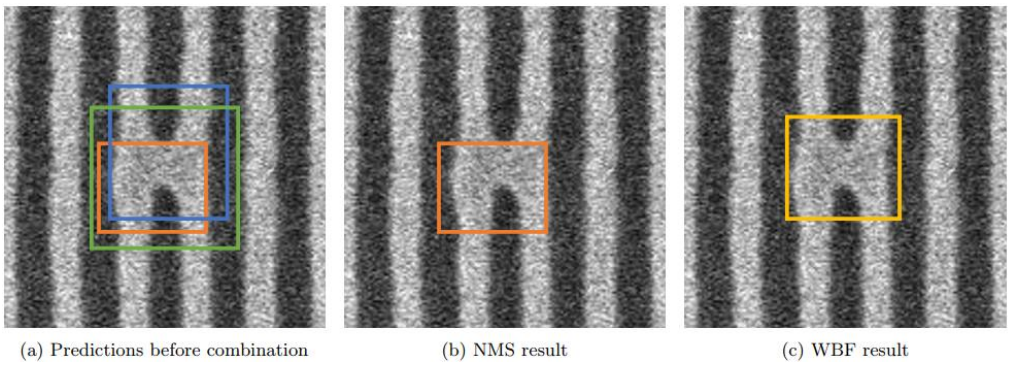


Figure 2.45: Illustration of ensemble model (adopted from Dehaerne et al. (2023))



(a) Predictions before combination (b) NMS result (c) WBF result

Figure 2.46: Example of NMS and WBF (adopted from Dehaerne et al. (2023))

Table 2.19: Results of ensemble model (adopted from Dehaerne et al. (2023b))

Models	**Prediction Combination**	**AP@0.5**					
		microbridge	**gap**	**bridge**	**Line collapse**	**p-gap**	**mAP**
Default	NMS	0.873	0.967	0.602	1.000	0.508	0.790
	WBF	0.709	0.960	0.790	1.000	0.604	0.812
Default, Tiny, Base-X	NMS	0.849	0.968	0.760	1.000	0.546	0.825
	WBF	0.852	0.968	0.823	1.000	0.565	0.842
Default, Vertical Flipping, Angle	NMS	0.877	**0.969**	0.809	1.000	0.634	0.858
	WBF	**0.878**	**0.969**	**0.850**	1.000	**0.642**	**0.868**

Dehaerne et al. (2023) optimised the YOLOv7 by training and evaluating various models with different hyperparameters to enhance the detection precision of semiconductor line space pattern defects. Table 2.15 shows the detail and distribution of the dataset, which contains 3265 images obtained from

scanning electron microscopy. The model's hyperparameters that were chosen for testing and experimentation were believed to have a substantial influence on detection performance. Table 2.16 shows the hyperparameter that was chosen for the experiment in terms of the model hyperparameter and data augmentation parameter.

Table 2.17 presents the Average Precision (AP) outcomes on test images for models with distinct model hyperparameters. Some models perform better than others for certain defect classes, such as bridge and p-gap. The Tiny model, in particular, achieves an AP of 0.819 for the bridge class. However, the overall mean AP of the model is lower than that of the default model. According to the results presented in Table 2.18, which displays the Average Precision (AP) outcomes on test images for models with distinct data augmentation hyperparameters, it was found that using vertical flipping with a value of 0.5 can increase the Average Precision (AP) for bridge classes to 0.790 and the mean Average Precision (mAP) to 0.812, which is an improvement compared to the default model.

An ensemble technique was utilised to enhance mean Average Precision (mAP) performance by combining multiple models that produced the highest Average Precision (AP) for distinct defect classes. Figure 2.45 provides a conceptual illustration of the ensemble model. Moreover, a more advanced prediction combination method, weighted box fusion (WBF), which takes a weighted average of each box in a group to create the final prediction box, is being tested by replacing the original Non-Maximum Suppression (NMS) method. Figure 2.46 shows example prediction results of NMS and WBF. A collection of models with varying hyperparameter values that exhibited optimal per-class performance were assembled, comprising the default model in addition to models with vertical flipping and a 45-degree angle data augmentation technique. The ensemble model utilised the weighted box fusion (WBF) method and resulted in the highest AP for all defect classes and achieved the best mAP, as presented in Table 2.20.

The ensemble model's benefit is the great accuracy it achieves by integrating the predictions of other models. Nevertheless, training numerous models and refining hyperparameters take a lot of time. To explore and

determine which hyperparameter values are suited for the best results, repeatedly training models with various hyperparameters is necessary.

2.4 Comparison

No	Author	Title	Technique used	Hyperparameter	Strength/Limitations	Result	Future Work
1	Shankar and Zhong, 2005	Defect detection on semiconductor wafer surfaces	- Image processing • Reference Method/Template-based Method	-	-Defect tolerance: able to distinguish between critical and non-critical defects via defect specification rule. -High-quality reference image is a prerequisite	-	-
2	Yeh et al., 2010	A Wavelet-Based Approach in Detecting Visual Defects on Semiconductor Wafer Dies	- Image processing • Two-dimensional wavelet transform (2-D WT)	-	-Non-reference method: does not rely on pixel-by-pixel matching. -Does not require the training process -Need to determine the proper parameter. -Cannot classify the defect into defect categories.	-	-Develop Simple classification to further differentiate the discovered defective pixels based on their geometrical coordinates or characteristics.
3	Cheon et al., 2019	Convolutional Neural Network for Wafer Surface Defect Classification and the Detection of Unknown Defect Class	- Deep learning: classification • CNN - Clustering model • k-NN - Data augmentation	- SDG with LR: 0.001 - Batch size: 32	- High accuracy - Able to identify the unknown defect - Long training time and reference time	- Train accuracy: 99.4% - Valid accuracy: 98.7% - Test accuracy: 96.2%	- Implement an unsupervised cluster model to create a cluster when a new image is collected

4	Yuanfu Yang and Sun, 2022	Semiconductor Defect Pattern Classification by Self-Proliferation-and-Attention Neural Network	- Deep learning: Classification • SP&A-Net: Self-Proliferation-and-Attention Block	- NAG descent	- High accuracy - Able to capture spatial-wise relationships in the feature map. - Generate extra feature maps at a low cost. - Requiring fewer parameters and floating-point operations	- AEI dataset accuracy: ≈97.5% - ADI dataset accuracy: ≈97.9% - API dataset accuracy: ≈96.9% - CIFAR-10 accuracy: 92.93% - IMAGENET Top-1 error: 23.06%	-
5	Chen et al., 2023	Multi-scale GAN with transformer for surface defect inspection of IC metal packages	- Deep learning: Classification • MST-GAN	- AdamW with LR: 0.0002 - Momentums $\beta1 = 0.9$ and $\beta2 = 0.999$ - Weight decay: 0.01 -Batch size: 8	- High accuracy - Able to generate the defect-free template - Suppress the reconstruction errors via pixel shuffle - Computationally and memory intensive	- F1: 99.6% - Accuracy: 99.3% - Precision: 99.8% - Recall: 99.3% - FPS: 70.9	- Integrate the framework into an analysis of the correlation between neighbouring image patches. - Develop a GAN that will be capable of generate defect-highlighting templates while preserving the original texture details. - Explore effective methods for network pruning and quantization while minimizing the degradation in network performance
6	You et al., 2022	Die-Level Defects Classification using Region-based Convolutional Neural Network	- Deep learning: Object detection • R-CNN - Data augmentation	- Mini Batch Size: 15 - LR: 0.000001 - Max epochs: 80	- Low accuracy - Manual capture of the region of interest (ROI) is required for high-exposure image is needed	- Accuracy for detection: 88.5% - Accuracy for classification: 71.4%	- Train the network with more datasets - Find the most appropriate CNN model for classifying die images - Consider detecting more defect type
7	(Lu et al., 2022)	Defect Detection of Integrated Circuit Based on YOLOv5	- Deep learning: Object detection	- Batch Size: 16 - Epoch: 600 - LR: 0.001	- High accuracy - Require high computational cost and large memory	- mAP@0.5: 95.40% - mAP@0.1: 97.79%	- Develop a lightweight model that is suitable for deployment in an industrial scenario

			• Modified YOLOv5x with SELayer	- Cosine annealing strategy			
8	Dehaerne et al., 2023b	Optimizing YOLOv7 for Semiconductor Defect Detection	- Deep learning: Object detection • Ensemble model of optimized YOLOv7 using WBF - Data augmentation	- Batch Size: 2 - Epoch: 200 - Weights: refer to Table 2.16	- Relatively high accuracy - Requires a significant amount of time and resources to train the model with different hyperparameters	- mAP@0.5: 86.8%	- Enhance the outcomes by implementing advanced hyperparameter optimization techniques

2.5 Overview of the object detection model

2.5.1 R-CNN

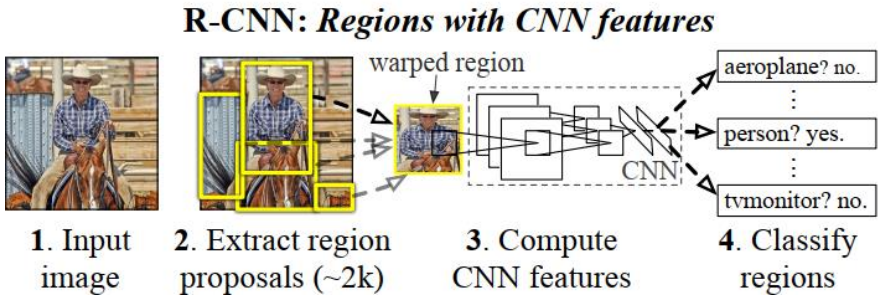


Figure 2.47: Overview of R-CNN (adopted from Girshick et al. (2013))

RCNN can be said to be the pioneer of target detection using deep learning. Prior to R-CNN, object detection relied heavily on manually designed feature extractors, such as the Viola-Jones detector and Histogram of Oriented Gradients (HOG). These models were slow, inaccurate, and performed poorly (Zaidi et al., 2021). The accuracy of traditional object detection models was only able to achieve around 33.7 in the VOC 2007 dataset. However, R-CNN showed very good performance, achieving an accuracy of 58.5% (Zou et al., 2019). R-CNN convert detection into classification and localisation problem. The R-CNN model can be divided into four modules, which are region proposal, feature extraction, classification, and bounding box regression. Below is the algorithm flow of R-CNN, as illustrated in Figure 2.47. Step 1 (corresponds to region proposal), almost 2,000 candidate regions will be generated using the selective search algorithm. In step 2 (corresponding to feature extraction), the candidate region is cropped to 227*227 pixels, and a deep neural network, acting as the backbone network, is used to extract features from these candidate regions and generate a feature vector of 4096 dimensions for each candidate region. In step 3 (corresponding to classification), the feature vector is passed to the Support Vector Machine (SVM) classifier to obtain scores, specifying the class to which the candidate region belongs. NMS is applied based on the interception over union (IOU) and threshold to remove some overlapping candidate boxes. In step 4 (corresponding to bounding box regression), the class-specific bounding box regressor is applied to perform regression operations on the remaining candidate region. (Girshick et al., 2013).

2.5.2 Fast R-CNN

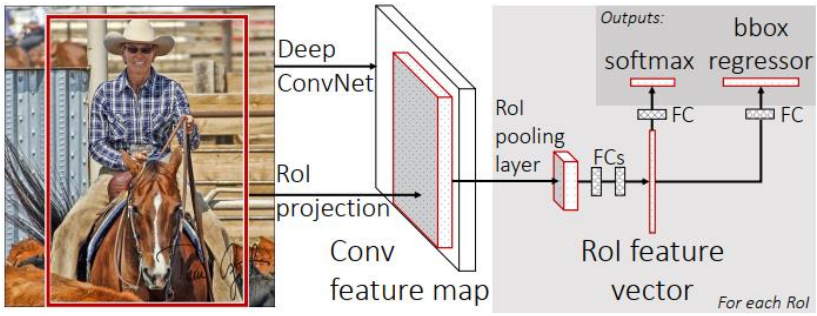


Figure 2.48: Overview of Fast R-CNN (adopted from Girshick (2015))

Unlike R-CNN, Fast R-CNN combines feature extraction, classification, and bounding box regression in a single neural network, as illustrated in Figure 2.48, instead of using separate neural networks, SVM, and regressor for each task (Girshick, 2015). Similar to R-CNN, Fast R-CNN also uses selective search to generate candidate regions from the input image. The input image is fed into the deep convolutional network to obtain the corresponding feature map. Then, the candidate regions generated are projected onto the feature map. The feature map will then pass through the ROI pooling layer, which scales the candidate region to a uniform size and extracts a fixed-length feature vector from it (Ahmed Fawzy Gad, 2021). The ROI pooling layer in Fast R-CNN is inspired by the SPPnet, which enables Fast R-CNN to accept arbitrary-sized inputs. In the ROI pooling layer, each candidate region in the feature map is divided into a fixed grid of cells, such as 7×7, and max pooling is performed on each grid to obtain the fixed-length feature vector. The fixed-length feature vector is then passed through two fully connected layers to obtain the ROI feature vector. ROI feature vector will pass through two fully connected layers that are connected in parallel. One includes the SoftMax layer used for the class score prediction and another for bounding box regression parameter prediction, which generates four real-valued numerical values corresponding to each of the K classes of objects (Girshick, 2015). The SoftMax layer outputs the probability of $K + 1$ categories, where K is the number of classes and "1" indicates the probability of the candidate region as the background. Fast R-CNN addresses the issue of training many systems independently in R-CNN, allowing calculations to be shared in

the feedforward process (Zaidi et al., 2021). Compared to the prior R-CNN model, Fast R-CNN is a faster and more accurate object detection model. Using the VOC 2007 dataset, it was almost 200 times faster than R-CNN while increasing accuracy from 58.5% (R-CNN) to 70% (Zou et al., 2019).

2.5.3 Faster R-CNN

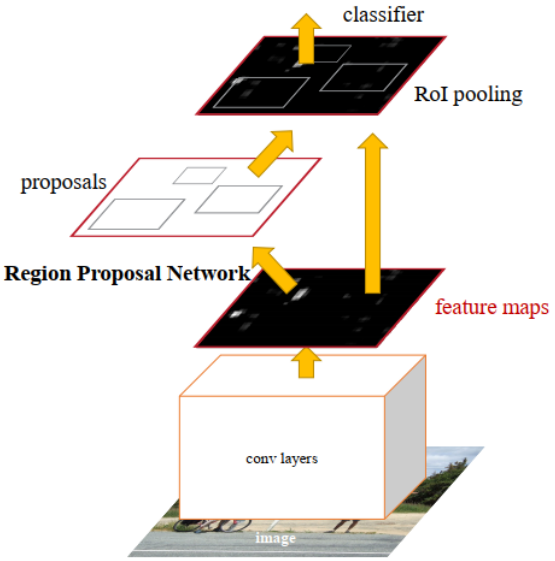


Figure 2.49: Overview of Faster R-CNN (adopted from Ren et al. (2015))

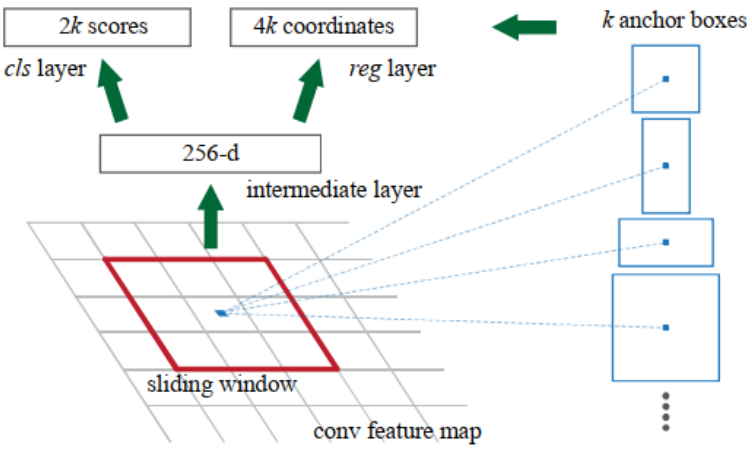


Figure 2.50: Overview of RPN (adopted from Ren et al. (2015))

Faster R-CNN is a two-part architecture consisting of an RPN and Fast R-CNN, as shown in Figure 2.49 (Ren et al., 2015). While Fast R-CNN uses selective search to generate candidate regions, it limits its speed as each image takes

around two seconds to produce the candidate regions (Zaidi et al., 2021). RPN replaces selective search and allows for end-to-end training. A backbone network, often composed of convolutional layers, will receive the input picture and generate the feature map. This backbone network is shared between the RPN and Fast R-CNN components in Faster R-CNN. The RPN takes the feature map generated by the backbone as input and applies a sliding window approach. A small network is applied to each sliding window on the feature map. For each sliding window, the centre point in the sliding window corresponding to the centre point on the original image is calculated, and k anchor boxes are applied to the centre point. Using anchor boxes with various scales and aspect ratios allows for the detection of objects of various sizes and shapes. Typically, three scales and three aspect ratios are used, resulting in a total of nine anchor boxes per sliding window position. The sliding window is then mapped to a lower-dimensional feature vector (e.g., 256-dimension for ZF backbone) and fed into two fully connected layers, as shown in Figure 2.50. The first fully connected layer produces $2k$ scores. The first score corresponds to the probability of the anchor box being a background, while the second score corresponds to the probability of it being an object. In order to more precisely position and enlarge each anchor box to match the object, the second fully connected layer generates the $4k$ regression scores. Cross-boundary anchors are ignored, NMS is applied to the candidate regions based on their classification scores and top-scoring proposals boxes are selected to form the final set of candidate regions. The candidate regions produced by the RPN are projected onto the shared feature map and processed using the same Fast R-CNN architecture as the original Fast R-CNN (Ren et al., 2015). To Summarise, Faster R-CNN achieved nearly real-time object identification by introducing RPN to address the slow region proposal generation in Fast R-CNN. With the use of ZF-net, Faster R-CNN was able to achieve 17 FPS on a K40 GPU. Faster R-CNN Achieved a mean average precision of 42.7% in COCO datasets and 73.2% in the VOC 2007 dataset, outperforming Fast R-CNN in terms of accuracy (Zou et al., 2019).

2.5.4 SSD

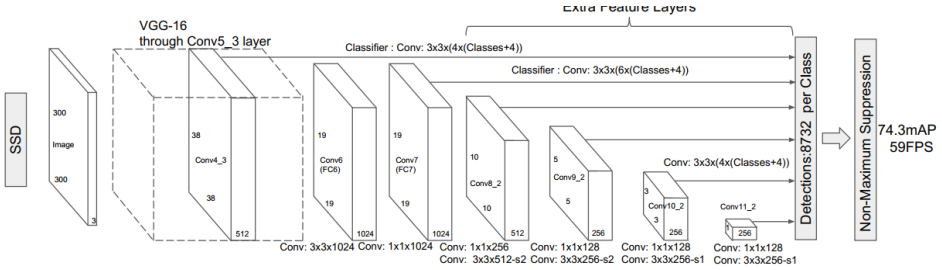


Figure 2.51: Network architecture of SSD (adopted from Liu et al. (2015))

The SSD uses a 300*300 input picture as the input and a modified or truncated VGG-16 as the backbone to extract feature maps. These feature maps then pass through an auxiliary structure consisting of different-sized convolutional layers, which progressively decrease in size and output feature maps of varying scales (Zaidi et al., 2021). A total of six feature maps are generated and responsible for identifying a different size of object, as shown in Figure 2.51. For instance, the larger feature map with a lower receptive field in the first layer retains more detailed information and is used to detect relatively smaller targets. As the level of abstraction increases, smaller feature maps with higher receptive fields are used to detect relatively larger targets. The SSD generates a total of 8,732 default boxes, with each different-sized feature map having different default box scales and aspects, which are determined through calculations and specific conditions. For prediction, a small kernel is applied to each location in the feature map, and acts as a predictor to predict both the offsets of the default boxes and the scores for all object categories. Each location in the feature map uses $(c + 4)k$ filters, requiring a total of $(c + 4)kmn$ filters in the feature map, where k is the number of default boxes, m and n represent the size of the feature map, and c is the number of classes. Thus, each location in the feature map produces ck class scores and $4k$ regression offsets. The process of non-maximum suppression is applied to obtain the final predictions. (Liu et al., 2015). The feature map utilized as input to the RPN and Fast R-CNN architecture in Faster R-CNN is obtained by extracting it through a backbone, in which the receptive in the feature map is quite large, causing the feature map to have lower resolution and lose some of its detailed information (Eggert et al., 2017).

