PCB Defect Detection using Computer Vision and Genetic Algorithms

A DISSERTATION PRESENTED

BY

SNEHA NAIDU

TO

THE DEPARTMENT OF COPMUTER SCIENCE

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELORS OF COMPUTER SCIENCE

IN THE SUBJECT OF

ARTIFICIAL INTELLIGENCE

SUPERVISED BY Mohammad Majid Al-Rifaie,
Hooman Oroojeni Mohamad Javad

University of Greenwich
April 2023

Abstract

At the heart of every electronic device, Printed Circuit Boards (PCBs) are the backbone. Their production needs to be done with great care and attention to detail so that they meet the high quality standards of electrical goods. Manufacturing defects, such as missing holes or short circuits, compromise PCB functionality and are often caused by mishandling or technical problems that could possibly happen during the production phase.

This study investigates techniques for automating the visual inspection of Printed Circuit Boards (PCBs) using Computer Vision (CV) methods. Various approaches were studied, including both Convolutional Neural Networks (CNNs) and YOLO architectures. The focus was on prioritizing lightweight solutions suitable for deployment on edge computers on factory floors.

The cutting-edge YOLOv9 model was chosen for custom training due to its recent introduction and outstanding performance benchmarks. To further enhance data quality and improve model accuracy, the RoboFlow tool was utilized for sophisticated image pre-processing and augmentation techniques. Additionally, a Genetic Algorithm was implemented to optimize the model's performance, refining its ability to detect even subtle defects in PCBs.

This research explores how genetic algorithms (GA) can we employed to improve model performance while minimizing detection time. This ensures that the model not only performs efficiently in terms of speed but also maintains robustness. The custom-trained YOLOv9 model with implementation of GA takes an average detection time of 1-2 seconds, demonstrating its suitability for real-time factory floor inspection, where rapid defect detection is critical.

Preface

Printed Circuit Boards (PCBs) form the backbone of modern electronics. As their designs grow increasingly intricate, meticulous quality control has become paramount to prevent costly defects and ensure product functionality. Traditionally, PCB inspection relies heavily on manual processes, checking for:

- **Component Errors:** Missing or misaligned parts (resistors, capacitors, etc.)
- **Circuit Faults:** Open circuits, shorts, trace damage, etc.
- **Solder Defects:** Insufficient solder, bridges, voids, etc.
- **Surface flaws:** Scratches, markings, or contamination.

Automating these inspections with Computer Vision (CV) and AI presents an opportunity for manufacturers to revolutionize their quality processes. Key benefits include:

- AI-powered vision systems can outperform humans in repeated tasks.
- By detecting defect detection early on that is with real-time detection one can lowers scrap rates and minimizes costly rework.
- Computer Vision solutions get rid of subjective opinion, which makes sure that quality standards are always the same.

This project investigates innovative CV technologies for PCB quality control, and employment of Genetic Algorithms. The primary objectives are:

- **Maximize Accuracy of Model:** To maximize the accuracy of the defect detection model, a custom Genetic Algorithm (GA) was introduced. Inspired by biological evolution, this GA approach intelligently explores a wide range of model parameters (hyperparameters). Through a process mimicking natural selection, the GA identifies combinations that lead to superior model performance.
- **Speed:** The solution must achieve real-time defect detection, outpacing manual inspection for efficiency gains.

This project explores the potential of advanced AI and Computer Vision to transform the way manufacturers ensure the quality of their PCBs. By optimizing for speed, adaptability, and employing a unique, custom Genetic Algorithm to boost accuracy. The approach aims to deliver a solution that drives efficiency, reduces waste, and enhances the reliability of the countless electronic devices we rely on.

Acknowledgements

I extend my gratitude to my professors and supervisors for their guidance and support throughout the development of my final year project. Their expertise and insights have been crucial in refining my work and pushing the boundaries of my capabilities. Special thanks to my teachers, whose classes inspired the techniques utilized in this project. The knowledge imparted in these sessions helped in shaping the methodologies that I applied. Thank you all for your mentorship, wisdom, and for investing your time in my development.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the manufacturing world, two things are critical: producing high-quality products and doing so efficiently. This is particularly true in the production of printed circuit boards (PCBs), where traditional inspection methods, such as manual visual inspections and functional testing, are prone to high error rates and inefficiencies. In manual visual inspection , corporations assign a operators to each station and conduct crucial checks with the naked human eye. However this process has limitations like eye fatigue, slow detection speed. On the other hand, in functional inspection approach the PCB is tested to ensure it performs its intended function properly. But this approach necessitates specialized test equipment and is incapable of detecting multiple faults. The article "Manufacturing Tomorrow" [10] states that traditional visual inspection methods have error rates as high as 30%. This can jeopardize the quantity and quality of the companies' output.

If quality control fails, it is not only inconvenient it is also costly. Repairing errors, recalling products, processing warranty claims, and dealing with all of the waste materials all add up to significant expenditures. There's also the possibility of a company's reputation being harmed and losing consumers to competition. To address these challenges, this project employs Computer Vision (CV) to automate quality inspections of PCBs, aiming to enhance efficiency, reduce costs, and improve market position. Our study explores various automation techniques, including template-based and non-template-based methods, alongside multiple Machine Learning (ML) models and tuning techniques.

We developed two novel approaches by itergrating GA(genetic algorithm) with YOLOv9 model. Our approach incorporates mutation and crossover techniques to intelligently search a vast hyperparameter space. This method is compared to manual tuning, which provided valuable insights into effective hyperparameter ranges. Ultimately, this project aims to leverage the speed of YOLO and the optimization power of a custom Genetic Algorithm, specifically tailored to address the demands of real-time PCB inspection within factory environments.

Reseach Questions
To fulfill the project goals outlined previously, the study will address the following research questions:

- How can genetic algorithms be integrated into computer vision models to optimize the detection of PCB defects?
- In order to enable "real-time" detection of PCB defects, which Computer Vision AI model should be used?

- Which AI models (ML inference) are suitably lightweight to operate on factory floor edge devices?'

Chapter 2

Literature Review

Extensive research has been dedicated to the automated examination of Printed Circuit Boards (PCBs). This research identifies four primary strategies for assessing PCBs:

1. Image segmentation using template-based detection
2. CNN Based Classification
3. R-CNN Based Detection
4. YOLO Based Detection

In addition to these strategies, there has been a significant exploration into the application of Genetic Algorithms (GAs) within the fields of Machine Learning and Computer Vision. Genetic Algorithms, inspired by the process of natural selection, are search heuristics that mimic the process of natural evolution. This approach has proven to be particularly effective in optimization and search problems within these domains. The insights gained from such research will be addressed in the following sections.

2.1 Image segmentation using template-based detection

Building upon existing methodologies for defect detection in PCBs, Vikas Chaudhary, Ishan R. Dave, and Kishor P. Upla developed a refined system for identifying PCB defects in their paper [2]. They developed a system that could sort defects into 14 distinct categories through a five-stage process: aligning the images (image registration), preparing the image for analysis (pre-processing), dividing the image into segments (image segmentation), identifying defects, and finally classifying those defects.

Their approach is designed to accurately inspect PCB images, even those that have been altered in terms of rotation, scale, or position. They achieved this through 'Image Registration,' a crucial step that adjusts the test image to match the template image by addressing any changes in rotation, size, etc. Image Registration In practical scenarios, such as on a factory assembly line, positioning discrepancies between the PCB under inspection and the template can occur. To tackle these discrepancies, 'Registration' adjusts the test image to eliminate differences in rotation, size, and position relative to the template image.

The following steps were used to register the PCB images:

1. The test PCB image and the design PCB image are turned into grayscale images.

2. Algorithms like SURF [16] are used to pull out the feature points from the two images. SURF's feature points are stable and don't change when the image is rotated, scaled, or lit up.

3. A transformation matrix is generated to make the test image fit the template image's orientation and position.

Following image registration, a technique known as median filtering was applied to clear out 'salt-and-pepper' noise, which shows up as random specks of black and white on digital images. This type of noise, occasionally seen in digital photos, disrupts the image clarity. Median filtering, a digital method not following a straight line, works by examining each pixel and updating it with the median value from its nearby pixels, effectively reducing noise. Additionally, to manage variations in brightness, a Gaussian low-pass filter was used, which helps in smoothing out the image.

After aligning the images and removing noise, the researchers suggested dividing the images into three key components to further pinpoint defects: 1.the wiring tracks, 2. soldering pads, and 3. holes. This division helps in comparing the tested PCB image against the standard template, identifying differences. These differences indicate either positive or negative defects, with positive ones identified by what's in the tested image but missing in the template, and the reverse for negative defects.

This method's strength lies in its precise defect detection through the initial image alignment and the detailed classification of defects into various types. It's also quick, taking about 2.5 seconds to inspect a PCB, making it suitable for real-time inspection in manufacturing environments. However, its accuracy drops when dealing with complex PCB designs, where image division might not be as effective.

2.2 Defect Detection with CNN-Based Classification

In their pivotal work, "HRIPCB: a challenging dataset for PCB defects detection and classification," [7]Weibo and Peng introduced an advanced method to increase the precision of spotting defects in PCBs using a template-based approach coupled with a CNN (Convolutional Neural Network). This method uses the traditional template-based technique for detecting defects but advances the classification of defects through a machine learning model CNN.

Their technique, named the RBCNN approach, deciphers as follows:

- R stands for the image Registration process, aligning images for analysis.

- B denotes the Binarisation of images, transforming grayscale images into clear-cut black and white for better contrast.

- CNN refers to the Convolutional Neural Network model used for the nuanced task of defect classification, capitalizing on its prowess in extracting and analyzing image features, a skill highly valued in computer vision tasks such as classification, segmentation, and object recognition.