Consequently, Faster R-CNN is not proficient in detecting objects from low resolution, making it unsuitable for identifying small objects (Cao et al., 2019). SSD solve this problem by utilizing detection techniques that involve multiple references and multiple resolutions. The two-stage models, such as Faster R-CNN, are often slower since generating candidate regions takes more time. As opposed to the R-CNN family of models, SSD is a one-stage model that does not propose candidate regions, leading to faster inference times. SSD achieved real-time object detection as it achieved 59 frames on an Nvidia Titan X GPU and achieved a mAP of 76.9% in VOC 2007 datasets (Liu et al., 2015).

2.5.5 YOLO

YOLO was the first one-stage object detector. It converted the object detection problem into a regression problem, where estimating the coordinates of the bounding box, confidence score and class probability. The concept of YOLO is to split the image into a $S \times S$ grid cell, where each cell was responsible for detecting an object if the object's centre fell within it. In other words, each cell estimates the object's centre or whether the object's centre was in that cell. Each cell also predicted B bounding boxes and C class probability scores. Each bounding box predicted five values, which were x, y, w, h representing the location of the centre and the dimensions of the bounding box relative to the grid cell, and a confidence score, which indicated the degree of overlap with the ground truth box, using IOU (Redmon et al., 2015).

YOLOv1 directly predicts the location of the bounding box rather than using anchor boxes. As a result, it could be difficult to detect objects with different aspect ratios. Each grid cell in YOLOv1 produces two bounding boxes, and each cell can only have one class assigned to it. Due to these restrictions, YOLOv1 may have trouble detecting tiny objects that are grouped together. YOLOv2 or YOLO9000 have addressed these limitations. YOLOv2 implement anchor boxes to improve the detection of objects with different aspect ratios and scales, increasing the recall of the model. Moreover, it introduces a passthrough layer that concatenates higher-resolution feature maps with lower-resolution feature maps, helping the model to detect smaller objects more accurately (Redmon and Farhadi, 2016).

To improve accuracy, YOLOv3 adopted a new backbone architecture called Darknet53, which is deeper and more powerful than the Darknet19 used in YOLOv2. In addition, YOLOv3 uses a similar concept to the feature pyramid network (FPN), enabling it to predict across multiple scales. This helps YOLOv3 detect objects of various sizes with higher accuracy (Redmon and Farhadi, 2018).

YOLOv4 uses the CSP-Darknet53 backbone, which increases the model's accuracy while reducing memory costs. Moreover, YOLOv4 introduces advanced data augmentation techniques such as mosaic, which combines four images into one, to expand sample diversity and improve the model's generalisation ability (Bochkovskiy, Wang and Liao, 2020).

The YOLOv5 architecture features the New CSP-Darknet53 backbone, which addresses the issue of duplicate gradients in large convolutional networks, ultimately leading to reduced computational cost. The neck of YOLOv5 is composed of SPPF and New CSP-PAN (Jacob Solawetz, 2020). SPPF is an adaptation of the SPP module used in YOLOv4, achieving the same results but with faster speeds (Ultralytics, 2023).

YOLOv6 builds its backbone and neck based on the Rep-VGG style. The backbone is built using RepBlock and CSPStackRep block, which efficiently uses hardware computing power while maintaining strong feature representation. The neck is Rep-PAN, which adopts a PAN topology with RepBlocks and CSPStackRep to achieve efficient inference while maintaining good multi-scale feature fusion capability. The backbone and Neck in YOLOv6 solve the problem of increased latency and reduced memory bandwidth utilization in previous YOLO versions that used a CSP-based backbone. The head is an efficient decoupled head that maintains accuracy while reducing additional latency overhead (Li et al., 2022a).

The backbone of YOLOv7 is E-ELAN. ELAN controls the shortest and longest gradient paths, allowing the network to learn and converge more effectively. E-ELAN expands, shuffles, and merges cardinality to further increase the learning ability of ELAN. The authors found that the identity connection in RepConv destroys the residual in ResNet, resulting in low accuracy when RepConv is added to ResNet. To address this issue, they

introduced RepConvN, which is RepConv without an identity connection (Wang, Bochkovskiy and Liao, 2022).

2.6 Conclusion

Based on the approaches reviewed, it can be concluded that several methods have been developed for defect detection. However, reference-based methods always require a template, which is a fatal drawback. Even though the 2D wavelet transforms using non-reference methods which do not need to perform pixel-by-pixel matching, the golden image is still needed to perform statistical properties comparison, and proper parameters need to be set, which might require some experience or experimentation to figure out which one is suitable. Most importantly, it cannot further classify the defect pixels into specific categories.

The MST-GAN has solved the problem of these two approaches. It can generate a defect-free template based on the learned intrinsic quality pattern and use a multi-scale strategy to calculate the multi-scale adaptive threshold and some defective measurements. Only a few parameters, such as the threshold and sensitivity factor, need to be determined, and other parameters will be learned through the training process or obtained directly from the image, such as standard deviation and means of the image. However, the golden image is still needed for training purposes, and the MST-GAN is computationally expensive.

The above methods do not mention how the model can detect unknown defects. Cheon et al. proposed a defect classification system that combined CNN and k-NN, which can detect unknown defect classes using a clustering method, but its training and reference time are long. SP&A Net has been proposed by Yuanfu Yang and Sun, requiring fewer parameters and GLOPS while maintaining high accuracy. However, all the approaches above are about classification methods. To perform real-time inference in classification, combining the classification model with an object detection or segmentation algorithm is necessary. The algorithm helps to locate the semiconductor or capture the ROI of semiconductor, while the classification model can then classify the semiconductor based on their identified features. This can result in increased inference time.

In terms of object detection methods, You et al. (2022) proposed an RCNN for die-level detection, but it suffers from low accuracy, and manual capture of ROI is sometimes needed. The speed for RCNN is slow as it is a two-stage algorithm, which is not applicable for real-time inference. Lu et al. proposed a modified YOLOv5x, which achieves high accuracy but requires high computational cost and large memory. An optimized YOLOv7 has been proposed by Dehaerne et al., which uses an ensemble model and reaches relatively high accuracy, but there is still room for improvement in accuracy.

In summary, traditional image processing techniques usually require a reference template and manual parameter tuning. Some deep learning classification methods have high computational costs and long inference times. Object detection methods have varying levels of accuracy, with some suffering from low accuracy (RCNN) and others having room for improvement (ensemble YOLOv7) or requiring high memory costs (modified YOLOv5x). This study aims to develop a deep learning based inspection approach for die defect detection with a high accuracy of 90% , further improve accuracy modifying the model's components. Additionally, the model was optimized for less computational cost by pruning to reduce the number of parameters and make the model as lightweight as possible.

CHAPTER 3

METHODOLOGY AND WORK PLAN

3.1 Development Tool

3.1.1 PyTorch

PyTorch is a Python-based deep learning framework that Meta developed based on the Torch library. Because of its simplicity of use and broad support for GPUs, PyTorch has gained popularity. In the deep learning community, PyTorch has also served as the foundation for a large number of open-source models. The deep learning model used in this study is also based on PyTorch. PyTorch is required to modify components inside the YOLO model and to train and evaluate the model. PyTorch is also necessary to integrate the SAM model into YOLOv5 for ensemble processing.

3.1.2 MMAGIC

MMagic is an advanced and versatile AIGC toolkit based on PyTorch developed by OpenMMLab. It offers various generative models, super-resolution models, and multimodal models. It facilitates model training and fine-tuning. Since the configuration files for models in MMagic are similar, transitioning between different models is seamless. In this study, MMagic was utilized to train StyleGANv2, StyleGANv3, and perform fine-tuning for stable diffusion.

3.1.3 OpenCV

OpenCV is a popular open-source toolkit for computer vision and machine learning, mainly used for a variety of image and video processing tasks. In this study, OpenCV was utilized for image processing purposes. For example, to capture the ROI of the IC.

3.1.4 Visual Studio Code

Visual Studio Code was used as the IDE for making modifications to the source code of YOLO, adding supplementary code and editing the configuration file for deep learning models.

3.1.5 Anaconda

Anaconda is an open-source platform that supports Python and R programming languages. Anaconda was used to create virtual environments on the local machine, and all necessary packages and dependencies required for deep learning model were installed. Anaconda enables the creation of multiple independent environments, each tailored for different GitHub repositories (deep learning models). As different repositories may require varying versions of packages, this approach prevents any conflicts with the global environment.

3.1.6 Label Studio

Label Studio is a versatile open-source data labelling tool that supports multiple types of data. In this study, Label Studio was utilized for annotating images as it supports the YOLO output format. It is important to note that Label Studio will only be installed on the local machine and will not be deployed to an external network or cloud due to the confidentiality of the datasets used in this study.

3.1.7 Albumentations

Albumentations is an efficient and adaptable library for image augmentation. For this study, the Albumentations was used to perform data augmentation to the segmentation dataset and object detection datasets. The Albumentations package will also return the augmented bounding boxes, eliminating the need for reannotation of the augmented images.

3.2 Evaluation metric in object detection

The following section will describe the common evaluation metric used in object detection to evaluate the model's performance.

Table 3.1: Description of evaluation metrics in object detection

True Positive (TP)	Ground truth that is correctly detected by bounding box, based on IoU, IoU threshold, confidence score and confidence score threshold
False Positive (FP)	A bounding box detects the non-existence ground truth (such as detecting the background as an object), or a bounding box detects the wrong object based and IoU and IoU threshold
False Negative (FN)	Ground truth that is not detected by the bounding box
True Negative (TN)	A number of bounding boxes should not be detected. The concept of "TN" is not relevant in object detection, as the model can correctly ignore an infinite number of background regions that do not contain objects.

3.2.1 IoU

IoU, or Intersection over Union, is a common evaluation metric in object detection that measures the degree of overlap between predicted bounding and ground truth bounding boxes. It is calculated as the area of overlap between two boxes divided by the area of their union, as illustrated in Figure 3.1. IoU is used as a criterion to distinguish between TP and FP in object detection. For instance, an IoU threshold of 0.5 is set. If the IoU between a predicted bounding box and its corresponding ground truth bounding box is greater than 0.5, it is considered a TP, indicating a correct detection. Otherwise, it is classified as FP, indicating a false detection (Koech, 2020).

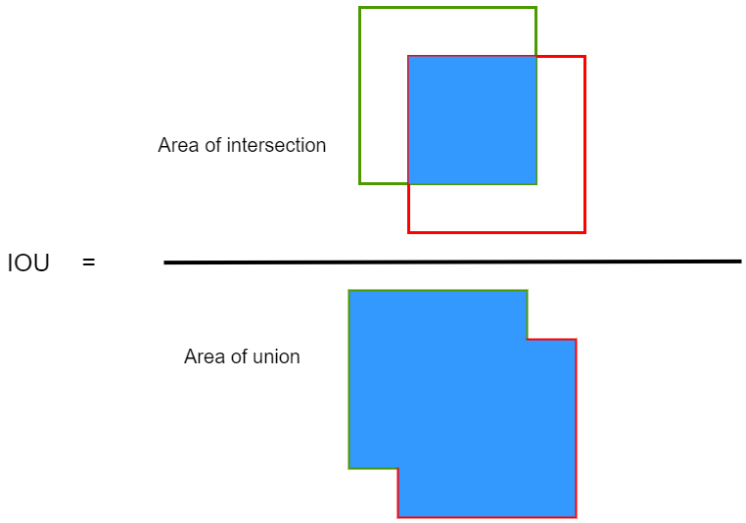


Figure 3.1: Illustration of IoU (adopted from Padilla et al. (2020))

3.2.2 Precision and recall

Precision is a performance metric that measures the accuracy of a model in predicting only the relevant objects by calculating the percentage of correct predictions out of all the bounding boxes predicted by the model. In other words, precision refers to how precisely the model can detect relevant items (Padilla, Netto and da Silva, 2020).

$$Precision = \frac{TP}{TP+FP} = \frac{TP}{\text{All observations (bounding box)}} \qquad (3.1)$$

Recall is the ratio of correctly detected positive samples to the total number of positive samples present in the ground truth data. It measures the model's ability to detect all relevant cases, meaning how many of the ground truth boxes are successfully detected by the model (Padilla, Netto and da Silva, 2020).

$$Recall = \frac{TP}{TP+FN} = \frac{TP}{\text{All Ground Truth}} \qquad (3.2)$$

3.2.3 Precision-Recall curve

In order to illustrate the trade-off between precision and recall, Precision and recall for each bounding box are calculated based on the corresponding confidence score (confidence threshold), and a precision-recall curve is plotted. A model is considered to have good performance when the precision stays high

while recall remains high. The model is said to perform well, even if the confidence threshold varies. The x-axis of the graph is recall, and the y-axis is precision, as shown in Figure 3.2. The Precision-Recall curve can be plotted using one of two approaches. The 11-point interpolation method uses 11 evenly spaced standard recall levels. The all-point interpolation approach utilises every recall point available (Koech, 2020).

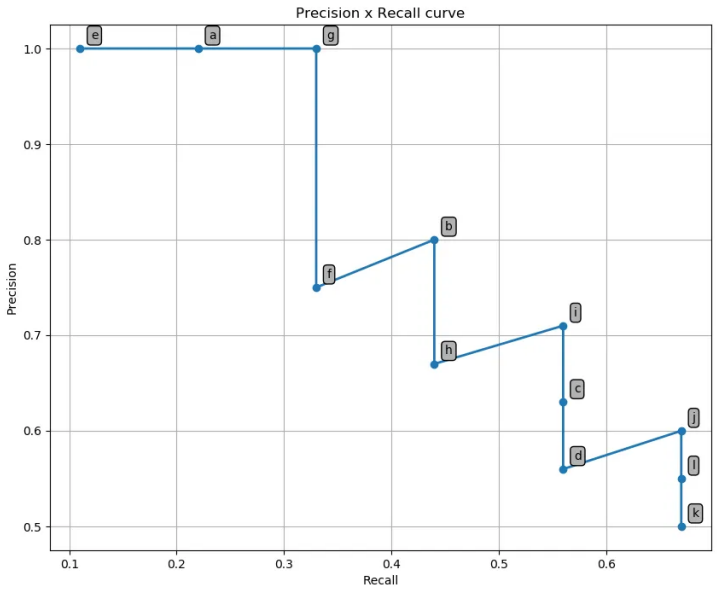


Figure 3.2: Precision x Recall curve (adopted from Koech (2020))

3.2.4 mAP

When evaluating the performance of an object detection model, it's important to consider both precision and recall. This is because precision only considers FP, not FN, and recall only considers FN, but not FP. For example, let's consider a scenario where there are five ground truth boxes in an image. Model number one predicts only one bounding box with a high confidence score and IoU, and the prediction is correct. In this case, the TP is one, FN is four, and FP is zero, resulting in a high recall. On the other hand, Model number two predicts ten bounding boxes, five of which predict the object correctly (with high IoU and confidence score), and the other five detect the background as an object. In this case, the TP will be five, FN will be zero, and FP will be five, resulting in a high recall. From these cases, it is evident that the performance of both models is not good, but they get high precision and recall, respectively. Therefore, it's crucial

to evaluate precision and recall together to better understand the model's overall performance.

To evaluate both precision and recall and represent it numerically, AP is calculated for each class by computing the area under the Precision-Recall curve. Then, mAP is calculated by averaging the AP values for all classes. This comprehensively assesses the model's performance or accuracy (Padilla, Netto and da Silva, 2020).

The mAP in PASCAL VOC and COCO datasets are not the same due to differences in their calculation methods. In PASCAL VOC, the mAP is calculated with an IoU threshold of 0.5, and the mAP is the average of APs across classes. In contrast, in COCO, the AP is calculated by considering IoU thresholds starting from 0.50 and incrementing by 0.05 until 0.95. The AP is then averaged over all IoU thresholds and categories, which is equivalent to the mAP in PASCAL VOC (Jonathan Hui, 2018).

3.3 Workflow of model training

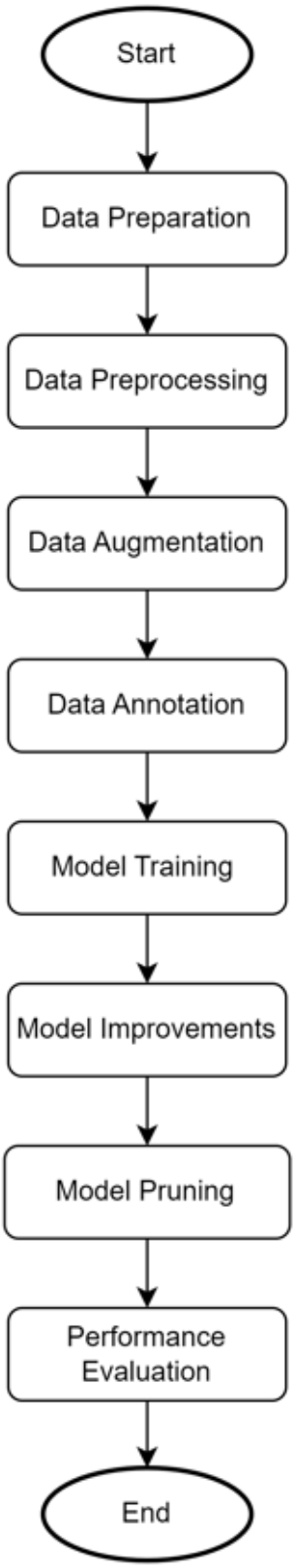


Figure 3.3: Workflow for training YOLOv7-tiny object detection model

The workflow for training the YOLOv7-tiny object detection model is shown in Figure 3.3. It encompasses various stages, including data preparation, data pre-processing, data augmentation, data annotation, model training, model improvements, model pruning and performance evaluation. The reason for choosing the YOLOv7 repository for training the object detection model is that YOLOv7 performs better than YOLOv5 in COCO dataset. While YOLOv8 has been introduced, it is still considered to be under development and lacking proper releases on GitHub. As a result, YOLOv7 stands out as the most stable and up-to-date YOLO model.

3.3.1 Data preparation

Table 3.2: Table of IC defect types and details

Defect type	Defect details
PCB defect	• Foreign molecules on PCB
Die defect	• Foreign molecules on Die • Epoxy Overflow • Die crack • Die chip • Die scratch • Die rotate • Misaligned Die (Die offset) • Missing Die
Wire and bonds defect	• Wire broken • Wire bond offset • Smash bond • Lifted on Pad • Lifted on Lead • Wire sweep • Missing wire • Bond tail defect • Double bond • Unbounded wire
LED defect	• Missing LED • Epoxy Overflow

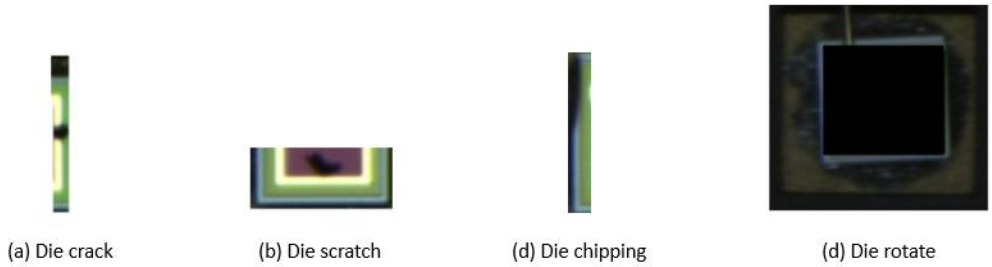


Figure 3.4: Die defects

The dataset used in this study has been kindly provided by ASPL Malaysia Sdn Bhd, which contains images of integrated circuits used in wireless earphones from a renowned manufacturer with various types of defects, including one type of PCB defect, eight types of die defects, ten types of wire and bond defects, and two types of LED defects, as outlined in Table 3.2. However, this study will only focus on two types of die defects, namely die crack and die rotate. Since the defect codes for die crack, die scratch and die chipping are the same, these three defects were categorized as die crack. Figure 3.4 shows the die defects involved in this study.

Each image in the datasets is available in three forms, which are images captured with a normal brightfield microscope, dark field with white light, and dark field with blue light. As a result, the images will appear similar to the images with green, red, and blue channels, respectively. The image used in this study is images captured by a normal brightfield microscope. Hence, OpenCV is used to filter the images based on the mean colour values of the images. Images are passed into the OpenCV with “cv2.imread()” function, and then “np.mean()” function is applied to the image to get the mean colour value. If the mean colour value for blue, green, and red channels exceeds 15, the image is considered captured by a normal brightfield microscope, else if the mean colour of the red channel is higher than blue channel, the image is considered captured by dark field with white light, else, the images is considered as captured by dark field with blue light.

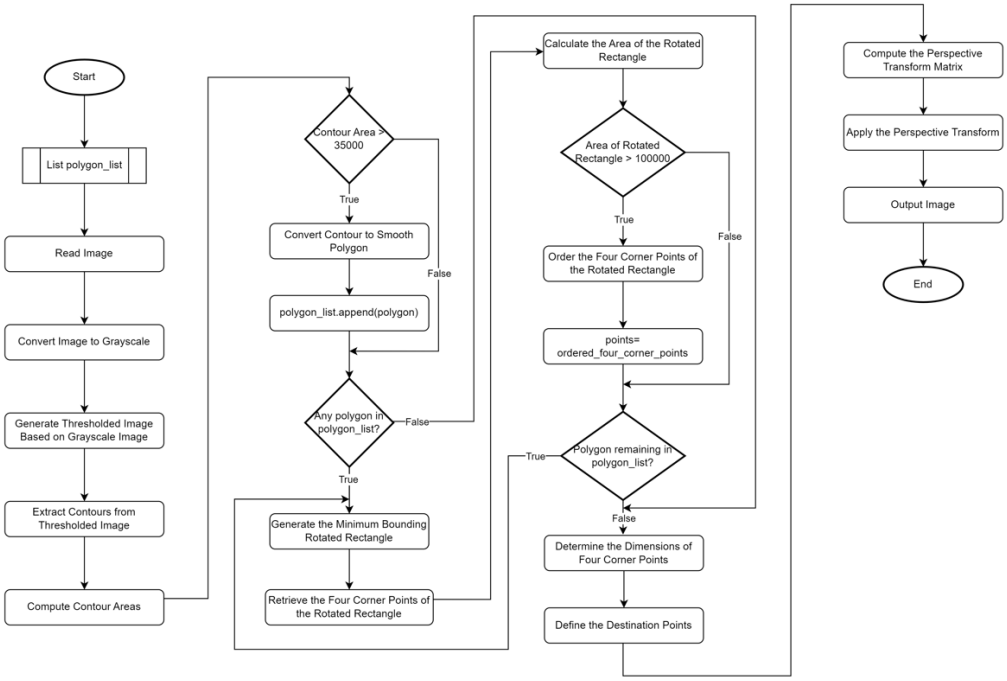


Figure 3.5: Flowchart for Capturing ROI of IC Chip Using OpenCV

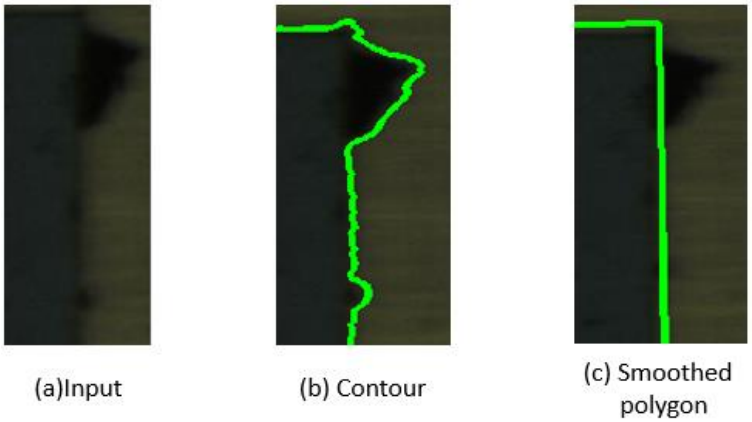


Figure 3.6: Effects of Smoothed Polygon

3.3.2 Data preprocessing

The provided image includes the IC and the background. To detect die rotation defects, the images passed into the YOLOv7-tiny must be upright, or else even though the die is not rotated but the whole IC chip itself is rotated, YOLOv7-tiny will detect the defect-less die as die rotation. Thus, capturing the ROI of the IC chip and making the image upright is needed.

3.3.2.1 Capture ROI with OpenCV

Figure 3.5 illustrates the step-by-step process for capturing the ROI of an IC chip using OpenCV. The workflow begins with the image being loaded into OpenCV via the “cv2.imread()” function. Subsequently, the image is converted into grayscale using “cv2.cvtColor()”. The subsequent step is focused on enhancing relevant features within the images and addressing issues related to inconsistent illumination. To tackle this, an adaptive thresholding technique is employed using the “cv2.adaptiveThreshold()” function. The chosen strategy involves calculating the threshold based on the arithmetic mean of the local pixel neighbourhood, with a neighbourhood pixel size of 151 and a constant “c” value of 3. The contours of this thresholded image are extracted with “cv2.findContours()”. If the area enclosed by a contour exceeds 35,000 units, it is considered a candidate. To ensure accuracy, contours are approximated into smooth polygons using “cv2.approxPolyDP()”, with an epsilon value set at 1% of the contour path length. It's important to note that some scratches may extend beyond the IC's borders, as depicted in Figure 3.6. These anomalies are included in the contour, and the smoothing process is critical to eliminate such artifacts. Without this step, the ROI cannot be captured successfully.

Following the smoothing process, the resulting polygon is added to the polygon_list. For each polygon within this list, a rectangle is generated using “cv2.minAreaRect()”. The four corner points of this rectangle are extracted using “cv2.cv.BoxPoints()”, and the rectangle's area is calculated based on these corners points. If the area of the rectangle exceeds 100,000 units, the corner point will be rearranged into a specific order of top-left, top-right, bottom-right, and bottom-left. The coordinates of the ordered four corner points will be assigned to a variable named “points”. Subsequently, the width and height between the four corner points saved in the “points” variable are computed. These dimensions are pivotal in determining the destination points for the perspective transform matrix. The perspective transform matrix is then calculated using both the four corner points and the destination points, facilitated by the “cv2.getPerspectiveTransform()” function. With this matrix in place, a perspective transformation is applied to the original image through “cv2.warpPerspective()”. Few assumption was made within this process. Firstly,

there are always polygons exceeding 35,000 units, and there are always a rectangle with an area exceeding 100,000 units, else the script will be terminated or an error will be thrown when calculating the width and height of corner points, as no valid corner points would exist. Lastly, it is assumed that there is only one rectangle with an area exceeding 100,000 units. If multiple such rectangles exist, only the last one encountered will be considered for perspective transformation, as the four ordered corner points saved in the variable “points” will be overwritten.

3.3.2.2 Capture ROI with YOLOv5

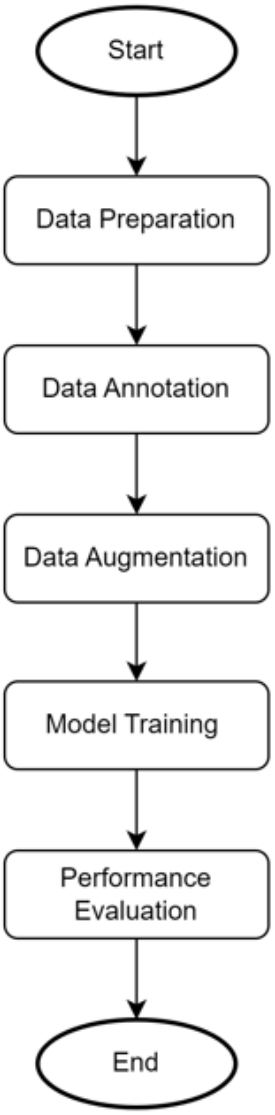


Figure 3.7: Workflow for training YOLOv5 segmentation model

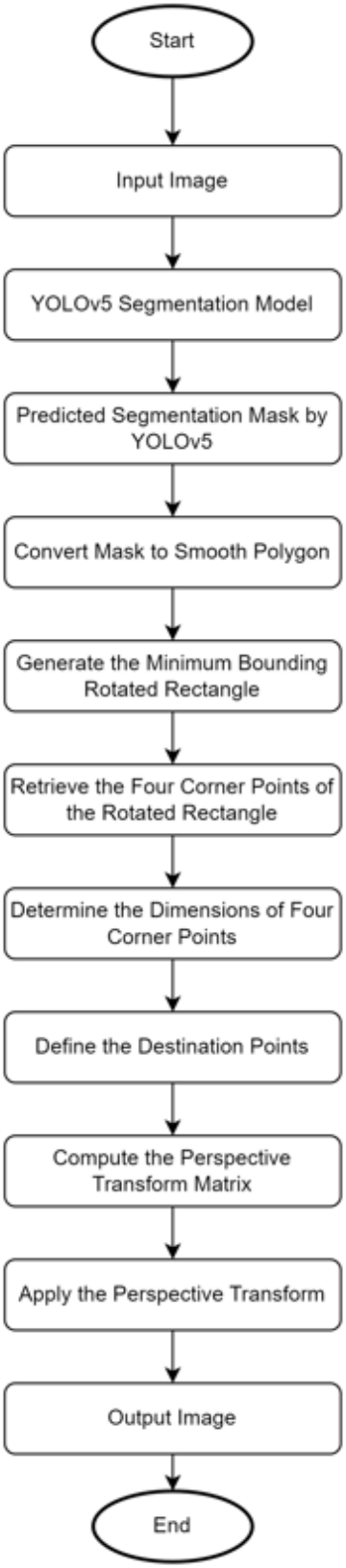


Figure 3.8: Flowchart for Inference Process of YOLOv5 Segmentation Model in Capturing the ROI of IC Chip

The workflow for training YOLOv5 segmentation model is shown in Figure 3.7. It encompasses data preparation, data annotation, data augmentation, model training and performance evaluation. The reason for selecting YOLOv5 as the segmentation model stems from the fact that while YOLOv7 also supports segmentation, the associated repositories, namely “u7” and “mask”, have not been properly maintained. These repositories lack essential maintenance and many convenience functions that could aid in the development of custom code. One such example is “masks2segments”, a function provided in YOLOv5, which allows for the conversion of masks to polygons. The absence of such functionalities in YOLOv7 necessitates the creation of custom code, making YOLOv5 a more practical choice for segmentation tasks as extra custom code was implemented for segmentation in this study. Figure 3.8 shows the inference process of YOLOv5 segmentation in capturing the ROI of the IC chip.

3.3.2.2.1 Data preparation

The dataset consists of 802 images, including 200 images from die crack, die rotation, and defect-less IC categories, respectively, as well as an additional 202 images from the category of missing die.

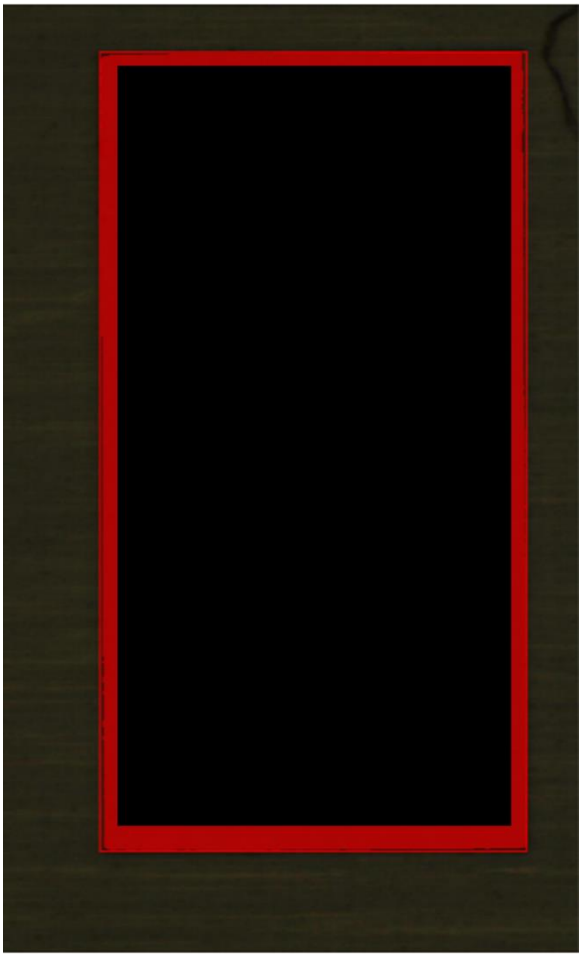


Figure 3.9: Masks and Bounding Box of IC Chip Obtained from SAM

[{"id":3786,"annotations":[{"id":2954,"completed_by":1,"result":[{"original_width":785,"original_height":1295,
"image_rotation":0,"value":{"x":16.81528662420382,"y":5.01930501930502,"width":74.01273885350318,
"height":84.16988416988417,"rotation":0,"rectanglelabels":["IC"]},"id":"I","from_name":"RectangleLabels",
"to_name":"image","type":"rectanglelabels","origin":"manual"},{"original_width":785,"original_height":1295,
"image_rotation":0,"value":{"format":"rle","rle":[0,62,11,252,57,27,255,255,255,0,255,255,224,31,255,252,3,

Figure 3.10: Exported Dataset in JSON Format

Figure 3.11: Exported Dataset in COCO Format

```
0.0 0.10828877005347594 0.06807511737089202 0.10026737967914438
0.9358372456964006 0.8783422459893048 0.9405320813771518 0.8917112299465241
0.07120500782472614
```

Figure 3.12: Exported Datasets in YOLO Text File Format

3.3.2.2.2 Data annotation

These 802 images were annotated using Label Studio. To ensure high-quality annotations, instead of manual labelling, a SAM model was integrated into Label Studio to assist in the labelling process. An initial annotation box encompassing the entire IC chip was manually drawn. Subsequently, SAM accurately predicted and returned both the bounding box and mask for the annotated regions, streamlining the annotation process and ensuring precise results. The Label Studio integrated with SAM was obtained from the Open-MMLab Playground. Figure 3.9 shows the masks and bounding box returned by SAM.

Since the masks returned by SAM are in uncompressed RLE format, Label Studio supports this format only when the dataset is exported to a JSON file, as illustrated in Figure 3.10. Subsequently, the JSON file is converted into COCO format using a script provided by Open-MMLab Playground, as shown in Figure 3.11. Afterwards, a custom script is employed to convert the COCO format into YOLO format. This script essentially retrieves the uncompressed RLE masks from the COCO file and converts them into polygon format, which is then written into a ".txt" file, as shown in Figure 3.12. The dataset was initially divided into a training set and a test set, with a 9:1 ratio, respectively. Subsequently, a validation set was created by splitting 10% of the training set. This partitioning resulted in 648 images in the training set, 81 images in the test set, and 73 images in the validation set.

3.3.2.2.3 Data Augmentation

To enhance the robustness of the dataset, data augmentation was applied using the Albumentations library. The employed data augmentation techniques included "VerticalFlip", "HorizontalFlip", "Rotate", "CLAHE", "AdvancedBlur", "MultiplicativeNoise", "ElasticTransform", "GridDistortion" and "OpticalDistortion".

For the training set, each image was augmented to generate an additional three augmented images, effectively expanding the dataset. To maintain a balanced distribution and assess the model's robustness, augmentation was also performed on both the testing and validation sets, resulting in an extra augmented image generated for each image in these sets. The augmented dataset comprises a total of 2,592 images in the training set, 162 images in the testing set, and 146 images in the validation set.

```
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.01  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.5  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 1.0  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
# anchors: 3  # anchors per output layer (0 to ignore)
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 90  # image rotation (+/- deg)
translate: 0.1  # image translation (+/- fraction)
scale: 0.5  # image scale (+/- gain)
shear: 0  # image shear (+/- deg)
perspective: 0.0 # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.0  # image flip left-right (probability)
mosaic: 0.0  # image mosaic (probability)
mixup: 0.0  # image mixup (probability)
copy_paste: 0.0  # segment copy-paste (probability)
```

Figure 3.13: Hyperparameters Configuration for YOLOv5 Segmentation Model

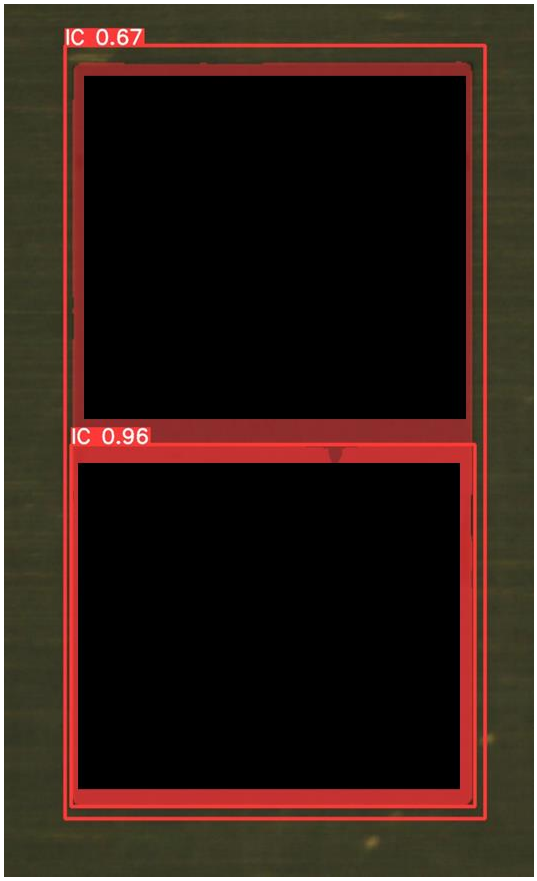


Figure 3.14: Inference Result with Mosaic Enabled

3.3.2.2.4 Model training

YOLOv5n-seg and YOLOv5s-seg were trained with the hyperparameters shown in Figure 3.10. Notably, mosaic augmentation was disabled during training. This decision was made to prevent the mosaic augmentation from splitting the IC chip into multiple parts, which could lead to confusion for the model. When the mosaic is enabled, the model might consider part of the IC chip as a complete IC chip, resulting in two predicted masks for a single IC chip. One mask would represent the entire IC chip, while another would only cover half of it, as illustrated in Figure 3.11. The training process consisted of 50 epochs, with a batch size of 16. The input image size was set to 640x640 pixels and pre-trained weight was used to speed up the convergence of the model.