In image processing, convolution involves a kernel (or filter) moving across an input image to calculate the dot product between the kernel and the image at each position.

This calculation results in a new matrix known as the feature map, which records these dot products. Kernels, which are also tools in image editing software like Photoshop for creating effects such as blurring, are crucial because they can be designed to detect specific features within an image. By applying a kernel across the entire image systematically, it's possible to identify the presence of the targeted feature anywhere in the image.

It's important to note that a convolutional layer in a neural network can utilize multiple kernels, each designed to detect different features. For example, after processing an input layer, a convolutional layer with six different filters would produce six distinct output matrices, each corresponding to a different feature detected in the input.

Additionally, a pooling layer often follows the convolutional layer. The purpose of pooling is to reduce the dimensions of the feature maps, simplifying the data by retaining only essential information and eliminating unnecessary details. This results in smaller, more manageable output matrices that are quicker to analyze. In the study by Weibo and Peng, defect detection in PCBs is executed using a comparative template method, with convolutional neural networks (CNNs) further classifying detected defects. This approach demonstrates the practical application of convolution and CNNs in identifying and classifying defects efficiently as shown in the figure below :

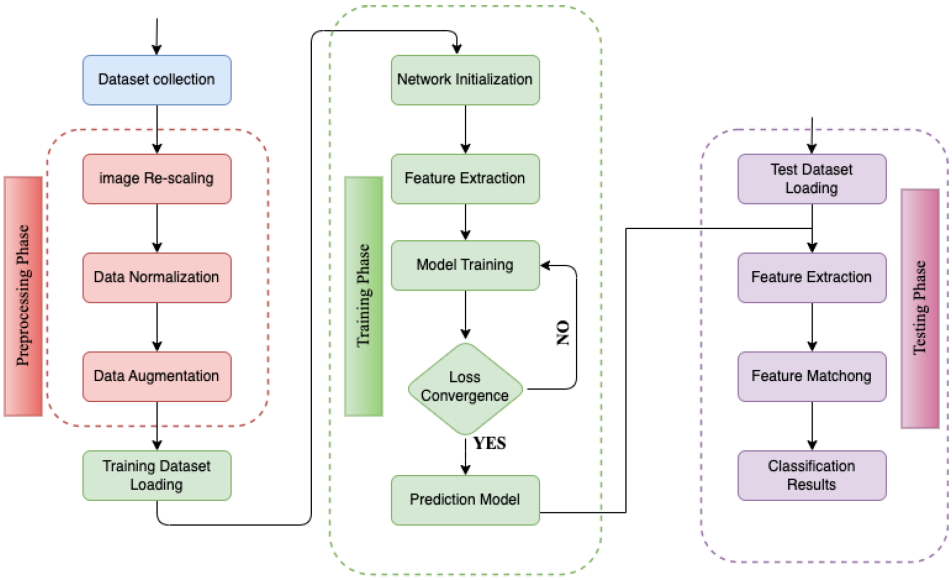

Figure 2.1: CNN Classification Flowchart (image adapted from [9])

The CNN architecture designed for classifying defects consisted of two units, each comprising six convolutional layers. This design allows every layer to use the output from all preceding layers, creating a highly interconnected structure.

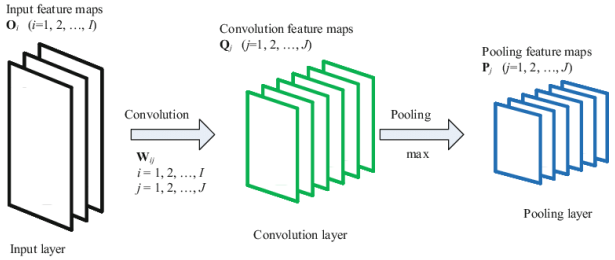

Figure 2.2: CNN Architecture (image adapted from [15])

This method achieved impressive accuracy, with average precision rates ranging between 97% and 99%. Additionally, the model could identify defects in PCBs in roughly 0.98 seconds, significantly quicker than previous methods developed by Vikas Chaudhary [2] . However, a notable limitation of this model is the substantial amount of time and computational power required for training the CNN, highlighting a trade-off between efficiency and resource consumption.

2.3 Applying R-CNN for defect detection

In addition, we examined the research conducted by Bing Hu and J. Wang, who utilized R-CNN (Region Proposal – Convolutional Neural Network) for identifying surface defects on PCBs in their paper [6]. Their approach, rooted in deep learning, proposes a novel strategy for recognizing PCB defects. The R-CNN method applies a segmentation algorithm to pinpoint specific areas within an image that might contain an object of interest. These areas are then outlined with bounding boxes and assessed by a classifier. This classifier relies on the CNN to accurately determine the nature of the objects within these areas. The use of R-CNN has significantly streamlined the feature extraction process, making it both more automated and efficient.

The employment of the R-CNN model for defect detection places this method within the "non-referential" category, highlighting its capacity to detect defects without the need for template comparison.
In every non-referential approach employing a neural network, specifically CNN, two main activities occur simultaneously:

1. Identifying defects.

2. Categorizing each detected defect.

The researchers analyzed approximately 12,000 images of defects, standardizing each to a 600x600 pixel size. They utilized an open-source tool, LabelImg, for annotating defects within these images. This involves drawing bounding boxes around the defects and assigning a specific label (or class) to each. The details of these annotations, including the locations of the bounding boxes and the class labels of the defects, were saved in XML format. For extracting features from the images, a more advanced neural network known as "ResNet50" was employed. This network aids in identifying even minor faults using a method called Feature Pyramid Networks. Dealing with deep CNNs, it's found that increasing the layers complicates training and may reduce accuracy in certain cases. This issue, known as the Vanishing/Exploding gradient problem, arises when the gradient signal weakens significantly or grows too large, slowing down the gradient descent algorithm. To address this, techniques like Residual Blocks and Skip Connections were introduced. Skip Connections allow the activation of one layer to be directly connected to another, skipping some layers in between. This setup forms what is known as a residual block. Stacking these blocks creates resnets. The ResNet50 model, which is 50 layers deep, has been recognized for its excellence in various competitions, showcasing its prowess in image detection and classification tasks. Building upon the R-CNN framework, researchers have explored methods to improve efficiency. Notably, Ren et al introduced the Region Proposal Network (RPN) in their work titled "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." [13]

The RPN leverages a fully convolutional architecture to simultaneously predict bounding boxes and associated object confidence scores. This model, trained to generate high-quality proposals, feeds detections into the subsequent Fast R-CNN stage. The key innovation lies in merging the RPN and Fast R-CNN into a unified network that shares convolutional features. This design enables a single neural network to perform both detection and classification tasks.

2.4 Defect Detection with YOLO

In 2016, Joseph Redmon and Santosh Divvala in their paper [12] introduced a groundbreaking method for object detection in images, named YOLO (You Only Look Once). This method surpassed other models like R-CNN in performance and was notably faster. R-CNN, as previously discussed, uses a system that proposes regions in an image to find possible objects, then classifies these regions, with additional steps to refine the process. These steps, however, are time-consuming and not ideal quick and fast real-time object detection, such as finding flaws in printed circuit boards (PCBs).

YOLO revolutionizes object detection by employing a regression technique. The YOLO algorithm detects objects by doing a single forward propagation across a neural network. YOLO utilizes a CNN that simultaneously predicts bounding boxes and class probabilities. Not only is YOLO incredibly quick, making it perfect for real-time detection, but its base model can analyze images at a rate of 45 frames per second. A lighter version, FAST YOLO, can even reach 155 frames per second. Understanding YOLO's effectiveness involves two metrics: Intersection over Union (IoU) and mean Average Precision (mAP).

IoU (intersection over union), is a key measure for working out localization errors and checking how accurate object identification models are at localization. The process of finding where one or more things are in a picture and drawing a bounding box around them is called object localization.

When detecting objects, the model suggests a bounding box, known as the 'Predicted Bounding Box,' around what it identifies as an object. The 'Ground Truth Bounding Box' refers to the actual or correct box that should encompass the object. The Intersection over Union (IoU) is calculated by dividing the overlap area between the Predicted and Ground Truth Bounding Boxes by their combined area.

Formula:

$$\text{IoU} = \frac{\text{Area of overlap between Predicted Bounding Box and Ground Truth}}{\text{Area of union}} \tag{2.1}$$

A higher IoU indicates the Predicted Bounding Box is closer to the Ground Truth.

The precision measure of a model is determined by taking ratio of number of correctly identified positive samples by the total number of positive samples classified, regardless of whether they were classified correctly or erroneously. The model's precision measures its ability to accurately classify a sample as positive. The recall measure of a model is calculated by dividing the number of positive samples correctly classified as positive by the total number of positive samples. The model's recall measures its ability to identify positive samples. The model's average precision per class is determined by calculating the area under the precision versus recall curve. The mean Average Precision (mAP) averages these values across all classes. The operation of the YOLO model unfolds in stages:

- Initially, it partitions the image into N square grids, each measuring SxS. These grids are tasked with identifying and identifying objects within that area.
- Each grid predicts B bounding boxes by determining their coordinates, labeling the object, and estimating the probability of the object's presence. This step often leads to multiple grids predicting the same object differently, resulting in numerous overlapping predictions.
- To refine the predictions, YOLO employs the "Non Maximal Suppression" technique, eliminating bounding boxes that have a low likelihood score. This filtering is performed repeatedly to secure the most accurate bounding boxes.