3.3.2.2.5 Performance Evaluation

The evaluation metrics used for assessment included mAP@0.5:0.95(BOX) and mAP@0.5:0.95(Mask). These metrics were employed to evaluate how well the mask generated by the model aligned with the ground truth mask. In addition to these metrics, the FPS was also evaluated to assess the model's inference speed. Parameters and GLOPS were also evaluated to assess the computational cost of the model.

For FPS evaluation, inference time outputted by the official script exhibited some fluctuations. These fluctuations were attributed to the GPU's behaviour, particularly during the initial phase of inference when the GPU was not warmed up and was running at low power. To obtain more accurate FPS measurements, a custom script was developed.

In this custom script, based on a defined number, a batch of images was generated with random pixel values using “torch.randn()”, each sized at 640x640 pixels. The model underwent a warm-up process of 1000 iterations using these randomized images. After the warm-up phase, the model performed 1000 inferences with these randomized images, and the FPS was calculated based on the start time and end time of model inference. To ensure precise FPS measurements, “torch.cuda.synchronize()” was employed. The FPS were evaluated for both single batch and 16 batches of randomized images. It's important to note this script does not include NMS in the calculation of FPS. All

the trained YOLO model's FPS in this study were assessed using this script, with the exception of the ensemble model consisting of YOLOv5n and SAM.

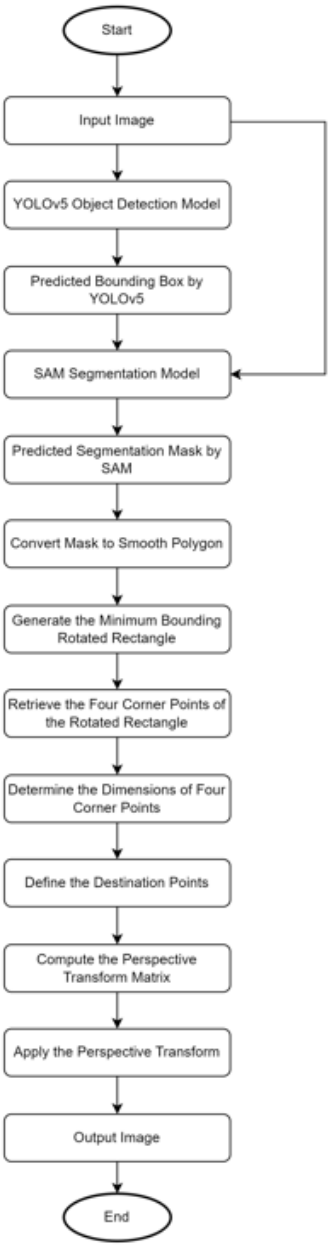


Figure 3.15: Flowchart for Inference Process of Ensemble Model in Capturing the ROI of IC Chip

3.3.2.3 Ensemble model of YOLOv5n object detection model and SAM

To further enhance the accuracy of segmentation, complex networks such as transformers are often considered due to their potential for superior performance compared to YOLO models. However, it's important to note that these networks typically come with slower inference times, and training them can be time-consuming and computationally expensive. Therefore, the choice of adopting a zero-shot segmentation model becomes a prudent one.

The performance of the SAM in returning accurate masks during data annotation has been observed, as detailed in Section 3.3.2.2.2. SAM has been pre-trained on a substantial dataset comprising 1.1 billion high-quality segmentation masks and SAM was built based on the vision transformer, making it an excellent candidate for the segmentation model. To achieve even more accurate segmentation results, bounding boxes need to be passed into the SAM model to act as a prompt.

To facilitate this, a YOLOv5n object detection model was trained. The dataset used for training YOLOv5n is the same as described in Section 3.3.2.2.3, with the only difference being that bounding boxes are retrieved from the COCO format instead of the uncompressed RLE mask format. The hyperparameters and model configuration remain consistent with Section 3.3.2.2.4. FastSAM-s, FastSAM-x, MobileSAM, SAM_VIT_b, and SAM_VIT_l were evaluated. Figure 3.15 shows the ensemble model's inference process in capturing the IC chip's ROI. The “detect.py” script was customized to fit the inference process.

Instead of evaluating the model's performance using mAP, a manual inspection of the predicted mask output by SAM was conducted. This approach was taken because the official script did not support measuring mAP for SAM, and no self-written code had been implemented to measure the mAP of SAM. FPS was also evaluated.

As the operational principles of YOLO and SAM differ, the FPS measurements were directly obtained using the “detect.py” script with self-written code, as the original “detect.py” does not support measuring the FPS of SAM. FPS was evaluated by measuring the time taken for YOLO inference and SAM inference separately, based on each model's inference process's start time and end time. It's important to note that this FPS measurement specifically

accounts for the inference time of YOLOv5n and SAM, without considering other post-processing or pre-processing steps such as NMS. The FPS measurement was conducted for a single batch consisting of one IC image, to align with the default behaviour of “detect.py”, which accepts one image at a time. A total of 1000 IC images were used to test the FPS, resulting in a total of 1000 iterations.

3.3.3 Data augmentation

After preprocessing the images using the ensemble model as mentioned in Section 3.3.2.3, a total of 170 images with die rotation defects and 386 images with die crack defects were obtained. However, this quantity was insufficient to train a YOLOv7-tiny model with a high mAP, and the class distribution was imbalanced.

To address the issue of limited defect images, StyleGANv2 and StyleGANv3 were trained to generate additional images. Additionally, Stable Diffusion was fine-tuned with LoRA (Low-Rank Adaptation) as a technique to fine-tune the stable diffusion by adjusting the attention mechanisms (q, k, and v). The images sent to the model depict the die itself, and they were manually cropped from the full IC chip. Since the input image dataset was limited, Adaptive Discriminator Augmentation (ADA) was enabled in StyleGANv2 and StyleGANv3 for better performance and to prevent overfitting.

To assess the performance of the StyleGAN models, an Inceptionv3 network was automatically trained to evaluate the Fréchet Inception Distance (FID). FID measures the dissimilarity between the distribution of generated images and a set of real images based on real covariance, real mean, fake covariance and fake mean statistics. These statistics are derived from feature vectors extracted by Inceptionv3 from the real image and generated image. These three models were trained for 50,000 iterations. The performance of StyleGAN models was evaluated using the FID metric, where 1000 fake images were generated to measure FID. As the stable diffusion provided by MMagic does not support the FID metric, manual inspection was performed on the output images of stable diffusion.

After manually filtering the images output by StyleGANv2, 330 die rotation images and 114 die crack images were selected. Random IC images were selected for each of these images, and the original die region in the IC chip was replaced with generated images manually. 500 defect-free IC images were added as negative samples, bringing the total to 1500 images. Horizontal flip augmentation was applied to these images, resulting in a dataset of 3,000 images.

Figure 3.16: Masks and Bounding Boxes of Die Rotation Obtained from SAM

3.3.4 Data annotation

These 3000 images were then annotated using Label Studio. The die rotation defects were labelled with the assistance of SAM, as shown in Figure 3.16, while die crack defects were labelled manually as SAM did not perform well on small objects. Defect-free IC images are not annotated. The dataset was subsequently divided into training, testing, and validation sets in approximately an 8:1:1 ratio, as same as previously mentioned in Section 3.3.2.2.2. This resulted in a training set with 2430 images, a test set with 300 images, and a validation set with 270 images.

Annotated objects in the dataset were categorized into three size groups based on the COCO definitions. Small objects were defined as those with bounding boxes smaller than 32x32 pixels, medium objects had bounding boxes ranging from 32x32 to 96x96 pixels, and large objects had bounding boxes larger than 96x96 pixels. A script was employed to calculate the sizes of these objects, resulting in a distribution of 1028 small objects, 142 medium-sized objects, and 1058 large objects within the dataset.

```
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.01  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.5  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 1.0  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
# anchors: 3  # anchors per output layer (0 to ignore)
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 0.0  # image rotation (+/- deg)
translate: 0.1  # image translation (+/- fraction)
scale: 0.5  # image scale (+/- gain)
shear: 0.0  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.0  # image flip left-right (probability)
mosaic: 0.0  # image mosaic (probability)
mixup: 0.05  # image mixup (probability)
copy_paste: 0.0  # image copy paste (probability)
paste_in: 0.05  # image copy paste (probability), use 0 for faster training
loss_ota: 1 # use ComputeLossOTA, use 0 for faster training
```

Figure 3.17: Hyperparameters Configuration for YOLOv7-tiny Object Detection Model

3.3.5 Model Training

The original YOLOv7 source code only contained three constant seed settings, namely “random.seed()”, “np.random.seed()” and “torch.manual_seed()” This resulted in training outcomes that varied each time. To ensure the reliability, accuracy, and reproducibility of the results, custom code was developed. This custom code sets all the seeds in the “numpy”, “torch”, and “random” libraries to a consistent value of one using “random.seed()”, “np.random.seed()”, “torch.manual_seed()”, “torch.cuda.manual_seed()”, and “torch.cuda.manual_seed_all()”. Additionally, to maintain consistent output for CUDA operations, “torch.backends.cudnn.deterministic” was set to True. From previous experience, if the seed is not completely constant, the mAP@0.5 may fluctuate around -2% to +2%, even in the same hyperparameter and model configuration.

YOLOv7-tiny were trained with the hyperparameters shown in Figure 3.17. Notably, mosaic augmentation was also disabled during training. This decision was made because some crack defects also appeared in the LED part

of the IC, which was not the focus of this study. To ensure that the model ignores these types of defects, full IC images must passed to the model. The training process consisted of 300 epochs, with a batch size of 16, the input image size was set to 640x640 pixels and no pre-trained weights were used. This decision was made to ensure the fairness of the network. As improvements were made to the model, some components in the network were changed and became different from the original network. For these changed or extra components, pre-trained weights were not loaded. Pre-trained weights were only loaded for layers that matched the original network, resulting in an unfair comparison. Instead of using predefined anchor boxes intended for the COCO dataset, which are big and not suitable for the datasets used in this study, the "AutoAnchor" option is enabled. This option generates anchor boxes through k-means clustering based on the ground truth boxes present in the dataset. A YOLOv7-tiny model was trained.

3.3.6 Model Improvements

3.3.6.1 Normalized Wasserstein Distance Loss Function

The authors claim that metric based on IoU is very sensitive to positional deviations of small targets, especially for pixelated targets, and that slight positional deviations can lead to significant IoU degradation. This sensitivity arises due to the discrete nature of bounding box positions, making it challenging for the network to converge effectively during training. To solve this problem, the authors proposed an approach to measure the similarity of the bounding box via wasserstein distance instead of the standard IoU. Generally, both the predicted and ground truth bounding boxes are modelled as 2D gaussian distributions and then the similarity of the derived 2D gaussian distributions is measured using NWD (Wang et al., 2021). The NWD between the 2D gaussian distributions of the ground truth box and the predicted box can be expressed by the following formula:

$$NWD(\mathcal{N}_a, \mathcal{N}_b) = \exp\left(-\frac{\sqrt{W_2^2(\mathcal{N}_a, \mathcal{N}_b)}}{C}\right) \qquad (3.3)$$

Where $W_2^2(\mathcal{N}_a, \mathcal{N}_b)$ is squared Euclidean distance between the predicted box and bounding box that is represented in 2D gaussian distribution, can be expressed as:

$$W_2^2(\mathcal{N}_a, \mathcal{N}_b) = \left\| \left(\left[cx_a, cy_a, \frac{w_a}{2}, \frac{h_a}{2} \right]^{\mathrm{T}}, \left[cx_b, cy_b, \frac{w_b}{2} \cdot \frac{h_b}{2} \right]^{\mathrm{T}} \right) \right\|_2^2 (3.4)$$

$cx, cy, \frac{w}{2}, \frac{h}{2}$ is defined as the coordinates and dimensions of the box.

3.3.6.2 CoordConv

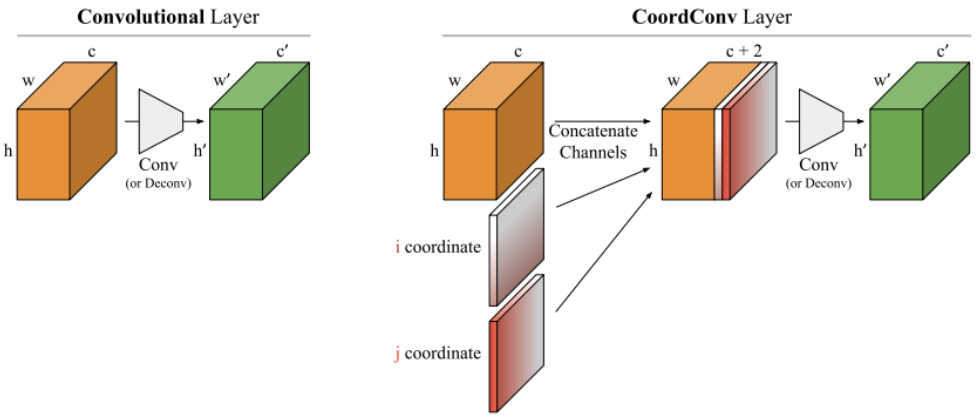


Figure 3.18: Convolutional Layer vs. CoordConv Layer: A Comparison (adopted from Liu et al. (2018))

The authors conducted experiments in which they applied traditional convolution to convert cartesian coordinates (i, j) into one-hot pixel space and vice versa. Through this experiment, they found out that traditional convolution lacked awareness of the positional information associated with each filter, which means traditional convolution captures local information but does not inherently consider the positional information of features within the image. To solve this problem, the authors propose CoordConv, which adds two additional coordinate channels that represent the original input's i and j coordinates to the original input feature map, as shown in Figure 3.18, so that CoordConv can capture spatial information of the feature map. In simple terms, these coordinate channels represent coordinates of feature map pixels, allowing the convolutional

learning process to have a certain level of spatial awareness regarding the coordinates and thus improving accuracy. Since object detection looks at pixel space and output bounding boxes in cartesian space, the author claims that CoordConv can help in the field of object detection (Liu et al., 2018).

3.3.6.3 Slim-neck by GSConv

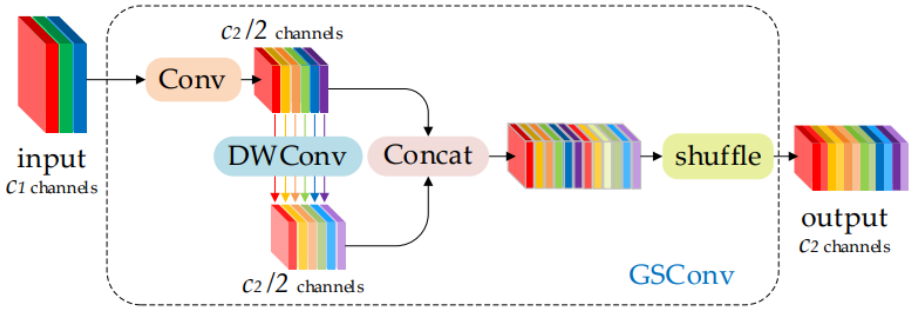


Figure 3.19: Structure of GSConv (adopted from H. Li et al. (2022))

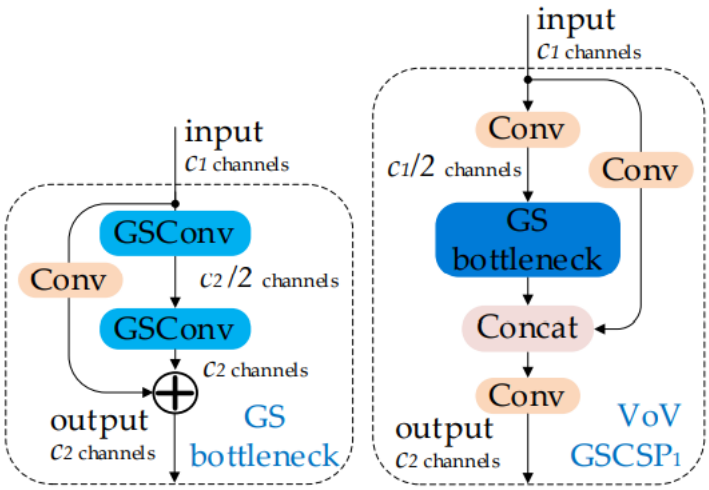


Figure 3.20: Structure of GS Bottleneck Module and VoV-GSCSP Module (adopted from H. Li et al. (2022))

Most of the lightweight modules are built by depthwise separable convolution (DSC). However, during the convolution process of DSC, the channel information of input images is segregated, resulting in lower feature extraction ability compared to standard convolution (SC) and thus leading to lower accuracy. To address this problem, the authors proposed a GSConv module that combines SC, DSC, and shuffle operations, as shown in Figure 3.19. The input

first undergoes a standard convolution, followed by depthwise convolution (DWConv). The results of these two convolutions are concatenated, and a shuffle operation is performed to exchange local feature information between these two output feature maps. Based on GSConv, the authors introduced the GS bottleneck module. Based on GS bottleneck module, the authors implement a one-shot aggregation method and design a VoV-GSCSP module, as shown in Figure 3.20. The authors replaced the original neck in scaled-yolov4 with a neck based on VoV-GSCSP and GSConv. The authors observed that this replacement successfully detected more small objects in the DOTA1.0 dataset while reducing the parameter count. The Slim-neck is formed by combining VoV-GSCSP and GSConv (Li et al., 2022b).

3.3.6.4 Model Configuration for Improved Network

```
nwd = wasserstein_loss(pbox, selected_tbox)
iou_ratio = 0.5
lbox += (1 - iou_ratio) * (1.0 - nwd).mean() + iou_ratio *  (1.0 - iou).mean()   # iou loss
# Objectness
iou = (iou.detach() * iou_ratio + nwd.detach() * (1 - iou_ratio)).clamp(0, 1).type(tobj.dtype) #nwd
tobj[b, a, gj, gi] = (1.0 - self.gr) + self.gr * iou #nwd
```

Figure 3.21: Combined Loss Function Code Incorporating CIOU and NWD

As mentioned in Section 3.3.4, the dataset consists of approximately an equal proportion of small and large targets. Instead of directly replacing the original CIOU loss function in YOLOv7-tiny with the NWD loss function, a combination of both loss functions is employed, where each contributes 50% to the total loss, resulting in a balanced approach, as shown in Figure 3.21.

Below outlines the modifications applied to the YOLOv7-tiny neck, including CoordConv, as well as the changes made to the neck through VoV-GSCSP and GSConvs. Specifically, only the neck architecture is modified, while the backbone remains unchanged. In the neck with CoordConv, layers 38, 40, 48, 50, 74, 75, and 76 are replaced by CoordConv layers. In the neck with VoV-GSCSP and GSConvs, the original e-elan module is substituted with VoV-GSCSP, and layers 38, 40, 43, 45, 48, and 51 are replaced with GSConvs. GSConvs are used instead of GSConv because pruning will be performed later, and the shuffle operation in GSConv does not support pruning. GSConvs simply

replaces the shuffle operation with a normal convolutional layer. The layers in VoV-GSCSP still remain as GSConv. The activation function used in CoordConv and Slim-neck is the same as the original convolutional layer in YOLOv7-tiny, which is LeakyReLU with a value of 0.1. All the hyperparameter settings and training configurations remain the same, as mentioned in Section 3.3.5.