YOLO's design is rooted in the GoogleNet structure, featuring 24 convolutional layers followed by two fully connected layers. The architecture, as outlined in the academic paper, includes these layers and was initially honed on the ImageNet dataset. This extensive dataset is widely used for advancing visual object recognition software. YOLO Architecture : adapted from[3]

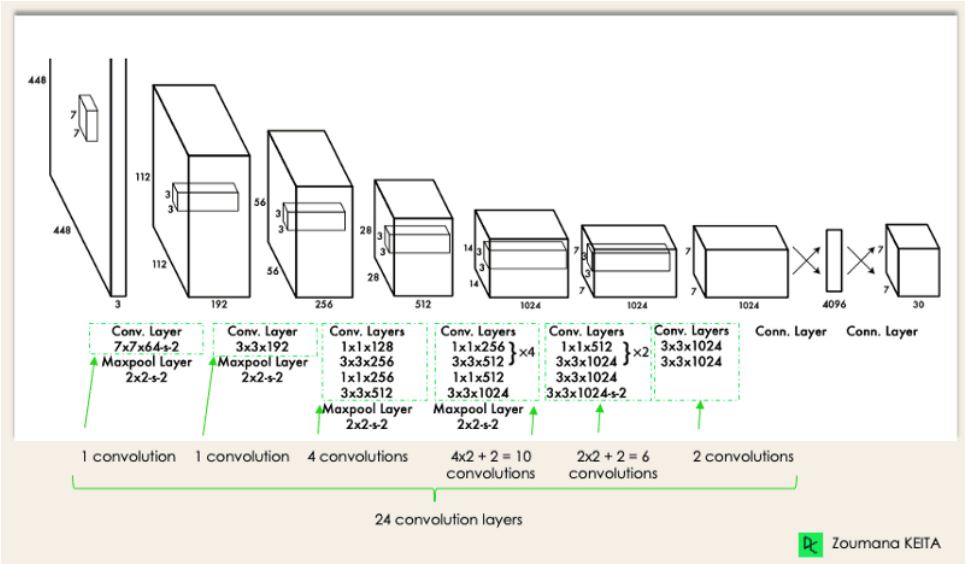

Figure 2.3: YOLO Architecture

However, YOLO encounters specific challenges:

Its grid-based detection system can identify only one object per grid, leading to difficulties in distinguishing small objects that appear closely together in groups. This makes it hard for YOLO to accurately detect and locate small items clustered together. Compared to some slower object detection methods, like Fast RCNN, YOLO shows lesser accuracy, marking a trade-off between speed and precision.

In recent years, the YOLO approach has seen continuous advancements, leading to significant improvements in both speed and accuracy. YOLOv3, released in 2018, addressed limitations of previous versions by enhancing the detection of smaller objects. This was achieved through an architecture that combined upsampled layer outputs with features from preceding layers, preserving crucial details and leading to better small object detection.

YOLOv5, introduced in 2020, built upon this foundation. It offered a set of pre-trained object detection models specifically designed for the COCO dataset (Microsoft Common

Objects in Context). This vast dataset, containing 328,000 images with annotations for various objects and individuals, is invaluable for training machine learning models in object identification and classification.

Building on the firm foundation of YOLO and its modification , recent development in depression of PCB defects yielded a new detection algorithm called PCB-YOLO based on the upgraded version YOLOv5 . It is discussed in the article by Junlong Tang and coauthors , titled "PCB-YOLO: An Improved Detection Algorithm of PCB Surface Defects Based on YOLOv5" [14]. The authors present the improved version the ensures that existing small imperfections on the PCB's surface will also be correctly identified .

The authors have ingeniously adapted and integrated several advanced technological strategies to significantly boost the accuracy and speed of defect detection. These adaptations include:

Optimization of Anchor Adjustments: Utilizing the K-means++ algorithm, the authors fine-tuned the anchor boxes specific to PCB defect characteristics, which are essential for improving detection accuracy especially for smaller or irregularly shaped defects.In refining the PCB-YOLO algorithm, the researchers employed the K-means++ algorithm for optimizing anchor boxes to enhance detection of smaller and irregularly shaped defects on PCBs. This optimization is pivotal, as it directly influences the model's ability to localize defects accurately. By iteratively recalculating cluster centers until convergence, as represented by Equation (2.4) from the source material, the algorithm ensures the generation of anchor boxes that are finely adjusted to the characteristics of the dataset. These statistical processes, grounded in rigorous mathematical foundations such as the probability calculation of selecting new cluster centers (Equation (2.2)), contribute to the nuanced improvements in the algorithm's performance

$$P(x) = \frac{D(x)^2}{\sum_{x \in \chi} D(x)^2} \tag{2.2}$$

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 \tag{2.3}$$

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x \tag{2.4}$$

Integration of a Small Target Detection Layer: This layer is particularly effective in focusing on smaller defects that previous models may have overlooked, ensuring comprehensive defect coverage across the PCB surface.

Implementation of Swin Transformer: By embedding this transformer into the backbone network, the algorithm enhances its ability to discern finer details and sub-

tle variations in the images, which are critical in distinguishing between defective and non-defective areas.

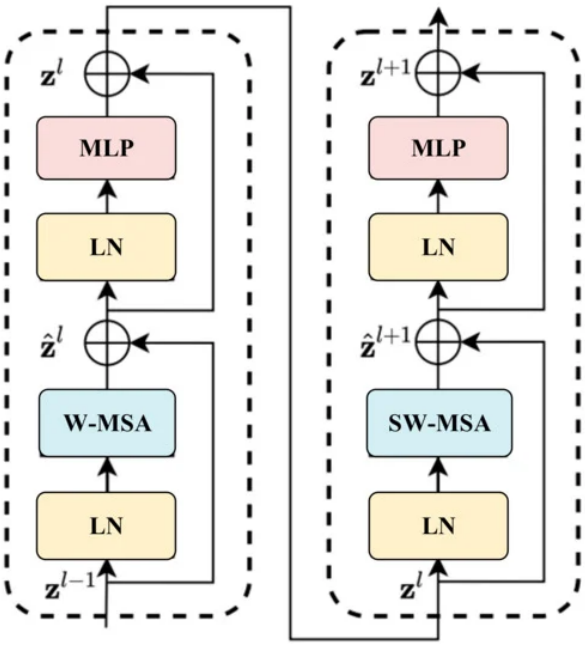

Figure 2.4: Swin transformer block structure.

United Attention Mechanism: This mechanism helps in reducing background noise and enhancing the focus on potential defects, thus improving the overall precision of the detection process. The practical outcomes of these enhancements are clearly reflected in the performance metrics of PCB-YOLO, which achieves an impressive mean Average Precision (mAP) of 90.97% and operates at a speed of 92.5 frames per second, making it suitable for real-time detection applications in PCB manufacturing.

The latest iteration of YOLO, released in February 2024, is YOLOv9. This version advances object detection with novel architectures: Programmable Gradient Information (PGI) and the Generalized Efficient Layer Aggregation Network (GELAN). PGI addresses information bottlenecks for accurate gradient updates during training, while GELAN optimizes lightweight models for superior performance across devices. These innovations position YOLOv9 to excel in real-time object detection, outperforming both convolutional and transformer-based methods. With four pre-trained models offering varying parameter counts, YOLOv9 caters to diverse computational needs. These advancements make YOLOv9 a compelling choice for a wide range of applications. However, its recent release means limited research exists on its performance specifically for PCB defect detection. This highlights the need for further exploration to determine its efficacy in this domain and its potential to surpass even YOLOv5's results.

This graph below [16]illustrates the performance comparison of various object detection models on the MS COCO dataset, which is a standard benchmark for evaluating

the accuracy of object detection algorithms. The Y-axis represents the mean Average Precision (mAP), a common metric for object detection performance, while the X-axis displays the number of parameters in millions (M), indicating the model's complexity.

From the graph, we can see that YOLOv9 outperforms other models in terms of mAP while maintaining a moderate number of parameters, showcasing its efficiency and effectiveness. GELAN also shows promising performance, suggesting that the underlying architecture of YOLOv9 is robust across different configurations.

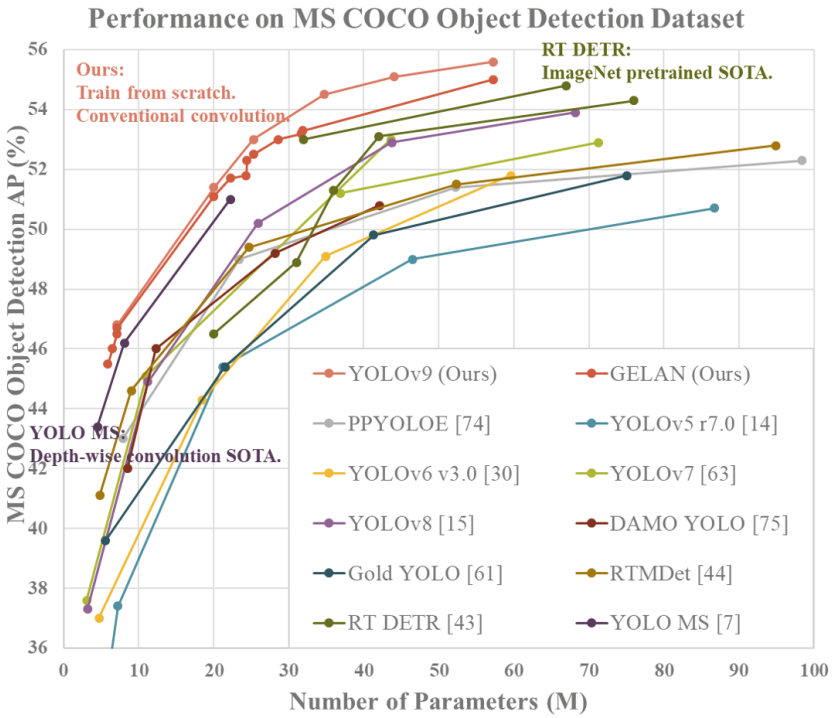

Figure 2.5: YOLO models

Model	Number of Parameters (M)	mAP (%)
YOLOv5s r7.0	7.2	37.4
YOLOv5m r7.0	21.2	45.4
YOLOv5l r7.0	46.5	49.0
YOLOv5x r7.0	86.7	50.7
YOLOv6-N	4.7	37.0
YOLOv6-S	18.5	44.3
YOLOv6-M	34.9	49.1
YOLOv6-L	59.6	51.8
YOLOv7	36.9	51.2
YOLOv7-X	71.3	52.9
YOLOv8	15	50

Table 2.1: Comparison of YOLO models based on parameter count and mAP performance.

2.5 Genetic Algorithms in Machine Learning

Hyperparameter tuning for models as complex as YOLO is a challenging task. Although grid search is frequently used, it becomes computationally expensive when dealing with search areas with high dimensions. Genetic algorithms are an intelligent alternative that uses processes of natural selection in order to continuously evolve not a living population, but a set of individuals with different hyperparameter configurations towards an optimal solution.

The article "Genetic Algorithm (GA): A Simple and Intuitive Guide" by Dana Bani-Hani [1] discusses the fundamental principles of GAs, describing how they emulate the process of natural evolution. The basics of GAs involve maintaining a population of potential solutions, referred to as chromosomes, which evolve through genetic operators like selection, crossover, and mutation. Bani-Hani elucidates how these algorithms iteratively apply these operators to evolve the population towards optimal solutions, providing a user-friendly introduction to the metaheuristic nature of GAs.

In another article "Introduction to Genetic Algorithm" published on Analytics Vidhya [8], the authors delve into the intricate workings of genetic algorithms (GAs) and their application in the realm of computer vision. The piece serves as a comprehensive introduction to GAs, elucidating their conceptual framework and practical implementations within the context of solving complex optimization problems. The genetic operators, such as selection, crossover, and mutation, are described in great detail. Based on the figures and text from the article, the genetic algorithm is explained as follows: Note: chromosomes are binary strings

1. Chromosomes and Genes: A chromosome is a string of genes, which are individual bits in a genetic algorithm context. An allele is the specific value of a gene.

Figure 2.6: Chromosomes and Gene Representation

2. Crossover: This is a reproduction process in computer vision where traits are mixed to form offspring. For instance, if two chromosomes (1 0 1 0 1 1 and 1 1 0 1 1 0)

crossover, the offspring might inherit a mix of bits from each parent (0 1 1 1 1 0), promoting diversity in solutions.

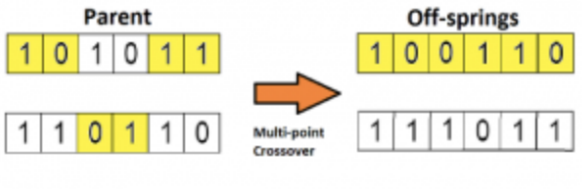

Figure 2.7: Crossover

3. Mutation: Mutation is a process of random alteration in a chromosome, introducing new traits not present in the parents. This could turn a gene from 0 to 1 or vice versa, ensuring the algorithm does not stagnate and continues exploring the solution space.

Figure 2.8: Mutation

So the full process is summarized in the figure.

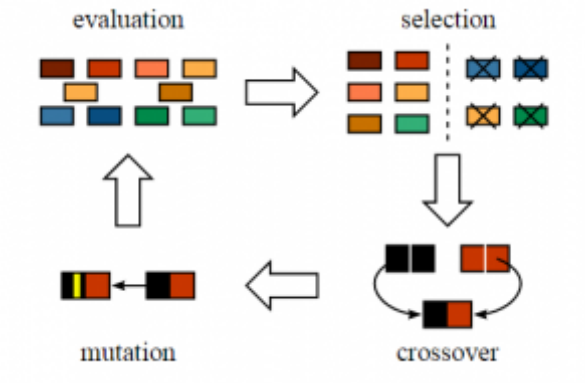

Figure 2.9: GA Working

In this paper we have proposed two Genetic Algorithm techniques(GA) for hyperparameter tuning of YOLOv9. The approach leverages concepts of crossover and mutation, inspired by biological processes, to generate potential iterations that exhibit improved performance. There are no existing articles that address the strategy of using crossover techniques in genetic algorithms (GA) for hyperparameter tuning in YOLOv9, specifically for PCB flaw identification. As a result, this research is not only novel but also crucial for advancing academic research.

These discussions across various sources not only outline the theoretical and practical aspects of GAs but also illustrate their evolving application in tackling real-world challenges, bridging the gap between theoretical optimization and practical machine learning solutions.

2.6 Operational Benchmarking of YOLO on Edge Devices

To ensure AI models work seamlessly on edge computers within manufacturing settings, the operational benchmarks were explored using YOLO on edge devices, a study led by Haogang Feng [5]. This investigation involved testing YOLO's functionality on three kinds of edge devices, often referred to as single board computers (SBC):

- NVIDIA Jetson Nano with TensorFlow GPU
- NVIDIA Jetson Xavier NX with TensorFlow GPU
- Raspberry Pi 4B equipped with an Intel Neural Compute Stick2 (NCS2).

The above list shows that the first two devices are SBCs powered by GPUs, while the last one falls under the category of an ASIC (Application Specific Integrated Circuit). The subsequent table outlines the outcomes from these tests.

TABLE II
BENCHMARKING JETSON NANO, JETSON XAVIER NX, AND RPI WITH NCS2. (NOTE: VIDEO1 HAS 1596 FRAMES WITHE A FRAME SIZE OF 768 × 436, VIDEO2 HAS 960 FRAMES WITH A FRAME SIZE OF 1920 × 1080.)

	Models	Accelerator -based SBCs	Mean Confidence(%)	FPS	CPU Usage %)	Memory Usage (GB)	Power (W)	Time (s)
Video1	YOLOv3	RPI+NCS2	99.3	2.5	4.3	0.33	6.0	690
		Nano	99.7	1.7	26.5	1.21	7.9	967
		NX	99.7	6.1	22.5	1.51	15.2	256
	YOLOv3-tiny	RPI+NCS2	0	18.8	15.5	0.11	6.5	121
		Nano	57.9	6.8	28.8	1.00	7.2	236
		NX	57.9	41.1	30.5	1.33	13.5	46
Video2	YOLOv3	RPI+NCS2	85.8	2.5	9.8	0.41	6.2	496
		Nano	71.5	1.6	28.8	1.36	8.0	587
		NX	71.5	5.9	26.8	1.69	15.2	162
	YOLOv3-tiny	RPI+NCS2	61.5	19.0	24.8	0.18	6.8	162
		Nano	54.1	6.6	37.3	1.16	7.4	152
		NX	54.1	35.6	55.5	1.47	13.2	31

Figure 2.10: Performance Benchmarking of YOLO Models on GPU and ASIC-Based Edge Devices

From the data presented, it's evident that edge devices powered by ASICs lead in performance, both in precision and CPU resource efficiency. However, integrating the AI model with ASIC's specific framework presents a significant hurdle. Moreover, as the

model's complexity increases, updating the corresponding ASIC hardware might present growing challenges. In the context of GPU-powered edge devices, selecting an appropriate model necessitates a balance between the speed of drawing conclusions (inference speed) and the accuracy of these conclusions (inference accuracy).

Comparative analysis of different methodologies Presented below is a table that clearly summarizes the comprehensive examination of literature by comparing the several established methodologies.

Approach	Lightweight (Can run on Edge)	Fast (Real time object detection)	High Accuracy	Multiple defect detection
Image segmentation using template-based detection	✗	✓	✗	✓
CNN Based Classification	✗	✓	✓	✓
R-CNN Based Detection	✗	✓	✓	✓
YOLO Based Detection	✓	✓	✓ (Fairly accurate)	✓ (May not be accurate if trained for large number of classes)

Figure 2.11: Comparison of various approaches

As seen from the information provided, YOLO exhibits significant potential when additional study is conducted to tailor the model for PCB flaw identification. This study work aims to verify the capability of YOLOv9 model to be trained for PCB detection with enhanced accuracy and the ability to identify numerous flaws.

Chapter 3

Data Handling

This chapter covers the selection and preparation of dataset for defect detection in PCBs and sets up the model training framework. It details why this dataset was chosen, outlines the pre-processing and augmentation techniques used to optimize it for training, and discusses the setup for training an effective machine learning model.

3.1 Data Preparation and Enhancement

3.1.1 Dataset

In order to choose the best dataset for this PCB defect detection project, a comprehensive evaluation of numerous open source data sets was done. The dataset for PCB defects, provided on Kaggle, emerged as the most suitable choice. Comprising 1386 meticulously collected images, this dataset covers six crucial defect types essential for quality control in electronics manufacturing: missing holes, mouse bites, open circuits, shorts, spurs, and spurious copper.

The image below depicts the six types of flaws present in the dataset.