3.3.7 Model Pruning

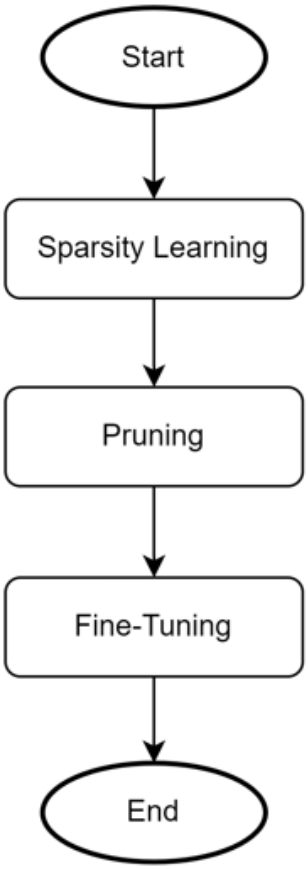


Figure 3.22: Workflow for Network Slimming

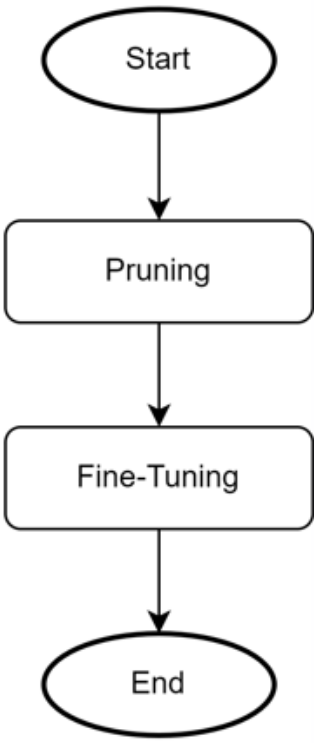


Figure 3.23: Workflow for LAMP Pruning

One of the most frequently used techniques in industry to lightweight or speed up models is model quantization. However, these kinds of techniques often rely on specific hardware. For example, OpenVINO requires running on Intel devices, and TensorRT requires running on Nvidia devices. In comparison, model pruning does not have hardware limitations, as the output of pruning is still a ".pt" weight file that can be easily converted to the ONNX format and deployed on any device. These pruned models can also be further accelerated through model quantization at a later stage.

3.3.7.1 Torch-Pruning

The pruning package used in this study is "Torch-Pruning". In the past, custom pruning algorithms were required for different networks because the parameters connecting to neurons varied in each network. To prune neurons, the parameters connected to that neuron also needed to be pruned. Torch-pruning uses a DepGraph algorithm to model the parameter dependencies in any network. The relationships between nodes in the network are determined recursively to identify their dependencies, which allows them to be grouped together, creating

a dependency graph. This graph helps determine which nodes and parameters need to be pruned together. In summary, torch-pruning is a network-agnostic pruning library that can be used for pruning in various neural network architectures (Fang et al., 2023).

3.3.7.2 Layer-Adaptive Magnitude-based Pruning (LAMP)

Each weight tensor is flattened into a one-dimensional vector, and these vectors are assumed to be arranged in ascending order, where |W[u]| $\leqslant$ |W[v]|. Here, u and v represent the indices in the weight vector, and the LAMP score for the u-th index in the vector can be calculated using the following formula:

$$\text{score}\ (u; W) := \frac{(W[u])^2}{\sum_{v \geq u}\ (W[v])^2} \quad (3.5)$$

The LAMP score measures the importance of all parameters or connections that are connected to a specific neuron or layer. Connections with the lowest LAMP scores will be globally pruned (Lee et al., 2020).

3.3.7.3 Network Slimming

In network slimming, the scaling factor γ from batch normalization is reused and introduced to the channel's output. L1 sparse regularization is applied to the factor γ during sparsity learning. The factor γ in batch normalization represent scaling factor that controlling the feature map's data and determining channel importance. L1 sparse regularization gradually drives γ in the channels towards zero during sparsity learning, identifying which channels are unimportant and subsequently pruning the unimportant channels (Liu et al., 2017).

3.3.7.4 Configuration for Pruning

Figures 3.22 and 3.23 depict the workflow for network slimming and LAMP pruning, with the main difference being that network slimming requires sparsity learning to determine unimportant channels, whereas LAMP pruning can directly calculate the LAMP score and prune the network. Notably, both

network slimming and LAMP pruning are implemented in the torch-pruning package as channel pruning techniques.

The pruning is applied to the modified YOLOv7-tiny with CoordConv neck and modified YOLOv7-tiny with slim-neck. The hyperparameter configuration and training settings for fine-tuning and sparsity learning remain the same as mentioned in section 3.3.5. For pruning, the speed_up rate is set to 1.5, which is calculated based on GLOPS. "max_ch_sparsity" is set to 1.0, indicating that unimportant channels will be completely removed. The "iterative_steps" parameter is set to 200. This ensures that the model iteratively prunes step by step, preventing excessive pruning that might exceed the specified "speed_up" thresholds. Regarding sparsity learning, "reg" is set to 5e-4, representing the coefficient of L1 sparse regularization. The GSConv layers in VoV-GSCSP module are excluded from pruning process The pruning script was developed based on the examples provided in the torch-pruning documentation.

3.3.8 Performance Evaluation

The metrics evaluated primarily focus on precision, recall, mAP@0.5, and mAP@0.5:0.95. Precision is used to assess false positives, while recall evaluates false negatives. mAP@0.5 represents the model's classification ability, determining whether it can identify die rotations or die cracks. mAP@0.5:0.95 indicates how accurately the predicted bounding boxes can locate defects. To evaluate the extent of model lightweight after pruning, parameters, GLOPS, and FPS for 16 batch sizes was assessed.

3.4 Work Breakdown Structure (WBS)

0.0 Deep Learning-Based Machine Vision for Defect Detection

1.0 Preliminary Planning

- 1.1 Understanding project background
- 1.2 Define problem statement
- 1.3 Define project objective

1.4 Define research question

1.5 Define project scope and limitations of study

1.6 Define proposed solution

2.0 Project Planning

2.1 Literature Review

2.1.1 Review deep learning

2.1.2 Review semiconductor defect detection systems that use image processing techniques

2.1.3 Review semiconductor defect detection systems that use classification techniques

2.1.4 Review semiconductor defect detection systems that use object detection techniques

2.1.5 Study on Object detection model

2.2 Define Methodology and Workplan

2.2.1 Proposed methodology

2.2.2 Develop WBS

2.2.3 Develop Gantt Chart

2.2.4 Define development tools

3.0 Model Developing

3.1 Preparation phase

3.1.1 Data pre-processing

3.1.2 Data augmentation

3.1.3 Data annotation

3.2 Modelling

3.2.1 Train YOLOv7-tiny object detection model

3.3 Model Improvements

3.3.1 Changing the model’s components

3.3.2 Retrain model

3.4 Performance Evaluation

3.4.1 Evaluate the model's mAP

3.4.2 Evaluate the loss graph and mAP graph

3.4.3 Evaluate the model's FPS

3.5 Model pruning

3.5.1 Prune model with network slimming

3.5.2 Prune model with LAMP pruning

3.5 Gantt Chart

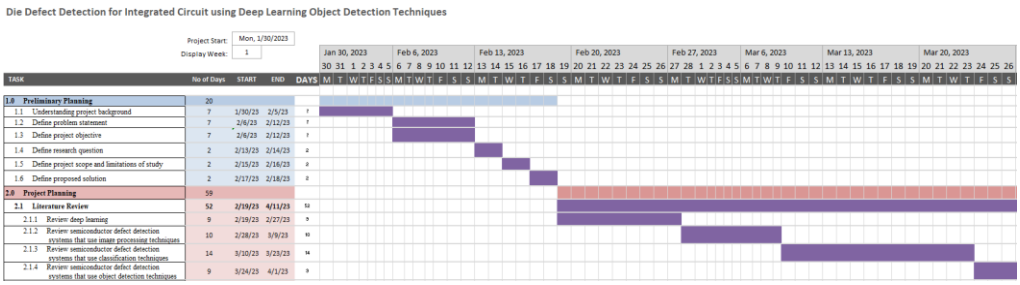


Figure 3.24: Gantt chart for Preliminary Planning and Project Planning from 30/1/2023 to 26/3/2023

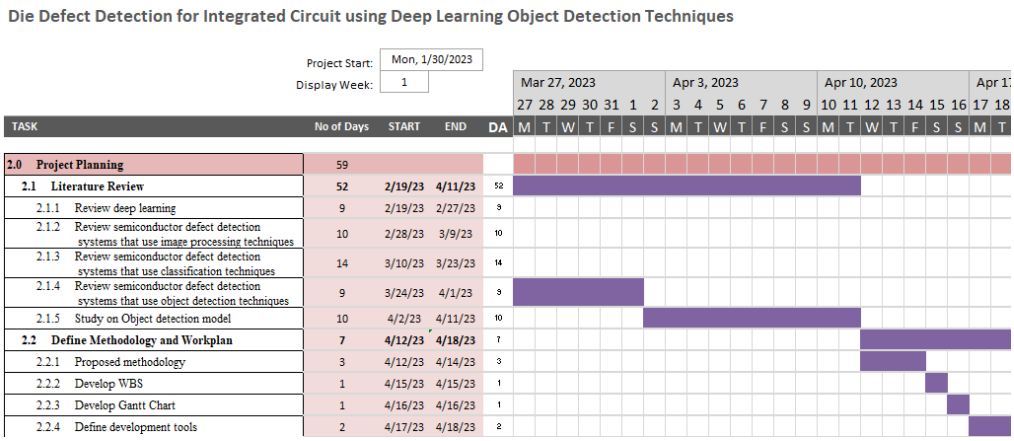


Figure 3.25: Gantt chart for Project Planning from 27/3/2023 to 18/4/2023

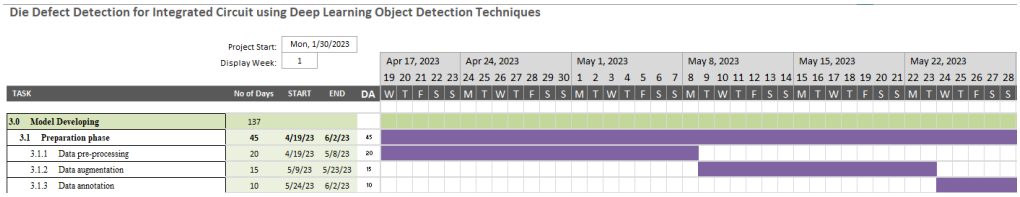


Figure 3.26: Gantt chart for Model Developing from 19/4/2023 to 28/5/2023

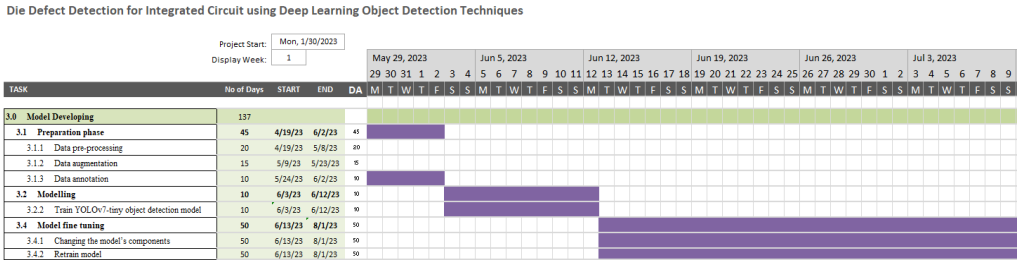


Figure 3.27: Gantt chart for Model Developing from 29/5/2023 to 9/7/2023

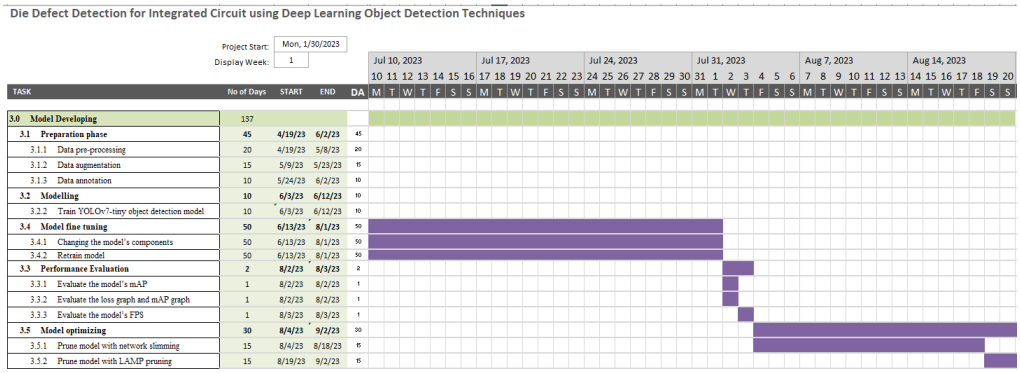


Figure 3.28: Gantt chart for Model Developing from 10/7/2023 to 20/8/2023

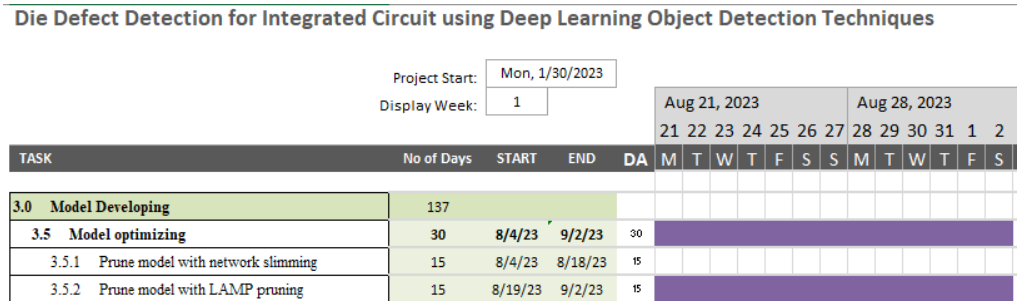


Figure 3.29: Gantt chart for Model Deployment from 21/8/2023 to 2/9/2023

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Comparison of OpenCV, YOLOv5, and SAM in Capturing ROI

4.1.1 OpenCV

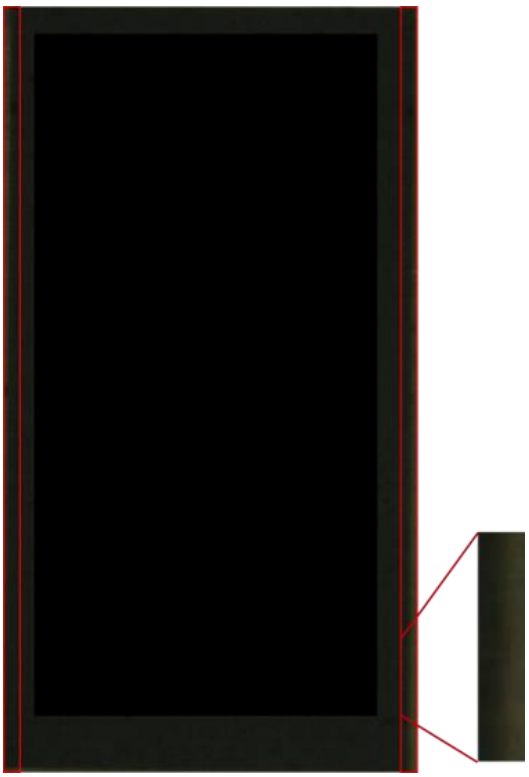


Figure 4.1: Results of OpenCV in Capturing ROI

In Figure 4.1, the output image generated by OpenCV demonstrates the challenges in accurately capturing ROI for the IC chip. OpenCV's performance in this task is imperfect, as it inadvertently includes a portion of the background (depicted in green), as illustrated by the red rectangular box in Figure 4.1. This issue arises from the fact that the IC's border lacks a distinct separation between the IC itself (appearing as a black colour) and the surrounding background (appearing as a green colour). Instead, there exists a gradual transition from black to green, making it difficult for OpenCV to precisely delineate this feature.

4.1.2 YOLOv5 Segmentation Model

Table 4.1: Metrics of YOLOv5 Segmentation Model

Model	mAP@0.5(BOX)	mAP@0.5:0.95 (BOX)	mAP@0.5(MASK)	mAP@0.5:0.95(MASK)
YOLO v5n-seg	99.5	99.5	99.5	99.5
YOLO v5s-seg	99.5	99.5	99.5	99.5

Table 4.2: Computational Costs of YOLOv5 Segmentation Model

Model	FPS(b1)	FPS(b16)	Params	GFLOPS
YOLOv5n-seg	114.5	17.7	1879750	6.7
YOLOv5s-seg	89.5	8.2	7398422	25.7

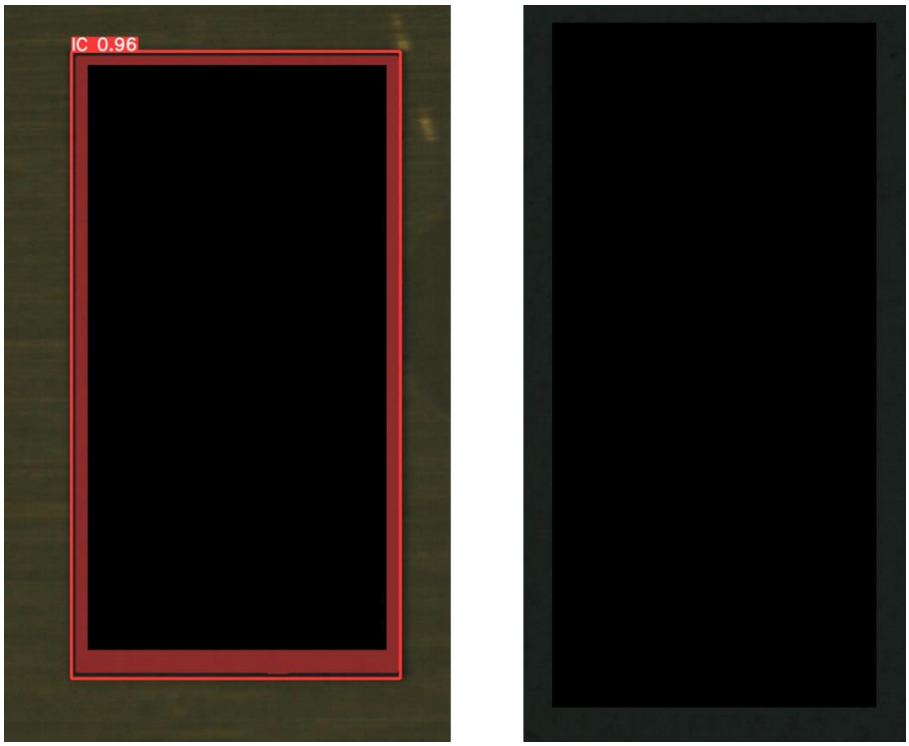


Figure 4.2: Results of YOLOv5n-seg in Capturing ROI

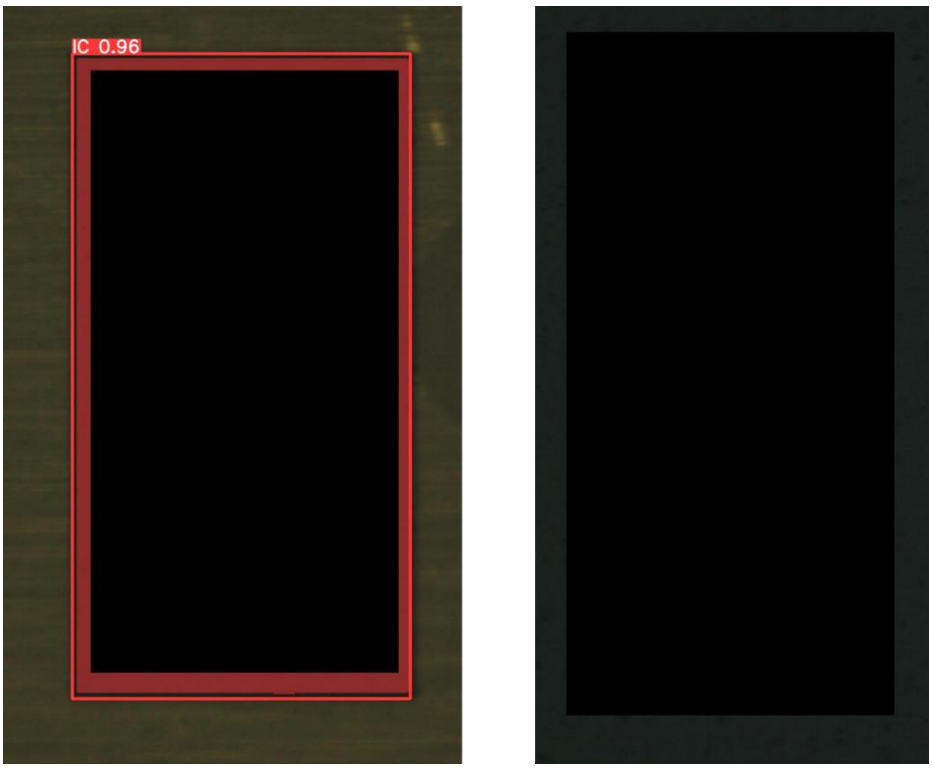
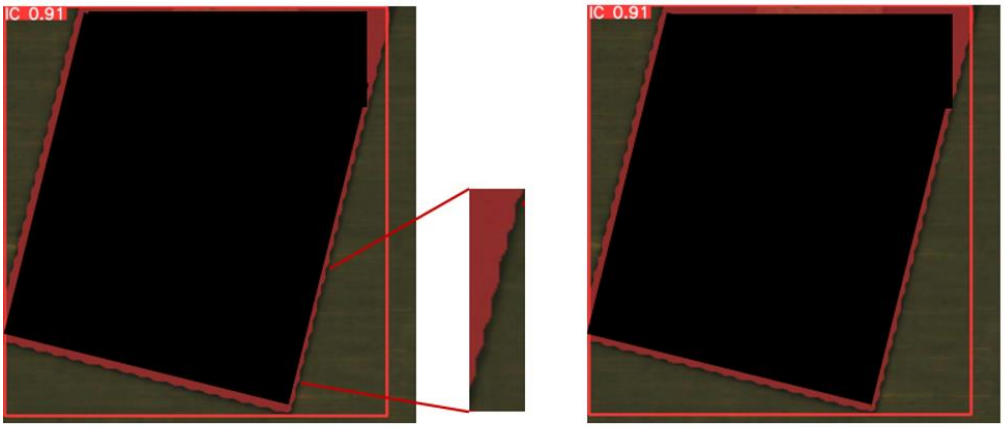


Figure 4.3: Results of YOLOv5s-seg in Capturing ROI



(a) YOLOv5n-seg (b) YOLOv5s-seg

Figure 4.4: Predicted Masks by YOLOv5n-seg and YOLOv5s-seg for Rotated IC Chip

While both YOLOv5n-seg and YOLOv5s-seg demonstrate impressive mAP@0.5:0.95 scores of 99.5 for both bounding boxes and masks, as shown in Table 4.1, there are notable differences in computational efficiency. YOLOv5s-seg incurs significantly higher computational costs and operates at a lower FPS rate compared to YOLOv5n-seg, as shown in Table 4.2.

However, it's important to acknowledge that both models have their limitations. They excel in accurately capturing and cropping ROI when the IC is in its default orientation, as exemplified in Figures 4.2 and 4.3. Yet, when the IC is rotated, both models exhibit a distinctive zig-zag pattern at the mask border, as illustrated in Figure 4.4.

4.1.3 SAM

Table 4.3: Metrics and Computational cost of YOLOv5n Object Detection Model

Model	mAP@0.5	mAP@0.5:0.95	FPS(b1)	FPS(b16)	Params	GFLOPS
YOLOv5n	99.5	99.1	120.26	35.8	1760518	4.1

Table 4.4: FPS Performance of Various SAM Models

Model	FPS(b1)
YOLOv5n + FastSAM-s	34
YOLOv5n + FastSAM-x	17.1
YOLOv5n + MobileSAM	29.2
YOLOv5n + SAM_VIT_b	3.3
YOLOv5n + SAM_VIT_l	2.2

Figure 4.5: Predicted Masks by Various SAM Models

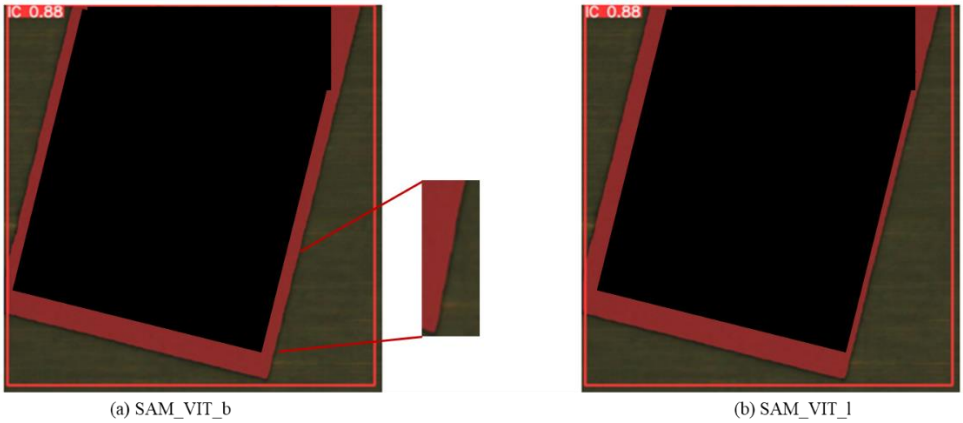


Figure 4.6: Predicted Masks by SAM_VIT_b and SAM_VIT_l for Rotated IC Chip

To tackle this problem, the SAM model was utilized for segmentation. Prior to that, a YOLOv5n object detection model was trained to roughly locate the IC's position and send the bounding box coordinates to SAM as a prompt, aiming to

improve segmentation success rates. Table 4.3 displays the metrics and computational costs of the YOLOv5n object detection model.

In Figure 4.5, various outputs from different SAM models are shown. Masks generated by FastSAM-s, FastSAM-x, and MobileSAM exhibited overflow, including parts of the background, whereas SAM_VIT_b and SAM_VIT_l did not face this issue. Figure 4.6 illustrates the mask outputs of SAM_VIT_b and SAM_VIT_l when the IC was rotated. Notably, there was no zig-zag pattern observed at the border of the predicted mask, in contrast to results obtained from the YOLOv5 segmentation model. Table 4.4 provides the FPS performance for different SAM models.

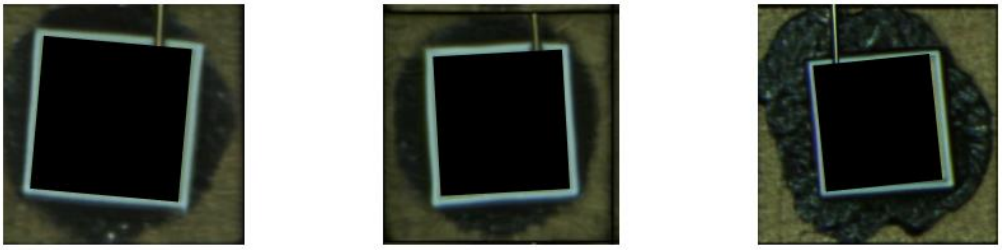
4.1.4 Discussion

Since real-time inference was not a requirement for this study, an ensemble model of YOLOv5n and SAM_VIT_b was chosen as the final model for image pre-processing. However, if real-time inference were necessary, YOLOv5n-seg would be the preferred choice, while YOLOv5s-seg would not be considered due to its similar results to YOLOv5n-seg but with slower FPS and higher computational cost. Additionally, FastSAM-s, FastSAM-x, and MobileSAM were excluded from consideration due to the overflow issues observed in the predicted masks. Furthermore, SAM_VIT_l would also not be considered as it exhibited similar performance to SAM_VIT_b but with lower FPS. The trade-off between the SAM_VIT_b and YOLOv5n-seg is evident, as SAM_VIT_b operates at 3.3 FPS, whereas YOLOv5n-seg runs at a significantly higher speed of 120.26 FPS.

4.2 Comparison of StyleGANv2, StyleGANv2, and Stable Diffusion

Table 4.5： FID Scores Comparison: StyleGANv2 vs. StyleGANv3

FID score	Die Rotate	Die Crack
StyleGANv2	22.32	14.86
StyleGANv3	81.43	76.62



(a) StyleGANv2 (b) StyleGANv3 (c) Stable Diffusion

Figure 4.7: Generated Die Rotation Images

In table 4.5, the FID scores for StyleGANv2 and StyleGANv3 in generating die crack and die rotation defect images are presented. Notably, StyleGANv2 outperformed StyleGANv3, achieving significantly lower FID scores of 22.32 for die rotation and 14.86 for die crack. In Figure 4.7, images generated by StyleGANv2, StyleGANv3, and stable diffusion are showcased. In terms of image fidelity, it is evident that the images generated by StyleGANv2 closely resemble or are more similar to the real images. Conversely, the dies generated by StyleGANv3 do not maintain straight edges, exhibiting distortions. Although the dies generated by stable diffusion preserve straight edges, the surrounding areas of the die suffer from lower fidelity. Considering these observations, the final model selected for generating additional images is StyleGANv2. StyleGANv3's poor performance may be due to a few reasons, which might not have had enough training data, and some hyperparameters, like "r1_gamma", may not have been tuned correctly.

4.3 Comparison of YOLOv7-tiny, Modified YOLOv7-tiny, and Pruned YOLOv7-tiny

Table 4.6: Metrics and Computational Cost of Various YOLOv7-tiny Models

Settings	**NWD**	**CoordConv**	**Slim-neck by GSConv**	**Pruning**	**Precision**	**Recall**	**mAP@0.5**	**mAP@0.5:0.95**	**Params**	**GFLOPS**	**FPS(b16)**
Baseline					97.4	87.0	92.1	69.8	6010302	13.0	19.5
Setting-1	✓				**98.0**	88.3	92.1	69.6	6010302	13.0	19.2
Setting-2	✓		✓		**98.0**	92.2	94.2	69.7	5746078	12.1	18.2
Setting-3	✓		✓	Slimming	96.7	89.9	93.2	69.5	3012905	**8.0**	19.8
Setting-4	✓		✓	LAMP	97.0	92.2	94.2	70.6	**2078651**	**8.0**	**21.1**
Setting-5	✓	✓			95.8	92.2	94.2	70.9	6027198	13.1	17.8
Setting-6	✓	✓		Slimming	94.8	89.4	93.6	71.9	2855550	8.7	19.4
Setting-7	✓	✓		LAMP	95.7	**93.8**	**95.1**	**72.3**	2082812	8.6	20.6

4.3.1 Baseline

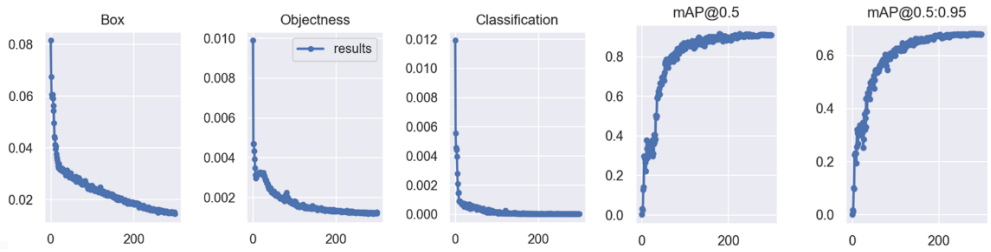


Figure 4.8: Loss and mAP Graphs for Baseline Model

In Figure 4.8, the graphical representations of box loss, objectness loss, classification loss, mAP@0.5, and mAP@0.5:0.95 are presented. Analysis of the mAP graph reveals that the model reaches convergence approximately at the 200th epoch, leading to the decision to train the model for 300 epochs.

4.3.2 Setting-1 Model

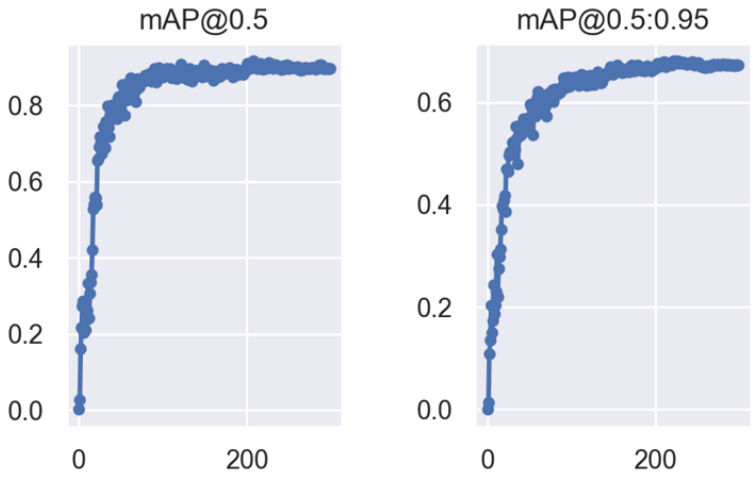


Figure 4.9: mAP Graph for Setting-1 Model

Table 4.6 provides insights into the impact of the NWD loss function (setting-1 model) on model performance. Notably, this setting resulted in a 0.6% increase in precision and a 1.3% improvement in recall. However, it is noteworthy that there was no change in mAP@0.5, and mAP@0.5:0.95 experienced a 0.2% drop. Figure 4.9 illustrates the mAP graph for the setting-1 model, which demonstrates that this configuration achieved faster convergence compared to the baseline model.

4.3.3 Setting-2 Model

For the setting-2 model, which incorporates NWD loss and a slim neck architecture, similar to the setting-1 model, there is a noteworthy improvement in precision, increasing by 0.6% compared to the baseline. The recall also exhibits substantial growth, with a remarkable 5.2% improvement. Furthermore, the mAP@0.5 metric experiences a 2.1% increase. However, there is a 0.1% decrease in mAP@0.5:0.95. Interestingly, despite these performance enhancements, the computational cost of Setting-2 is lower than the baseline model. Parameters have decreased by 4.4%, and GFLOPS have reduced by 6.92%. However, the FPS is 1.3 lower, possibly due to the slower speed in depth-wise operations.

4.3.4 Setting-3 Model

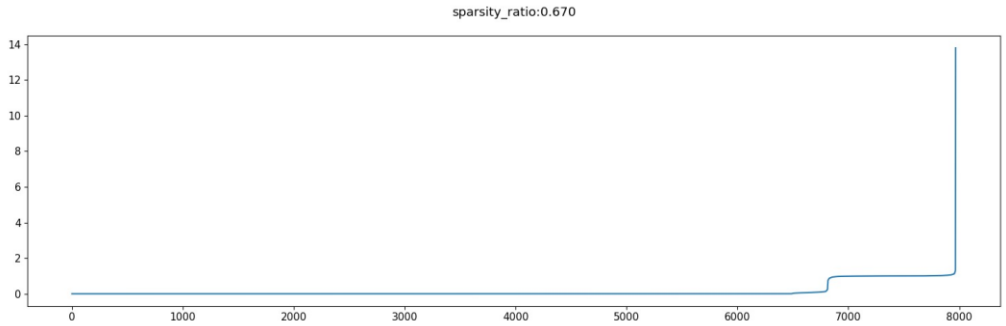


Figure 4.10: Graph of Ordering of Batch Normalization Parameters for Setting-3 Model

The setting-3 model is essentially the pruned version of the setting-2 model, achieved through network slimming. In Figure 4.9, the graph displays the ordering of batch normalization parameters after the end of sparsity learning. The x-axis represents the number of batch normalization parameters, and the y-axis represents the corresponding values of these parameters. From the graph, it is evident that 67.0% of the batch normalization values are close to zero, representing a sparsity ratio in the model. Compared to the setting-2 model, there is a drop in precision by 1.3%, recall by 2.3%, mAP@0.5 by 1%, and mAP@0.5:0.95 by 0.2%. However, these trade-offs are accompanied by

significantly lower computational costs, with 47.5% fewer parameters and 33.8% less GLOPS. Additionally, the FPS has increased by 1.6.

4.3.5 Setting-4 Model

The setting-4 model is a pruned version of the setting-2 model, utilizing LAMP pruning. Notably, the pruning results in setting-4 model are significantly better compared to setting-3 model. In comparison to the setting-2 model, there is a 1% drop in precision, while recall and mAP@0.5 remain unchanged. Moreover, mAP@0.5:0.95 increases by 0.9%. Furthermore, the model demonstrates substantial computational improvements, with GFLOPS reduced by 33.8%, which follows a similar trend as observed in the setting-3 model. Additionally, it boasts a significantly lower parameter count, with 63.82% fewer parameters compared to the setting-2 model, and the FPS has increased by 2.9.

4.3.6 Setting-5 Model

The setting-5 model, incorporating the NWD loss function and CoordConv neck, exhibits notable differences when compared to the baseline model. Specifically, there is a significant drop in precision, decreasing by 1.6%. Conversely, recall, mAP@0.5, and mAP@0.5:0.95 experience improvements, increasing by 5.2%, 2.1%, and 0.2%, respectively. However, it's worth noting that these enhancements come at slightly higher parameters and GLOPS, which are 0.28% and 0.77% higher, respectively.

4.3.7 Setting-6 Model

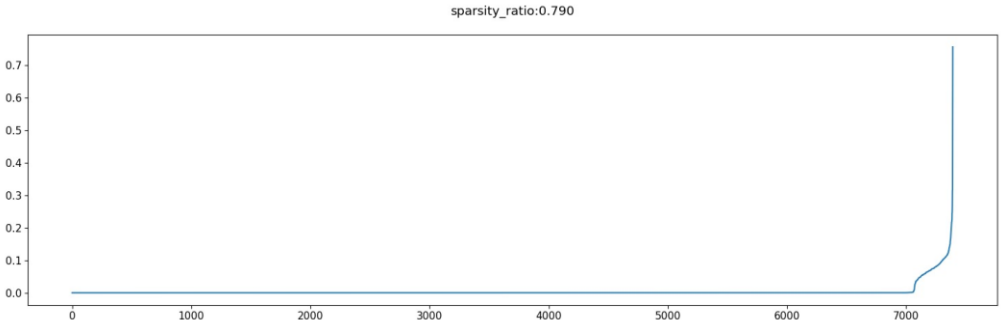


Figure 4.11: Graph of Ordering of Batch Normalization Parameters for Setting-6 Model

Setting-6 is the result of implementing network slimming on Setting-5. Comparatively, when measured against the Setting-5 model, Setting-6 demonstrates notable trade-offs. Specifically, there's a marked decline in precision by 1%, recall by 2.8%, and mAP@0.5 by 0.6%. Conversely, mAP@0.5:0.95 exhibits an encouraging 1% increase. Efficiency-wise, Setting-6 model delivers significant benefits, despite these performance shifts. The model parameters see a substantial reduction of 52.62%, accompanied by a GLOPS reduction of 36.49%. Furthermore, Setting-6 experiences a notable boost in FPS, increasing by 1.6. From Figure 4.11, it's evident that the sparsity ratio of setting-6 model is 79.0%, which represents a significant increase of 12% compared to setting-3. Notably, this high sparsity ratio means that setting-6 model can focus on pruning the majority of the batch normalization parameters, with 21% of these parameters remaining in focus. This indicates that setting-6 model has substantial potential for reducing computational costs.