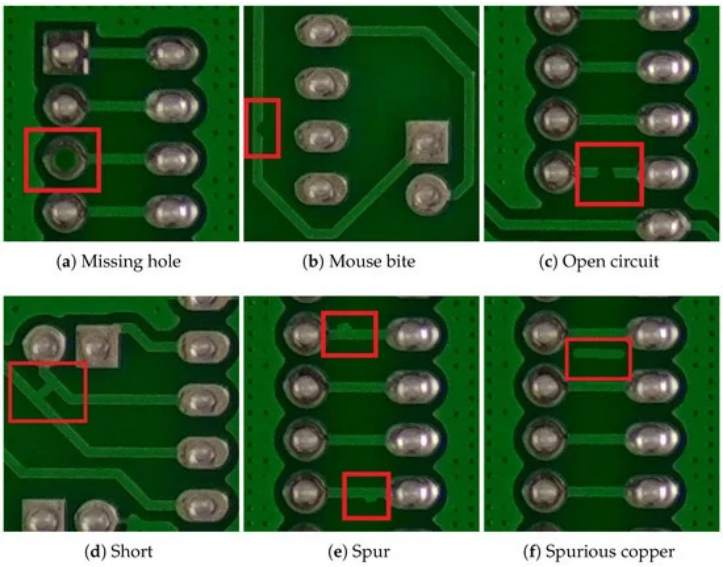

Figure 3.1: Defects in PCBs (adapted from [4])

All the types of defects above occurred during the plating and etching process of the PCB manufacturing which led to excess or absence of copper on the PCB plate. Excessive Deposits: Unintended deposition of conductive materials like copper can lead to faults such as shorts, spurs, and spurious copper.

- **Shorts:** These occur when two traces that shouldn't be connected end up connected, disrupting proper circuit function.
- **Spurs:** These are rigid conductor projections that can cause short circuits if they contact other PCB elements.
- **Spurious Copper:** This is copper that appears where it wasn't intended, increasing the likelihood of electrical failure.

Faulty Tooling: Issues in the tooling assembly can lead to excessive processing and the following defects:

- **Missing Holes:** Places on the PCB where holes should have been drilled but weren't.
- **Mouse Bites:** Depressions along a trace that can cause unreliable electrical connections and lead to open circuits.

Note: Due to its large size, this dataset can be downloaded from the source [11] and is not included in the code upload. The following graph displays the frequency of flaws among printed circuit boards (PCBs). A prominent pattern emerges, indicating that the majority of PCBs have either three or five faults.

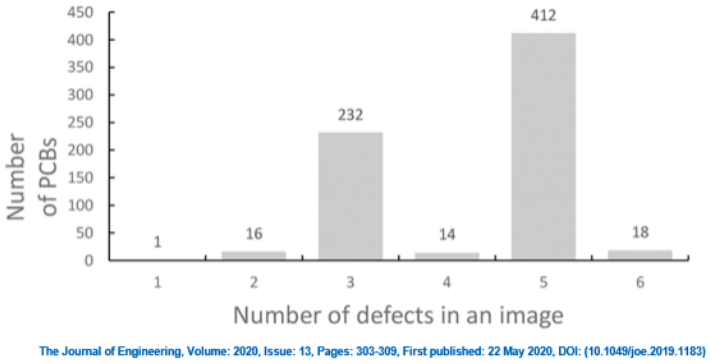

Figure 3.2: Frequency of flaws among PCBs

It is important to note that this projects analysis is limited to the six identified forms of defects in PCBs. There are many other flaws that can happen during PCB manufacture, however they are not within the scope of this research.

The HRIPCB database was chosen for its broad coverage of common PCB flaws, high-resolution image quality, and PCB diversity. These parameters are essential for training a robust model that can recognize and categorize real-world problems. The dataset provided the best foundation for this study because it could closely match practical application issues.

3.1.2 Data Pre-processing

Pre-processing is an essential initial step in the field of machine learning, as it prepares the dataset to achieve the highest level of efficiency in training the model. By refining the data before the training phase, we can significantly reduce training duration and improve the speed of the model's inference. For this project, pre-processing involves several techniques for which it leverages RoboFlow's capabilities to streamline the pre-processing tasks efficiently. RoboFlow is a tool used for handling and processing image data. The specific techniques employed include:

- **Static and Dynamic Crop:** These techniques adjust the framing of images to focus on key areas, helping the model to better understand the relevant features without the distraction of background noise.
- **Grayscale Conversion:** Transforming images to grayscale, reduces computational complexity, allowing the model to concentrate on structural and shape-related features rather than colour variations, which are not essential for defect detection in PCBs.
- **Auto-Adjust Contrast:** Improving the contrast of images can make the defects more pronounced and distinguishable, aiding in the model's accuracy during detection tasks.
- **Isolate Objects:** This method segregates the defects from the surrounding PCB area, enabling the model to learn from a clearer representation of the target objects.

- **Tiling:** Breaking down images into tiles can help in handling large images more effectively and also aid in focusing on local features within each tile.

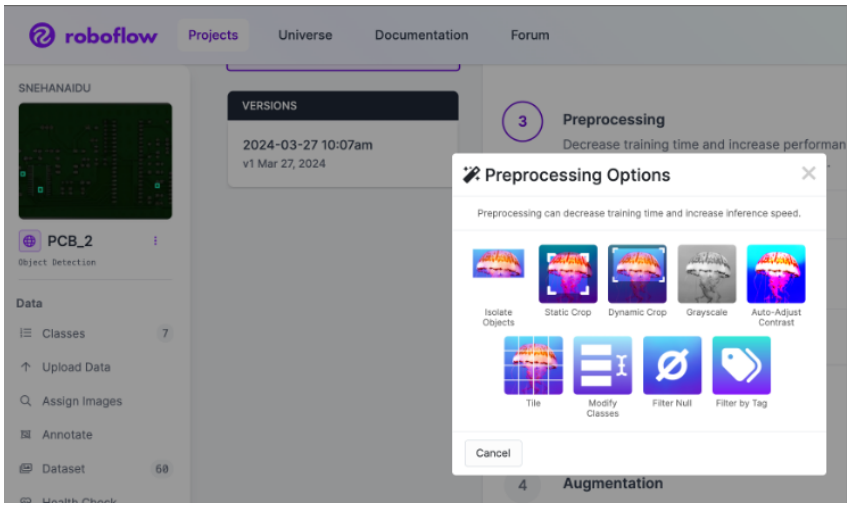

Figure 3.3: Data Pre-Processing Techniques

RoboFlow's pre-processing steps ensure the data fed into the YOLOv9 model is high-quality and conducive to learning. The model can thus be trained more effectively, setting a strong foundation for robust performance during the detection of PCB defects.

3.1.3 Data Augmentation

The relationship between training data quantity and deep learning model accuracy is well-established: more data often leads to more accurate predictions. This correlation poses a significant challenge, particularly in industries and research areas where extensive data is scarce. This project is no exception; the original dataset contained a finite number of PCB images, which necessitates the expansion of data.

Data augmentation offers a powerful approach to address the challenge of limited training data. This technique involves modifying existing data or generating synthetic samples to create a larger and more diverse training set. Traditional image augmentation methods, such as cropping, flipping, and rotating, can significantly increase the volume and variation of the data without requiring additional data collection.

Inspiration for more sophisticated augmentation strategies was drawn from Ayush Thakur's "Modern Data Augmentation Techniques for Computer Vision" [2]. A notable technique from his work is cutout augmentation, a deceptively simple yet effective method. This technique enhances data variability by erasing patches of the image and filling them with uniform pixels or noise.

Theoretical insights were derived from literature, and the practical application of augmentation strategies was performed using the RoboFlow platform, which offered a comprehensive set of tools tailored to the specific needs in image and bounding box augmentation. To achieve a higher degree of model accuracy, a series of data augmentation techniques were applied through RoboFlow. The image provided below illustrates the augmentation options utilized:

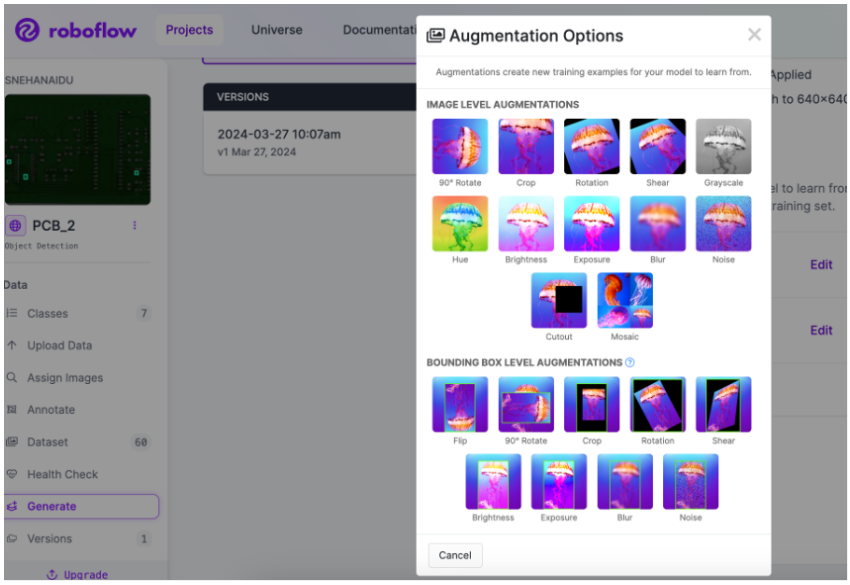

Figure 3.4: Data Augmentation Techniques

- Image Level Augmentations:
- Rotations (including 90° rotate)
- Crops
- Shear transformations
- Adjustments to grayscale for simulating different lighting conditions
- Hue, brightness, and exposure alterations to replicate different environmental settings
- Blur and noise filters to mimic camera and sensor imperfections
- Cutout and mosaic augmentations for object occlusion and environment variance
- Bounding Box Level Augmentations:
- Flipping and rotating bounding boxes to maintain accuracy after image manipulations
- Exposure and brightness adjustments on the object level to teach the model to identify defects under various lighting conditions.