4.3.8 Setting-7 Model

With the implementation of LAMP pruning applied to the settings-5 model, resulting in the creation of the setting-7 model, several notable performance and efficiency changes are observed when compared to setting-5. There is a slight drop in precision by 0.1%, which is accompanied by significant improvements in recall, mAP@0.5, and mAP@0.5:0.95, increasing by 1.6%, 0.9%, and 1.4%, respectively. However, the most substantial gains come in terms of computational efficiency. Setting-6 boasts a remarkable reduction of 65.44% in model parameters and a considerable decrease of 34.35% in GLOPs. Additionally, the model experiences enhanced speed, with an impressive 2.8 FPS increase.

4.3.9 Discussion

From Table 4.6, it can be deduced that LAMP pruning outperforms network slimming both in terms of computational cost and metrics. Some models, such as the setting-7 model, exhibit improved performance in term of metrics after pruning. While LAMP pruning achieves a greater reduction in parameters and an increased FPS compared to network slimming, based on the observed

sparsity ratio, it's noteworthy that network slimming still holds significant potential for further decreasing computational costs. This potential, however, was not fully demonstrated in this study due to the "speed_up" rate being set to 1.5 for the sake of fair comparison, as explained in Section 3.3.7.4. Additionally, enhancing network performance by modifying its components presents a challenge in this study. Many modules that focus on improving the detection of small objects may trade off performance for larger objects. The datasets used in this study contain both small and large objects, and making such modifications typically results in inferior performance compared to the baseline.

Considering the computational cost, mAP@0.5, and mAP@0.5:0.95, Setting-7 emerges as the top-performing model. Compared to the baseline, Setting-7 showcases notable improvements in Recall, mAP@0.5, and mAP@0.5:0.95, with enhancements of 6.8%, 3%, and 2.5%, respectively. However, there is a slight reduction in precision by 1.7%. Setting-1 and Setting-2 exhibit the highest precision, with an increase of 0.6%. Setting-4 stands out for having the fewest parameters, the lowest GFLOPS, and the highest FPS. It achieves a substantial reduction in GFLOPS by 65.42% and a 38.46% decrease in parameters while slightly increasing FPS by 1.6.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

In conclusion, this study introduced a comprehensive deep learning visual-based inspection approach based on object detection techniques. The preprocessing stage employs an ensemble model of YOLOv5n and SAM_VIT_b, followed by object detection using a modified YOLOv7-tiny model (Setting-7). This approach effectively detects die rotation and die crack defects, achieving impressive results with a 95.1% of mAP@0.5 and 72.3% of mAP@0.5:0.95, representing a 3% and 2.5% improvement over the original YOLOv7-tiny model. Furthermore, Setting-7 significantly reduces parameters by 65.34% and GLOPS by 33.84% compared to the original YOLOv7-tiny network.

By automating defect detection in integrated circuits, this approach minimizes the need for human intervention, allowing human resources to be allocated to more critical tasks. This enhancement ultimately enhances operational efficiency within the industry. In this study, all objectives were successfully achieved, and research questions were addressed. StyleGANv2 was chosen as the optimal solution for addressing the issue of insufficient dataset, achieving an FID score of 22.32 for die rotation and 14.86 for die crack. SAM_VIT_b emerged as the preferred technique for capturing ROI, while NWD loss function and Coordconv neck contributed to the enhancement of the YOLOv7-tiny model. Additionally, LAMP pruning effectively reduced the computational cost of YOLOv7-tiny, completing the toolbox for this innovative visual inspection system.

Several future enhancements are recommended:

- Incorporating Fine-Tuning with Multimodal Techniques:
  Consider integrating fine-tuning methods that combine stable diffusion with textual inversion. This can enable the generation of die rotations and die cracks based on user-specified degrees and locations, adding flexibility and customization to the generated image.

- Exploring Pruning Techniques for the Segmentation Model:

Investigate pruning techniques for segmentation model, particularly for the SAM model used in segmentation.

- Incorporating a Module for Die Rotation Angle Determination:
  Develop and implement a module that can determine and display the die rotation angles detected by the YOLOv7-tiny model. This would provide valuable information for inspection and analysis.

- Application of Knowledge Distillation:
  Explore the application of knowledge distillation techniques. This approach has the potential to enhance the mAP of the pruned model, contributing to improved defect detection performance.

REFERENCES

Ahmed Fawzy Gad, 2021. *Faster R-CNN Explained for Object Detection Tasks*. [online] Available at: <https://blog.paperspace.com/faster-r-cnn-explained-object-detection/> [Accessed 10 April 2023].

Alam, L. and Kehtarnavaz, N., 2022. *A Survey of Detection Methods for Die Attachment and Wire Bonding Defects in Integrated Circuit Manufacturing*. *IEEE Access*, https://doi.org/10.1109/ACCESS.2022.3197624.

Arena Solutions, 2023. *WHAT IS INTEGRATED CIRCUIT (IC)?* [online] Available at: <https://www.arenasolutions.com/resources/glossary/integrated-circuit/> [Accessed 9 March 2023].

Aryan, P., Sampath, S. and Sohn, H., 2018. *An overview of non-destructive testing methods for integrated circuit packaging inspection*. *Sensors (Switzerland)*, https://doi.org/10.3390/s18071981.

Batool, U., Shapiai, M.I., Tahir, M., Ismail, Z.H., Zakaria, N.J. and Elfakharany, A., 2021. A Systematic Review of Deep Learning for Silicon Wafer Defect Recognition. *IEEE Access*. https://doi.org/10.1109/ACCESS.2021.3106171.

Bhatt, P.M., Malhan, R.K., Rajendran, P., Shah, B.C., Thakar, S., Yoon, Y.J. and Gupta, S.K., 2021. *Image-Based Surface Defect Detection Using Deep Learning: A Review*. *Journal of Computing and Information Science in Engineering*, https://doi.org/10.1115/1.4049535.

Bochkovskiy, A., Wang, C.-Y. and Liao, H.-Y.M., 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. [online] Available at: <http://arxiv.org/abs/2004.10934>.

Cao, C., Wang, B., Zhang, W., Zeng, X., Yan, X., Feng, Z., Liu, Y. and Wu, Z., 2019. An Improved Faster R-CNN for Small Object Detection. *IEEE Access*, 7, pp.106838–106846. https://doi.org/10.1109/ACCESS.2019.2932731.

Chen, K., Cai, N., Wu, Z., Xia, H., Zhou, S. and Wang, H., 2023. Multi-scale GAN with transformer for surface defect inspection of IC metal

packages. *Expert Systems with Applications*, 212. https://doi.org/10.1016/j.eswa.2022.118788.

Cheon, S., Lee, H., Kim, C.O. and Lee, S.H., 2019. Convolutional Neural Network for Wafer Surface Defect Classification and the Detection of Unknown Defect Class. *IEEE Transactions on Semiconductor Manufacturing*, 32(2), pp.163–170. https://doi.org/10.1109/TSM.2019.2902657.

Dehaerne, E., Dey, B., Halder, S. and De Gendt, S., 2023. Optimizing YOLOv7 for Semiconductor Defect Detection. [online] Available at: <http://arxiv.org/abs/2302.09565>.

Eggert, C., Brehm, S., Winschel, A., Zecha, D. and Lienhart, R., 2017. A closer look: Small object detection in faster R-CNN. In: *2017 IEEE International Conference on Multimedia and Expo (ICME)*. [online] IEEE. pp.421–426. https://doi.org/10.1109/ICME.2017.8019550.

Fang, G., Ma, X., Song, M., Mi, M.B. and Wang, X., 2023. DepGraph: Towards Any Structural Pruning. [online] Available at: <http://arxiv.org/abs/2301.12900>.

Girshick, R., 2015. Fast R-CNN. [online] Available at: <http://arxiv.org/abs/1504.08083>.

Girshick, R., Donahue, J., Darrell, T. and Malik, J., 2013. Rich feature hierarchies for accurate object detection and semantic segmentation. [online] Available at: <http://arxiv.org/abs/1311.2524>.

Jacob Solawetz, 2020. *What is YOLOv5? A Guide for Beginners*. [online] Available at: <https://blog.roboflow.com/yolov5-improvements-and-evaluation/> [Accessed 14 April 2023].

Jin, Q. and Chen, L., 2022. A Survey of Surface Defect Detection of Industrial Products Based on A Small Number of Labeled Data. [online] Available at: <http://arxiv.org/abs/2203.05733>.

Jonathan Hui, 2018. *mAP (mean Average Precision) for Object Detection*. [online] Available at: <https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173> [Accessed 17 April 2023].

Kiprono Elijah Koech, 2020. *Object Detection Metrics With Worked Example*. [online] Available at: <https://towardsdatascience.com/on-object-detection-metrics-with-worked-example-216f173ed31e> [Accessed 17 April 2023].

Lee, J., Park, S., Mo, S., Ahn, S. and Shin, J., 2020. Layer-adaptive sparsity for the Magnitude-based Pruning. [online] Available at: <http://arxiv.org/abs/2010.07611>.

Li, C., Li, L., Jiang, H., Weng, K., Geng, Y., Li, L., Ke, Z., Li, Q., Cheng, M., Nie, W., Li, Y., Zhang, B., Liang, Y., Zhou, L., Xu, X., Chu, X., Wei, X. and Wei, X., 2022a. YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications. [online] Available at: <http://arxiv.org/abs/2209.02976>.

Li, H., Li, J., Wei, H., Liu, Z., Zhan, Z. and Ren, Q., 2022b. Slim-neck by GSConv: A better design paradigm of detector architectures for autonomous vehicles. [online] Available at: <http://arxiv.org/abs/2206.02424>.

Lim, J.Y., Lim, J.Y., Baskaran, V.M. and Wang, X., 2023. A deep context learning based PCB defect detection model with anomalous trend alarming system. *Results in Engineering*, 17. https://doi.org/10.1016/j.rineng.2023.100968.

Liu, R., Lehman, J., Molino, P., Such, F.P., Frank, E., Sergeev, A. and Yosinski, J., 2018. An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution. [online] Available at: <http://arxiv.org/abs/1807.03247>.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y. and Berg, A.C., 2015. SSD: Single Shot MultiBox Detector. [online] https://doi.org/10.1007/978-3-319-46448-0_2.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S. and Zhang, C., 2017. Learning Efficient Convolutional Networks through Network Slimming. [online] Available at: <http://arxiv.org/abs/1708.06519>.

Lu, Y., Sun, C., Li, X. and Cheng, L., 2022. Defect Detection of Integrated Circuit Based on YOLOv5. In: *2022 IEEE 2nd International Conference on Computer Communication and Artificial Intelligence,*

CCAI 2022. Institute of Electrical and Electronics Engineers Inc. pp.165–170. https://doi.org/10.1109/CCAI55564.2022.9807758.

Mahony, N.O., Campbell, S., Carvalho, A., Harapanahalli, S., Velasco-Hernandez, G., Krpalkova, L., Riordan, D. and Walsh, J., 2019. Deep Learning vs. Traditional Computer Vision. [online] https://doi.org/10.1007/978-3-030-17795-9.

Mat Jizat, J.A., P.P. Abdul Majeed, A., Ahmad, A.F., Taha, Z. and Yuen, E., 2021. Evaluation of the machine learning classifier in wafer defects classification. *ICT Express*, 7(4), pp.535–539. https://doi.org/10.1016/j.icte.2021.04.007.

O'Shea, K. and Nash, R., 2015. An Introduction to Convolutional Neural Networks. [online] Available at: <http://arxiv.org/abs/1511.08458>.

Padilla, R., Netto, S.L. and da Silva, E.A.B., 2020. A Survey on Performance Metrics for Object-Detection Algorithms. In: *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*. [online] IEEE. pp.237–242. https://doi.org/10.1109/IWSSIP48289.2020.9145130.

Prince, S.J.D., 2023. *Understanding Deep Learning*. [online] Available at: <https://udlbook.github.io/udlbook/> [Accessed 19 February 2023].

Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2015. You Only Look Once: Unified, Real-Time Object Detection. [online] Available at: <http://arxiv.org/abs/1506.02640>.

Redmon, J. and Farhadi, A., 2016. YOLO9000: Better, Faster, Stronger. [online] Available at: <http://arxiv.org/abs/1612.08242>.

Redmon, J. and Farhadi, A., 2018. YOLOv3: An Incremental Improvement. [online] Available at: <http://arxiv.org/abs/1804.02767>.

Ren, S., He, K., Girshick, R. and Sun, J., 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. [online] Available at: <http://arxiv.org/abs/1506.01497>.

Shankar, N.G. and Zhong, Z.W., 2005. Defect detection on semiconductor wafer surfaces. *Microelectronic Engineering*, 77(3–4), pp.337–346. https://doi.org/10.1016/j.mee.2004.12.003.

Shrestha, A. and Mahmood, A., 2019. *Review of deep learning algorithms and architectures*. *IEEE Access*, https://doi.org/10.1109/ACCESS.2019.2912200.

Ultralytics, 2023. *Architecture Summary*. [online] Available at: <https://docs.ultralytics.com/yolov5/architecture/> [Accessed 14 April 2023].

Wang, C.-Y., Bochkovskiy, A. and Liao, H.-Y.M., 2022. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. [online] Available at: <http://arxiv.org/abs/2207.02696>.

Wang, H. and Raj, B., 2017. On the Origin of Deep Learning. [online] Available at: <http://arxiv.org/abs/1702.07800>.

Wang, J., Xu, C., Yang, W. and Yu, L., 2021. A Normalized Gaussian Wasserstein Distance for Tiny Object Detection. [online] Available at: <http://arxiv.org/abs/2110.13389>.

Yang, Y. and Sun, M., 2022a. Semiconductor Defect Detection by Hybrid Classical-Quantum Deep Learning. [online] Available at: <http://arxiv.org/abs/2208.03514>.

Yang, Y. and Sun, M., 2022b. Semiconductor Defect Pattern Classification by Self-Proliferation-and-Attention Neural Network. *IEEE Transactions on Semiconductor Manufacturing*, 35(1), pp.16–23. https://doi.org/10.1109/TSM.2021.3131597.

Yeh, C.H., Wu, F.C., Ji, W.L. and Huang, C.Y., 2010. A wavelet-based approach in detecting visual defects on semiconductor wafer dies. In: *IEEE Transactions on Semiconductor Manufacturing*. pp.284–292. https://doi.org/10.1109/TSM.2010.2046108.

You, K.M., Sheikh, U.U. and Alias, N.E.B., 2022. Die-Level Defects Classification using Region-based Convolutional Neural Network. In: *IEEE International Conference on Semiconductor Electronics, Proceedings, ICSE*. Institute of Electrical and Electronics Engineers Inc. pp.144–147. https://doi.org/10.1109/ICSE56004.2022.9863135.

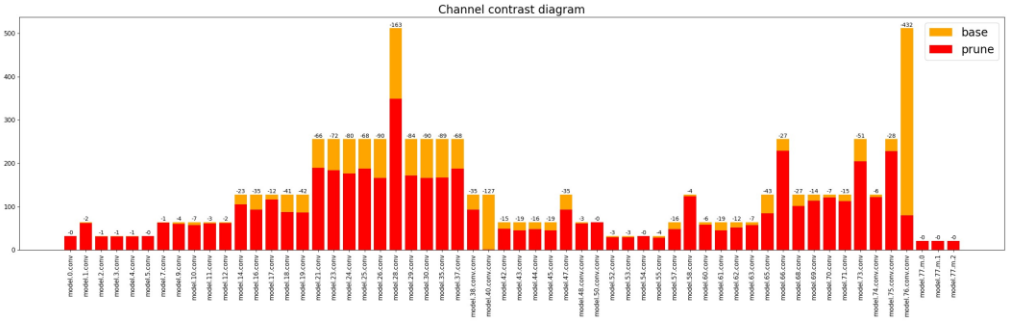
Zaidi, S.S.A., Ansari, M.S., Aslam, A., Kanwal, N., Asghar, M. and Lee, B., 2021. A Survey of Modern Deep Learning based Object Detection Models. [online] Available at: <http://arxiv.org/abs/2104.11892>.

Zhang, A., Lipton, Z.C., Li, M. and Smola, A.J., 2021. Dive into Deep Learning. [online] Available at: <http://arxiv.org/abs/2106.11342>.

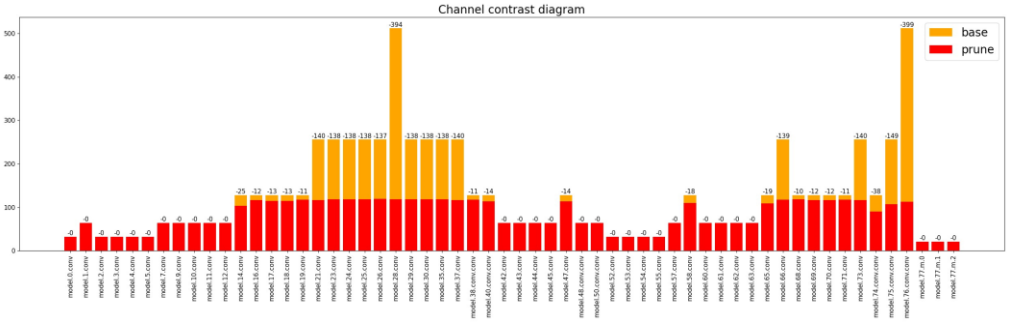
Zou, Z., Chen, K., Shi, Z., Guo, Y. and Ye, J., 2019. Object Detection in 20 Years: A Survey. [online] Available at: <http://arxiv.org/abs/1905.05055>.