The inclusion of these augmentation techniques is poised to not only diversify the dataset but also to train the model to recognize defects across a spectrum of variable conditions, ensuring robust detection and classification. The thoughtful application of these augmentation techniques not only broadens the dataset but also equips the model with the experience of a wide range of conditions, paving the way for the next crucial stage of the project: training the model for precise and resilient performance. Before we delve into the specifics of model training, it is important to address the ethical framework and bias mitigation strategies that underpin this research.

3.1.4 Ethical Considerations in Object Detection Research

This research utilizes a publicly accessible and synthesized dataset for object detection, thus negating the ethical concerns associated with surveillance and defense applications. With no confidential designs or sensitive business information involved, and with rigorous citation practices in place, the ethical implications typically associated with AI research are substantially minimized in this study.

3.1.5 Mitigating Bias in Dataset

Ensuring a balanced representation among the defect types in the PCB dataset was a priority to prevent bias in the resulting AI model. The use of RoboFlow for data augmentation helped maintain this balance. Since the dataset is centered on non-personal, inanimate objects, the common biases related to human demographics are not relevant in this research. With these ethical considerations and bias prevention strategies set in place, this project can proceed with confidence to the next phase.

3.2 Model Selection and Initial Setup

Upon careful consideration of advanced object detection models as discussed in the Literature Review, YOLOv9 was selected for its state-of-the-art capabilities. The YOLOv9 family presents a range of models, each with its trade-offs between speed and accuracy. The selection of YOLOv9-C ('Compact') was based on its ideal balance between accuracy and computational efficiency. Before selecting YOLOv9-C, the full range of YOLOv9 models was carefully considered

Model Comparison:

- YOLOv9-T (Tiny): Prioritizes speed, ideal for real-time scenarios with strict computational limits. May sacrifice accuracy for complex PCB defect patterns.
- YOLOv9-S (Small): Slightly improved accuracy over Tiny, still very fast. A good fit if detection precision is important but speed remains a high priority.
- YOLOv9-M (Medium): Further enhances accuracy, suitable for less time-critical scenarios. May be too computationally demanding for some edge servers.
- YOLOv9-E (Enterprise): Offers the highest accuracy, but with significant resource overhead. Best for powerful server environments where precision is paramount.
- YOLOv9-C ('Compact') was selected for its ideal balance between accuracy and computational efficiency. It is well-suited for PCB defect detection, as its lightweight nature facilitates deployment on edge servers. This deployment strategy significantly reduces computational demands while maintaining a high level of detection performance.

The YOLOv9-C model is architecturally complex and robust, comprising 618 layers and approximately 25,590,912 parameters, yet it requires no gradients during operation, which simplifies the inference process. This model operates at an impressive 104.0 GFLOPs, making it highly effective for real-time processing tasks. The 'Compact' version of YOLOv9 is specifically designed to provide a substantial detection capability while

managing to be lightweight enough for applications requiring quick deployment on less powerful devices or edge servers.

The complexities of the YOLOv9 architecture lie in its ability to process extensive amounts of data rapidly through its deep neural networks. This model is built on a convolutional neural network (CNN) framework, which is highly efficient in feature detection and localization - a crucial aspect for accurate defect identification in PCBs. By utilizing convolutional layers, YOLOv9 effectively captures the spatial hierarchies in images, allowing the model to detect nuanced anomalies and defects in PCBs that other models might overlook.

3.2.1 GPU Configuration and Deployment

We utilized Google Colab due to its scalable environment and GPU support, specifically taking advantage of the high performance of Tesla GPUs to effectively manage the computational demands of the complex YOLOv9 architecture.The YOLOv9-C model was trained and deployed using Tesla T4 GPUs, selected for its computational power and efficiency which match the demands of deep learning tasks.

3.2.2 Dataset Preparation and Configuration

Before delving into the actual training, it's crucial to understand the role of the yaml file. This configuration file serves as the blueprint for the training process, detailing the paths to the training and validation datasets, and defining the classes that the model will identify and learn from. It ensures that the model is exposed to the correct data and understands the categories it is expected to detect.

The first step involved customizing the dataset paths within the data.yaml file to point to specific directories. The train, val, and test paths were adjusted to ensure that YOLO would correctly locate and utilize the image datasets for training, validation, and testing phases, respectively. Given the unique requirements of the project, it was necessary to revisit the class definitions section. The nc (number of classes) value was updated to reflect the exact number of defect types the model needed to identify. Correspondingly, the names array was revised to list all the defect categories accurately, ensuring alignment with the dataset annotations.

Example of Revised data.yaml:

Notebook data.yaml ×

```
1 train: ../train/images
2 val: ../valid/images
3 test: ../test/images
4
5 nc: 6
6 # names: ['0', '1', '2', '3', '4', '5']
7 names: ['missing_hole', 'mouse_bite', 'open_circuit', 'short', 'spur', 'spurious_copper']
8
9 roboflow:
10   workspace: snehanaidu-6jbxy
11   project: pcb_2
12   version: 1
13   license: CC BY 4.0
14   url: https://universe.roboflow.com/snehanaidu-6jbxy/pcb_2/dataset/1
```

Figure 3.5: data.yaml file

In conclusion, Chapter 3 has effectively laid the groundwork for the commencement of model training by meticulously detailing the selection and preparation of the dataset specifically curated for defect detection in PCBs. Through comprehensive data preparation, advanced preprocessing techniques, and innovative data augmentation strategies, the dataset is now optimized to support the training of a machine learning model that is both efficient and precise. With the YOLOv9 model selected and the training framework established, we are well-positioned to begin the crucial phase of training the model, which promises to enhance defect detection capabilities significantly.

Chapter 4

Model Training

This section outlines Approach 1 for model training, detailing the design, experimentation, and results of manually tuning the 'epochs' hyperparameter for PCB defect detection.

APPROACH 1

4.1 Baseline Model

4.1.1 Manual Tuning of the 'Epochs hyperparameter

To create a foundational benchmark for the PCB defect detection task, we initiated the process by training the model, which considers the number of epochs as a key parameter. An epoch represents one complete pass of the model over the entire training dataset. During each epoch, the model's weights and biases were iteratively refined to improve its ability to identify defect patterns.

Given the importance of precise defect detection, Mean Average Precision (mAP) was selected as the primary metric to evaluate our model's performance. It represents the average of the Average Precision (AP) for each class within a dataset.The AP for a class is calculated by integrating the area under the precision-recall curve, which is generated by plotting precision against recall at various threshold levels of detection confidence.The mAP metric is often reported at specific Intersection over Union (IoU) thresholds to show how accurate the model's predicted bounding boxes are.

IoU measures the degree of overlap between a predicted bounding box and the ground

truth (true) bounding box for an object.

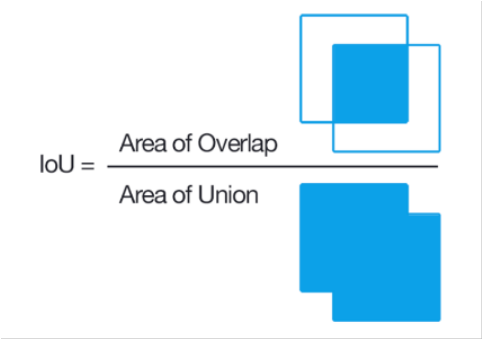

Figure 4.1:

This metric is essential as it encapsulates both precision and recall, offering a comprehensive assessment of detection accuracy. Specifically, there are variations of mAP: mAP50 and mAP50-95.

- **mAP50:** This metric evaluates the average precision of a model's object detections, focusing only on bounding boxes with an overlap greater than 50% compared to the ground truth (actual object location). It emphasizes highly accurate detections.
- **mAP50-95:** This metric provides a broader assessment. It considers all detections and assigns weights based on their overlap with the ground truth. It calculates the average precision across various overlap thresholds, typically ranging from 50% to 95%. It reflects the model's performance across a wider range of detection qualities.

In simpler terms, mAP50 focuses on the most accurate detections, while mAP50-95 offers a more comprehensive view of the model's overall detection ability. Our goal is to maximize the mean Average Precison(mAP) score during the hyperparameter tuning process to enhance the accuracy of defect detecion.

In our experimentation, we aimed to determine the optimal number of epochs to maximize mAP

4.1.2 Experimentation and Results

Experimentation with Epochs: We investigated the impact of the number of epochs on model accuracy. This was crucial for striking a balance between underfitting (too few epochs, leading to poor performance) and overfitting (too many epochs causing the model to become overly specialized to the training data and generalize poorly to new images).

Our experiments yielded the following results:

Optimal Epochs: Through experimentation, we determined that the optimal number of epochs for this model and dataset was 91 as seen in figure below.

Table 4.1: Epoch Experimentation and mAP Results

Number of Epochs	**Maximum mAP Achieved (%)**	**Observations**
20	45	Underfitting
50	60	Improved Accuracy
91	89	Optimal
100	Declined after 91st	Overfitting observed

epoch	metrics/mAP50(B)	metrics/mAP50-95(B)
90	0.89347	0.46149
91	0.89789	0.47569
92	0.83316	0.44474
93	0.83283	0.44157
94	0.84182	0.44505
95	0.84118	0.45267
96	0.84094	0.44036
97	0.83182	0.44012
98	0.83182	0.44004
99	0.84094	0.44232
100	0.84089	0.44217

Figure 4.2: mAP scores for number of epochs

The reason for choosing mAP50 as the performance metric is : PCBs are complex and dense with components, where even tiny flaws can lead to big problems. So, our focus is on being as accurate as possible. Using mAP50, we are setting a high bar for our system—it must be spot-on over half the time with its defect spotting. This way, we can trust that when our system flags a defect, it's very likely to be right, which is crucial for maintaining the integrity of the PCBs.

Our model training and evaluation process generated several informative outputs. Here's an analysis of the most significant ones:

PCB with Defects (bounding boxes)

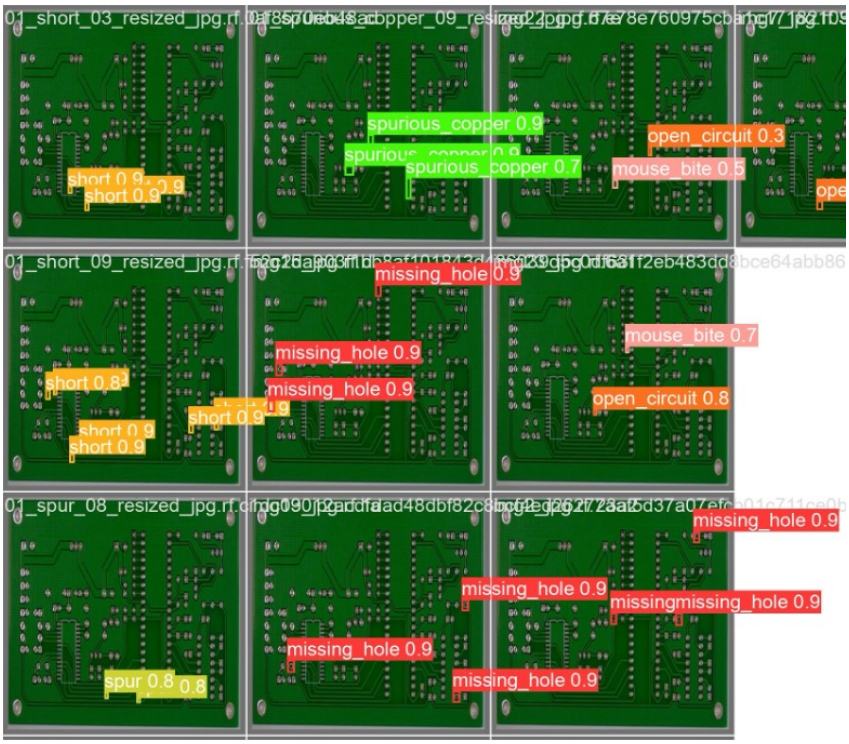

Figure 4.3: PCB with Defects (bounding boxes)

The image displays multiple PCBs, each with various defects highlighted by bounding boxes. The labels within each bounding box directly indicate the specific defect type and the number beside it specifies the Confidence Score.

- Missing Hole (multiple instances with confidence scores near 0.9)
- Mouse Bite (multiple instances with confidence scores ranging from 0.5 to 0.7)
- Open Circuit (with a lower confidence score of 0.3)
- Short (multiple instances with consistently high confidence scores of 0.8 or above)
- Spur (multiple instances with confidence scores around 0.8)
- Spurious Copper (with a confidence score of 0.9)

The confidence score (ranging from 0 to 1) represents the model's certainty in its defect classification. A score closer to 1 signifies high confidence. In this image analysis, the model demonstrates strong performance by accurately identifying most defect types with high confidence scores. Lower scores, such as the one for the open circuit defect, indicate areas for potential improvement. We will further refine the model's accuracy by employing hyperparameter tuning using genetic algorithms (GA).

Confusion Matrix In our confusion matrix the diagonal values highlight how well our model performs on each specific defect type, taking into account their frequency in the dataset. For instance, a high diagonal value for "Missing Hole" suggests our model is proficient at detecting this defect. Low diagonal values pinpoint specific defect classes where our model performs less reliably. We have analysed two types of confusion matrices:

- **Normalized Confusion Matrix:** This matrix reveals how well our model performs on each defect class relative to its frequency in the dataset, highlighting potential weaknesses even on less common defects. From the figure we can see that
- **Raw Count Confusion Matrix:** This matrix shows the absolute number of correct and incorrect predictions, pinpointing the most frequent errors overall, regardless of specific defect classes.

Insights from the Normalized Confusion Matrix (Figure 3.9):

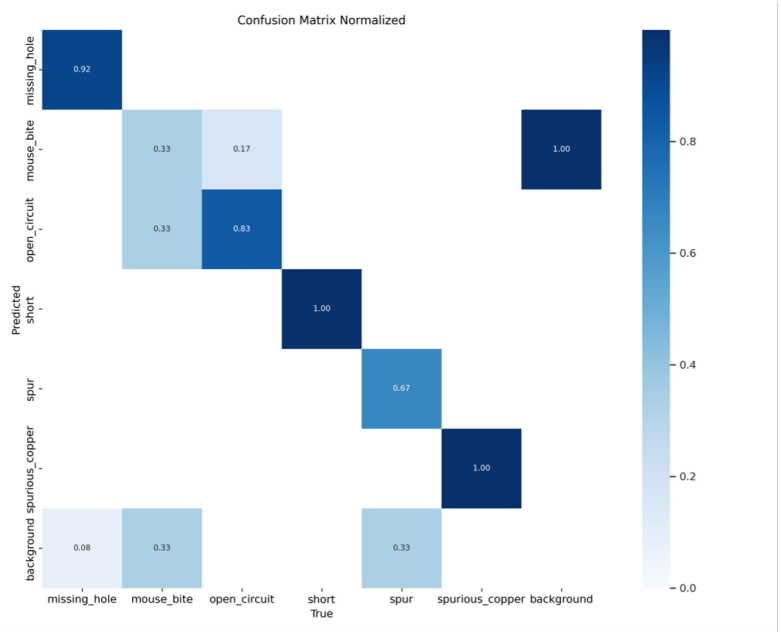

Figure 4.4: Normalized Confusion Matrix

- Accuracy by Class: The "Missing Hole" class has a high diagonal value of 0.92, indicating our model detects this defect type with 92% accuracy when it occurs.
- Challenges: The "Spur" class has a lower diagonal value of 0.67, suggesting our model is less accurate in recognizing this defect and there is room for improvement.
- Specific Misclassifications: The confusion between "Mouse bite" and "Open circuit" defects is indicated by an off-diagonal value of 0.33, which indicates the model confuses these two defect types.

Insights from the Raw Count Confusion Matrix (Figure 3.10):

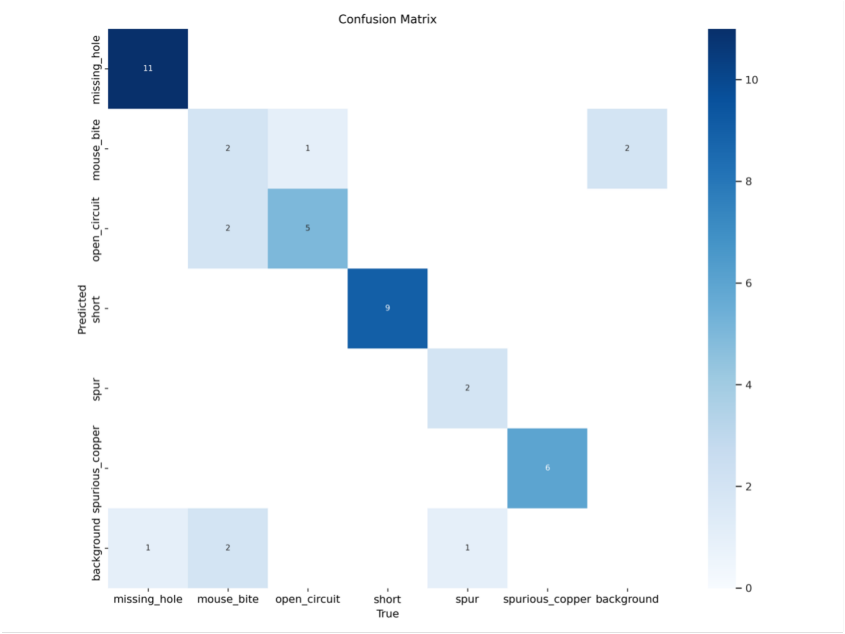

Figure 4.5:

- Most Frequent Correct Predictions: The highest count on the diagonal is 11 for the "Missing Hole" class , indicating that this defect type has the most number of correct predictions in the dataset.
- Additional Errors: The model also shows misclassifications with counts of 2 in places, such as between "Mouse Bite" and "Open Circuit", which suggests these are areas where the model may confuse one defect for another.

Performance Metrics Graphs: These graphs track loss, precision, recall, and mAP over training epochs.

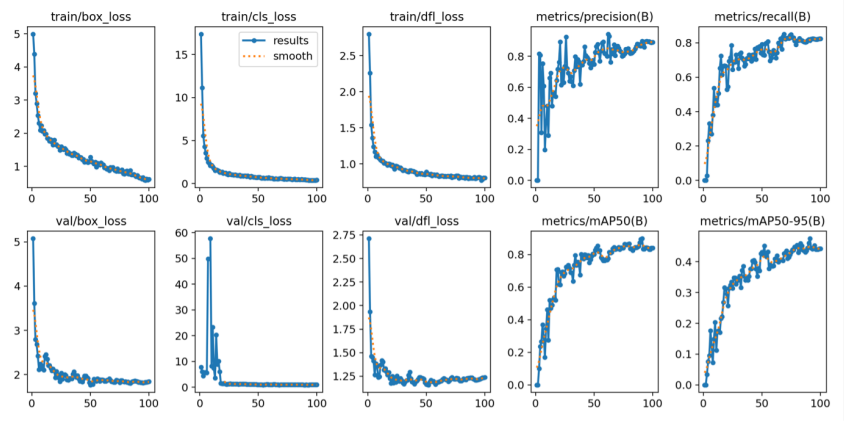

Figure 4.6: Performance Metrics Graphs

- Loss Trend :We observed a downward trend in the loss function over training epochs (left-side plots). This suggests that the model is progressively learning to map the features extracted from PCB images to the corresponding defect labels. This is an encouraging sign of successful model training.
- Precision/Recall Fluctuations: While the loss function indicates learning, the precision and recall plots (right side) show fluctuations. It's important to monitor the validation set metrics (blue lines) closely. Significant drops in validation precision or recall, while the training set metrics (orange lines) remain stable, could indicate overfitting. We will perform advanced hyperparameter tuning further to address this issue and ensure the model generalizes well to unseen PCB images.