APPENDICES

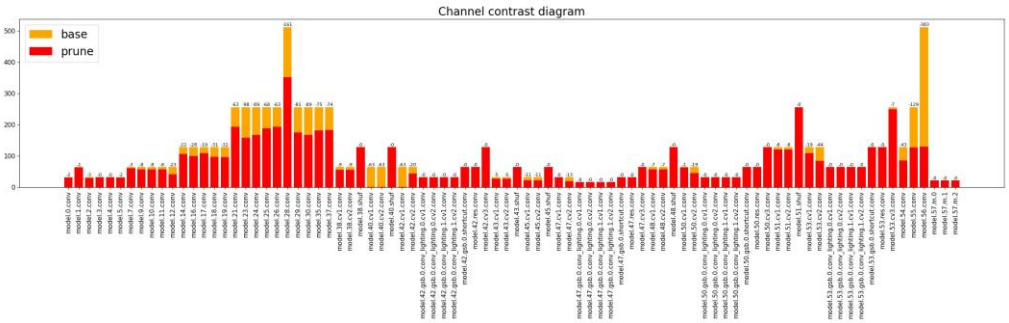
Appendix A: Comparison of Channel Graphs between Setting-2 and Setting-3 Models



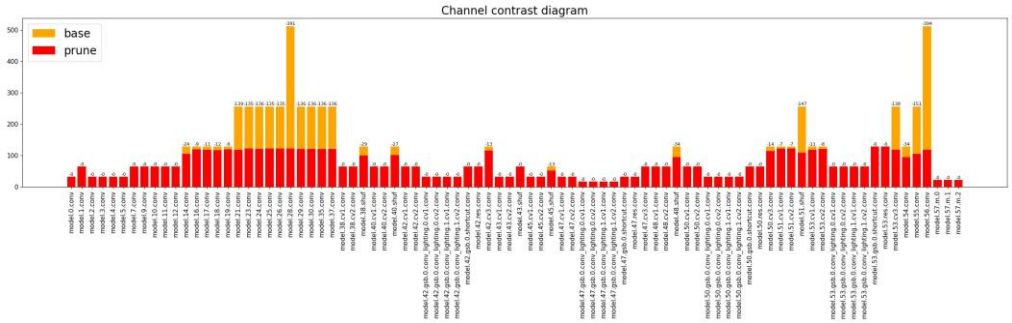
Appendix B: Comparison of Channel Graphs between Setting-2 and Setting-4 Models



Appendix C: Comparison of Channel Graphs between Setting-5 and Setting-6 Models



Appendix D: Comparison of Channel Graphs between Setting-5 and Setting-7 Models



Appendix E: Configuration File for StyleGANv2

```
_base_ = [
    '../_base_/datasets/unconditional_imgs_flip_256x256.py', '../_base_/models/base_styleganv2.py',
    '../_base_/gen_default_runtime.py'
]

load_from='https://download.openmmlab.com/mmediting/stylegan2/stylegan2_c2_ffhq_256_b4x8_20210407_160709-7890ae1f.pth'
# reg params
d_reg_interval = 16
g_reg_interval = 4

g_reg_ratio = g_reg_interval / (g_reg_interval + 1)
d_reg_ratio = d_reg_interval / (d_reg_interval + 1)

ema_half_life = 10.  # G_smoothing_kimg

aug_kwargs = {
    'xflip': 1,
    'rotate90': 0,
    'xint': 1,
    'scale': 1,
    'rotate': 0,
    'aniso': 1,
    'xfrac': 1,
    'brightness': 1,
    'contrast': 1,
    'lumaflip': 1,
    'hue': 1,
    'saturation': 1
}


model = dict(
    generator=dict(out_size=256),
    discriminator=dict(in_size=256,type='ADAStyleGAN2Discriminator', #
                       data_aug=dict(type='ADAAug', aug_pipeline=aug_kwargs, ada_kimg=100)),
    ema_config=dict(
        type='ExponentialMovingAverage',
        interval=1,
        momentum=1. - (0.5**(32. / (ema_half_life * 1000.)))),
    loss_config=dict(
        r1_loss_weight=10. / 2. * d_reg_interval,
        r1_interval=d_reg_interval,
        norm_mode='HWC',
        g_reg_interval=g_reg_interval,
        g_reg_weight=2. * g_reg_interval,
        pl_batch_shrink=2))

train_cfg = dict(max_iters=50000,val_interval=1000)

optim_wrapper = dict(
    generator=dict(
        optimizer=dict(
            type='Adam', lr=0.002 * g_reg_ratio, betas=(0,
                                                        0.99**g_reg_ratio))),
    discriminator=dict(
        optimizer=dict(
            type='Adam', lr=0.002 * d_reg_ratio, betas=(0,
                                                        0.99**d_reg_ratio))))

batch_size = 2
data_root = 'C:/Users/tackh/mmgeneration/data/rotate2'

train_dataloader = dict(
    batch_size=batch_size,num_workers=2, dataset=dict(data_root=data_root))

val_dataloader = dict(batch_size=batch_size,num_workers=2, dataset=dict(data_root=data_root))

test_dataloader = dict(
    batch_size=batch_size,num_workers=2, dataset=dict(data_root=data_root))

# VIS_HOOK
custom_hooks = [
    dict(
        type='VisualizationHook',
        interval=500,
        fixed_input=True,
        vis_kwargs_list=dict(type='GAN', name='fake_img'))
]

inception_pkl='C:/Users/tackh/mmgeneration/work_dirs/inception_pkl/rotate2.pkl'
num_sample = 1000
# METRICS
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-50k',
        fake_nums=num_sample,
        real_nums=num_sample,
        inception_style='StyleGAN',
        inception_pkl=inception_pkl,
        sample_model='ema'),
    dict(type='PrecisionAndRecall', fake_nums=num_sample, prefix='PR-50K'),
    dict(type='PerceptualPathLength', fake_nums=num_sample, prefix='ppl-w')
]
# NOTE: config for save multi best checkpoints
# default_hooks = dict(
#     checkpoint=dict(
#         save_best=['FID-Full-50k/fid', 'IS-50k/is'],
#         rule=['less', 'greater']))
default_hooks = dict(checkpoint=dict(save_best='FID-50k/fid',interval=1000, by_epoch=False, max_keep_ckpts=50),
                     logger=dict(type='LoggerHook',interval=100))

val_evaluator = dict(type='Evaluator',metrics=metrics)
test_evaluator = val_evaluator
```

Appendix F: Configuration File for StyleGANv3

```
_base_ = [
    '../_base_/models/base_styleganv3.py',
    '../_base_/datasets/unconditional_imgs_flip_lanczos_resize_256x256.py',
    '../_base_/gen_default_runtime.py',
]

synthesis_cfg = {
    'type': 'SynthesisNetwork',
    'channel_base': 32768,
    'channel_max': 512,
    'magnitude_ema_beta': 0.999
}
r1_gamma = 2  # set by user
d_reg_interval = 16
g_reg_interval = 4

g_reg_ratio = g_reg_interval / (g_reg_interval + 1)
d_reg_ratio = d_reg_interval / (d_reg_interval + 1)

load_from = 'https://download.openmmlab.com/mmediting/stylegan3/stylegan3_t_ffhq_1024_b4x8_cvt_official_rgb_20220329_235113-db6c6580.pth'  # noqa
# ada settings
aug_kwargs = {
    'xflip': 1,
    'rotate90': 0,
    'xint': 1,
    'scale': 1,
    'rotate': 0,
    'aniso': 1,
    'xfrac': 1,
    'brightness': 1,
    'contrast': 1,
    'lumaflip': 1,
    'hue': 1,
    'saturation': 1
}

ema_half_life = 10.  # G_smoothing_kimg

ema_kimg = 10
ema_nimg = ema_kimg * 1000
ema_beta = 0.5**(32 / max(ema_nimg, 1e-8))

ema_config = dict(
    type='ExponentialMovingAverage',
    interval=1,
    momentum=ema_beta,
    start_iter=0)

model = dict(
    generator=dict(
        out_size=256,
        img_channels=3,
        rgb2bgr=True,
        synthesis_cfg=synthesis_cfg),
    discriminator=dict(
        type='ADAStyleGAN2Discriminator',
        in_size=256,
        input_bgr2rgb=True,
        data_aug=dict(type='ADAAug', aug_pipeline=aug_kwargs, ada_kimg=100)),
    loss_config=dict(r1_loss_weight=r1_gamma / 2.0 * d_reg_interval),
    ema_config=ema_config)
optim_wrapper = dict(
     generator=dict(
         optimizer=dict(
             type='Adam', lr=0.0025 * g_reg_ratio, betas=(0,
                                                          0.99**g_reg_ratio))),
     discriminator=dict(
         optimizer=dict(
             type='Adam', lr=0.002 * d_reg_ratio, betas=(0,
                                                         0.99**d_reg_ratio))))

batch_size = 2
data_root = 'C:/Users/tackh/mmgeneration/data/rotate2'

train_dataloader = dict(
    batch_size=batch_size, num_workers=2,dataset=dict(data_root=data_root))

val_dataloader = dict(batch_size=batch_size, num_workers=2,dataset=dict(data_root=data_root))

test_dataloader = dict(
    batch_size=batch_size,num_workers=2, dataset=dict(data_root=data_root))

train_cfg = dict(max_iters=50000,val_interval=1000)

# VIS_HOOK
custom_hooks = [
    dict(
        type='VisualizationHook',
        interval=500,
        fixed_input=True,
        vis_kwargs_list=dict(type='GAN', name='fake_img'))
]
inception_pkl='C:/Users/tackh/mmgeneration/work_dirs/inception_pkl/rotate2.pkl'
num_sample = 1000
# METRICS
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=num_sample,
        inception_pkl=inception_pkl,
        inception_style='StyleGAN',
        sample_model='ema')
]

default_hooks = dict(checkpoint=dict(save_best='FID-Full-50k/fid',interval=1000, by_epoch=False, max_keep_ckpts=50),
                     logger=dict(type='LoggerHook',interval=100))

val_evaluator = dict(metrics=metrics)
test_evaluator = val_evaluator
```

Appendix G: Configuration File for Stable Diffusion Fine-Tune with LoRA

```
_base_ = '../_base_/gen_default_runtime.py'
dtype = 'float32'

# config for model
stable_diffusion_v15_url = 'runwayml/stable-diffusion-v1-5'

val_prompts = [
    'die crack'
]

lora_config = dict(target_modules=['to_q', 'to_k', 'to_v'])
model = dict(

    type='DreamBooth',
    data_preprocessor=dict(type='DataPreprocessor'),
    vae=dict(
        type='AutoencoderKL',
        from_pretrained=stable_diffusion_v15_url,
        dtype = dtype,
        subfolder='vae'),
    unet=dict(
        type='UNet2DConditionModel',
        subfolder='unet',
        dtype = dtype,
        from_pretrained=stable_diffusion_v15_url),
    text_encoder=dict(
        type='ClipWrapper',
        clip_type='huggingface',
        pretrained_model_name_or_path=stable_diffusion_v15_url,
        subfolder='text_encoder'),
    tokenizer=stable_diffusion_v15_url,
    scheduler=dict(
        type='DDPMScheduler',
        from_pretrained=stable_diffusion_v15_url,
        subfolder='scheduler'),
    test_scheduler=dict(
        type='DDIMScheduler',
        from_pretrained=stable_diffusion_v15_url,
        subfolder='scheduler'),

    prior_loss_weight=0,
    val_prompts=val_prompts,
    lora_config=lora_config)

train_cfg = dict(max_iters=50000)

optim_wrapper = dict(
    # Only optimize LoRA mappings
    modules='.*.lora_mapping',
    # NOTE: lr should be larger than dreambooth finetuning
    optimizer=dict(type='AdamW', lr=5e-4),
    accumulative_counts=1)

pipeline = [
    dict(type='LoadImageFromFile',key='img', channel_order='rgb'),
    dict(type='Resize', scale=(256, 256)),
    dict(type='PackInputs')
]
dataset = dict(
    type='DreamBoothDataset',
    data_root='C:/Users/tackh/mmgeneration/data/crack_lora',
    # TODO: rename to instance
    concept_dir='imgs',
    prompt='die crack',
    pipeline=pipeline)

train_dataloader = dict(
    dataset=dataset,
    num_workers=2,
    sampler=dict(type='DefaultSampler', shuffle=True),
    persistent_workers=True,
    batch_size=1)

val_cfg = val_evaluator = val_dataloader = None
test_cfg = test_evaluator = test_dataloader = None

default_hooks = dict(checkpoint=dict(interval=10000,
 by_epoch=False, max_keep_ckpts=50),logger=dict(interval=10))
custom_hooks = [
    dict(
        type='VisualizationHook',
        interval=100,
        fixed_input=True,
        # visualize train dataset
        vis_kwargs_list=dict(type='Data', name='fake_img'),
        n_samples=1)
]
```

Appendix H: Model Configuration File for Original YOLOv7-tiny Neck

```
# yolov7-tiny head
head:
  [[-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, SP, [5]],
   [-2, 1, SP, [9]],
   [-3, 1, SP, [13]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -7], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 37

   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [21, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]], # route backbone P4
   [[-1, -2], 1, Concat, [1]],

   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 47

   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [14, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]], # route backbone P3
   [[-1, -2], 1, Concat, [1]],

   [-1, 1, Conv, [32, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [32, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [32, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [32, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 57

   [-1, 1, Conv, [128, 3, 2, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, 47], 1, Concat, [1]],

   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 65

   [-1, 1, Conv, [256, 3, 2, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, 37], 1, Concat, [1]],

   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [128, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [128, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 73

   [57, 1, Conv, [128, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [65, 1, Conv, [256, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [73, 1, Conv, [512, 3, 1, None, 1, nn.LeakyReLU(0.1)]],

   [[74,75,76], 1, IDetect, [nc, anchors]],   # Detect(P3, P4, P5)
  ]
```

Appendix I: Model Configuration File for Modified Neck with CoordConv

```
head:
  [[-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, SP, [5]],
   [-2, 1, SP, [9]],
   [-3, 1, SP, [13]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -7], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 37

   [-1, 1, CoordConv, [128, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [21, 1, CoordConv, [128, 1, 1]], # route backbone P4
   [[-1, -2], 1, Concat, [1]],

   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 47

   [-1, 1, CoordConv, [64, 1, 1 ]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [14, 1, CoordConv, [64, 1, 1]], # route backbone P3
   [[-1, -2], 1, Concat, [1]],

   [-1, 1, Conv, [32, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [32, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [32, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [32, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 57

   [-1, 1, Conv, [128, 3, 2, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, 47], 1, Concat, [1]],

   [-1, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [64, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [64, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 65

   [-1, 1, Conv, [256, 3, 2, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, 37], 1, Concat, [1]],

   [-1, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [128, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [128, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, Conv, [128, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 73

   [57, 1, CoordConv, [128, 3, 1]],
   [65, 1, CoordConv, [256, 3, 1]],
   [73, 1, CoordConv, [512, 3, 1]],

   [[74,75,76], 1, IDetect, [nc, anchors]],   # Detect(P3, P4, P5)
  ]
```

Appendix J: Model Configuration File for Modified Neck with VoVGSCSP and GSConvs

```
# yolov7-tiny head
head:
  [[-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-2, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [-1, 1, SP, [5]],
   [-2, 1, SP, [9]],
   [-3, 1, SP, [13]],
   [[-1, -2, -3, -4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, -7], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1, None, 1, nn.LeakyReLU(0.1)]],  # 37

   [-1, 1, GSConvs, [128, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [21, 1, GSConvs, [128, 1, 1]], # route backbone P4
   [[-1, -2], 1, Concat, [1]],

   [-1,1,VoVGSCSP,[128]], #42

   [-1, 1, GSConvs, [64, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [14, 1, GSConvs, [64, 1, 1]], # route backbone P3
   [[-1, -2], 1, Concat, [1]],


    [-1,1,VoVGSCSP,[64]], #47

   [-1, 1, GSConvs, [128, 3, 2, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, 42], 1, Concat, [1]],

   [-1,1,VoVGSCSP,[128]], #50

   [-1, 1, GSConvs, [256, 3, 2, None, 1, nn.LeakyReLU(0.1)]],
   [[-1, 37], 1, Concat, [1]],


   [-1,1,VoVGSCSP,[256]], #53


   [47, 1, Conv, [128, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [50, 1, Conv, [256, 3, 1, None, 1, nn.LeakyReLU(0.1)]],
   [53, 1, Conv, [512, 3, 1, None, 1, nn.LeakyReLU(0.1)]],

   [[54,55,56], 1, IDetect, [nc, anchors]],   # Detect(P3, P4, P5)
  ]
```

Appendix K: CoordConv Implementation in YOLOv7-tiny Neck: Code Snippet

```python
class AddCoords(nn.Module):
    def __init__(self, with_r=False):
        super().__init__()
        self.with_r = with_r

    def forward(self, input_tensor):
        """
        Args:
            input_tensor: shape(batch, channel, x_dim, y_dim)
        """
        batch_size, _, x_dim, y_dim = input_tensor.size()

        xx_channel = torch.arange(x_dim).repeat(1, y_dim, 1)
        yy_channel = torch.arange(y_dim).repeat(1, x_dim, 1).transpose(1, 2)

        xx_channel = xx_channel.float() / (x_dim - 1)
        yy_channel = yy_channel.float() / (y_dim - 1)

        xx_channel = xx_channel * 2 - 1
        yy_channel = yy_channel * 2 - 1

        xx_channel = xx_channel.repeat(batch_size, 1, 1, 1).transpose(2, 3)
        yy_channel = yy_channel.repeat(batch_size, 1, 1, 1).transpose(2, 3)

        ret = torch.cat([
            input_tensor,
            xx_channel.type_as(input_tensor),
            yy_channel.type_as(input_tensor)], dim=1)

        if self.with_r:
            rr = torch.sqrt(torch.pow(xx_channel.type_as(input_tensor) - 0.5, 2) + torch.pow(yy_channel.type_as(input_tensor) - 0.5, 2))
            ret = torch.cat([ret, rr], dim=1)

        return ret
class CoordConv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=1, stride=1, with_r=False):
        super().__init__()
        self.addcoords = AddCoords(with_r=with_r)
        in_channels += 2
        if with_r:
            in_channels += 1
        self.conv = Conv(in_channels, out_channels, k=kernel_size, s=stride)

    def forward(self, x):
        x = self.addcoords(x)
        x = self.conv(x)
        return x
```

Appendix L: GSConv, GSConvs, and VoV-GSCSP Implementation in YOLOv7-tiny Neck: Code Snippet

```
class GSConv(nn.Module):
    # GSConv https://github.com/AlanLi1997/slim-neck-by-gsconv
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=nn.LeakyReLU(0.1)):
        super().__init__()
        c_ = c2 // 2
        self.cv1 = Conv(c1, c_, k, s, p, g, act)
        self.cv2 = Conv(c_, c_, 5, 1, p, c_, act)

    def forward(self, x):
        x1 = self.cv1(x)
        x2 = torch.cat((x1, self.cv2(x1)), 1)
        # shuffle
        # y = x2.reshape(x2.shape[0], 2, x2.shape[1] // 2, x2.shape[2], x2.shape[3])
        # y = y.permute(0, 2, 1, 3, 4)
        # return y.reshape(y.shape[0], -1, y.shape[3], y.shape[4])

        b, n, h, w = x2.size()
        b_n = b * n // 2
        y = x2.reshape(b_n, 2, h * w)
        y = y.permute(1, 0, 2)
        y = y.reshape(2, -1, n // 2, h, w)

        return torch.cat((y[0], y[1]), 1)

class GSConvs(GSConv):
    def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=nn.LeakyReLU(0.1)):
        super().__init__( c1, c2, k, s, p, g, act)
        c_ = c2 // 2
        self.shuf=nn.Conv2d(c_ * 2, c2,1,1,0,bias=False)
        self.act=nn.LeakyReLU(0.1)

    def forward(self, x):
        x1 = self.cv1(x)
        x2 = torch.cat((x1, self.cv2(x1)), 1)
        return self.act(self.shuf(x2))
```

```
class GSBottleneck(nn.Module):
    # GS Bottleneck https://github.com/AlanLi1997/slim-neck-by-gsconv
    def __init__(self, c1, c2, k=3, s=1, e=0.5):
        super().__init__()
        c_ = int(c2*e)
        # for lighting
        self.conv_lighting = nn.Sequential(
            GSConv(c1, c_, 1, 1),
            GSConv(c_, c2, 3, 1, act=False))
        self.shortcut = Conv(c1, c2, 1, 1, act=False)

    def forward(self, x):
        return self.conv_lighting(x) + self.shortcut(x)
```

```
class VoVGSCSP(nn.Module):
    # VoVGSCSP module with GSBottleneck
    def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):
        super().__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = Conv(c1, c_, 1, 1)
        self.cv2 = Conv(c1, c_, 1, 1)
        self.gsb = nn.Sequential(*(GSBottleneck(c_, c_, e=1.0) for _ in range(n)))
        self.res = Conv(c_, c_, 3, 1, act=False)
        self.cv3 = Conv(2 * c_, c2, 1)  #


    def forward(self, x):
        x1 = self.gsb(self.cv1(x))
        y = self.cv2(x)
        return self.cv3(torch.cat((y, x1), dim=1))
```

Appendix M: Normalized Wasserstein Loss Function Implementation in YOLOv7-tiny: Code Snippet

```
def wasserstein_loss(pred, target, eps=1e-7, constant=12.8):


    center1 = pred[:, :2]
    center2 = target[:, :2]

    whs = center1[:, :2] - center2[:, :2]

    center_distance = whs[:, 0] * whs[:, 0] + whs[:, 1] * whs[:, 1] + eps #

    w1 = pred[:, 2]  + eps
    h1 = pred[:, 3]  + eps
    w2 = target[:, 2] + eps
    h2 = target[:, 3] + eps

    wh_distance = ((w1 - w2) ** 2 + (h1 - h2) ** 2) / 4

    wasserstein_2 = center_distance + wh_distance
    return torch.exp(-torch.sqrt(wasserstein_2) / constant)
```