Precision-Recall Curve:

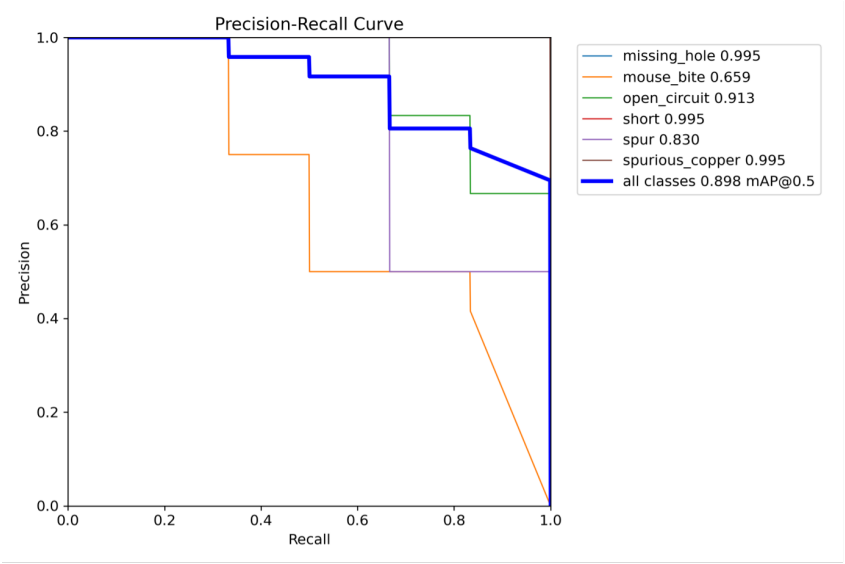

Figure 4.7:

- The Precision-Recall Curve graph is used to evaluate the performance of a classification model at different probability thresholds. The precision (y-axis) is the ratio of true positive predictions to the total number of positive predictions (true positives + false positives), and recall (x-axis) is the ratio of true positive predictions to the total actual positives (true positives + false negatives).
- For the "Missing Hole" and "Short" classes, the precision is very high (0.995) across all recall levels, which indicates excellent model performance for these defect types.
- The "Open Circuit" class also has high precision (0.927), suggesting that when the model predicts this class, it is correct most of the time.
- The "Mouse Bite" class has lower precision (0.733), indicating that the model has more false positives for this class, and the precision for "Spur" is the lowest at

0.693, showing that it is the most challenging defect type for the model to predict correctly.

- The "Spurious Copper" class has a very high precision (0.995), indicating reliable predictions for this defect type.
- The mean average precision (mAP) at an Intersection over Union (IoU) threshold of 0.5 is 0.898 for all classes combined. This is a summary metric that combines precision and recall across all classes, suggesting an overall high performance of the model.

Experimenting with epochs revealed the best epoch is the 91st epoch with a mAP of 89% . Analyzing loss functions, precision/recall, and confusion matrices provided valuable informaton. We observed strong performance for common defects, particularly achieving precision in the mid-to-high 90s for classes like "Missing Hole" and "Short." However, some defect classes like "Spur" and "Mouse Bite" showed room for improvement, with precision potentially hovering in the mid-70s.

While manual experimentation provided valuable insights, we plan to further explore the use of a Genetic Algorithm (GA) for hyperparameter tuning. GAs can automate the search for optimal hyperparameter values, including the number of epochs, learning rate, and other factors affecting model performance.

Chapter 5

Hyperparameter Tuning

This section concludes with a comparison of different hyperparameter tuning methods for PCB defect detection. It evaluates the baseline model, mutation-based tuning, and crossover tuning.

5.1 Genetic Algorithm for Hyperparameter Tuning

In order to increase our models performance we studied advanced hyperparameter tuning techniques along with GA. While straightforward, "Grid Search" quickly becomes computationally expensive for complex models with numerous hyperparameters. "Bayesian" methods offer efficiency but may struggle in high-dimensional search spaces like those encountered in YOLO models.

YOLO models rely on numerous hyperparameters that influence their performance. These include:

- Learning Rate (lr0, lrf): Controls the step size for updating the model's weights.
- Momentum: Optimizes training speed and stability.
- Weight Decay: Helps prevent overfitting.
- Data Augmentation (degrees, translate, scale, etc.): Artificially expands the training dataset for better generalization. etc.

Inspired by how GA works, we decided to implement it for hyperparameter tuning. Similar to the principle of 'survival of the fittest' observed in nature, Genetic Algorithms (GA) choose the most efficient solutions from a large set of options. They iteratively evaluate and refine sets of hyperparameter values, aiming to discover configurations that improve model performance. We have implemented two GA techniques:

- Mutation:The mechanism in mutation works by introducing variability into the pool of hyperparameters. In mutation hyperparameters are tweaked in each itteration aiming to the best set of hyperparameters.
- Crossover: Inspired by how genes combine during reproduction, in crossover the best performing hyperparameter sets are merged, forming a new set. We have developed a custom crossover mechanism from scratch.

APPROACH 2

5.1.1 Hyperparameter Tuning using Mutation :

We have employed the Mutation approach for hyperparameter tuning. This method selectively updates hyperparameters, keeping only those changes that improve the model's performance. With each iteration, the algorithm makes these minor changes, learning which tweaks result in improved accuracy and gradually updating the hyperparameters.

The tuning process which takes two key parameters: Iterations and Epochs

- **Iteration:** Each iteration is a cycle where a set of hyperparameters is adjusted, the model is trained, and its performance is evaluated. In our case, we used 10 iterations, generating 10 distinct sets of hyperparameters.
- **Epoch:** Within each iteration, an epoch is one complete pass of the model over the entire training dataset. We set the epoch count to 100 based on our prior manual tuning, which revealed a good balance between training and avoiding overfitting.

Experimentation and Results

We chose 10 iterations as a trade-off between thorough exploration and computational cost. Each iteration generates a distinct set of hyperparameter values, so we have 10 sets. Since we set the number of epochs to 100 within each iteration, each of our 10 hyperparameter sets undergoes 100 epochs of training. Increasing the iterations would significantly increase computational demands due to the additional sets and their associated training time.

Process

1. **Initial Hyperparameter Population:** We begin by defining a set of hyperparameters for the machine learning model. In our case, we identified 23 key hyperparameters that could potentially influence the model's performance. This initial set represents the starting point for our exploration.
2. **Mutation: Exploring the Hyperparameter Landscape** We apply mutate function to introduce small, random changes to the hyperparameters set to produce a new set.
3. **Model Training :** Each newly generated set of hyperparameters (after mutation) is used to train the model for a predefined number of epochs (iterations through the training data). In our case, we set the training duration to 100 epochs for each set.

4. **Fitness Evaluation:** After training, we need to assess how well each set performed. This is where the fitness function comes into play. We designed a fitness function that combines three important metrics relevant to our task: precision (p), recall (r), and mean Average Precision (mAP).

   • Precision tells us how often the model correctly identified a positive case. (measure of accuracy for positive predictions)

   • Recall indicates how many actual positive cases the model didn't miss. It reflects how well the model captures all the relevant positive cases.

   • mAP is a broader measure that considers precision at different thresholds, providing a more comprehensive picture of the model's performance.

   We assigned weights to each metric in the fitness function based on the specific priorities of our task. In our case the mAP had the highest weightage. Essentially the fitness function evaluates and gives a score based on how well the model has performed on the validation set. A higher score indicates better overall performance for that particular set.

5. **Iteration and Selection:** The process of mutation, training, and fitness evaluation is repeated for a predefined number of iterations (generations). In our case, we chose 10 iterations as a balance between thorough exploration and computational cost.

   With each iteration, the GA prioritizes sets with higher fitness scores (better performing models) for creating the next generation's hyperparameter sets. This ensures the GA focuses on areas of the hyperparameter space that lead to better performance. The above process is repeted for each and every set in our case 10 sets (iteration is set to 10).

   Through this iterative process, the GA gradually converges towards the optimal set of hyperparameters for the specific model and task.

mAP50 Metric Trends and Observations To evaluate the effectiveness of our tuning process, we carefully analyzed the mAP metric outputs.

Trends Observed:

- Initially the scores seem to vary quite significantly in the initial iterations (0 to 3), with a peak at iteration 2.
- After iteration 3, the mAP scores are consistently higher than the first three iterations, suggesting that the tuning is leading to an improvement in the model's performance.
- From the sixth iteration onwards, the mAP scores achieve a plateau, indicating that the mutation process has arrived at a highly effective set of hyperparameters. The score remains constant at 0.56077 from iterations 6 through 10, suggesting that by the sixth mutation, we had reached the best hyperparameter configuration within the constraints of our tuning algorithm

Mutation	mAP50
1	0.44118
2	0.44118
3	0.55015
4	0.55015
5	0.55015
6	0.56077
7	0.56077
8	0.56077
9	0.56077
10	0.56077

Figure 5.1: mAP Score

The consistent increase in mAP scores after the third iteration indicates that the mutation method is successfully aiding the model's learning and enhancement. The positive trend confirms that the algorithm is working as intended, consistently improving the model's ability to make accurate predictions. However, despite these advancements, the model has yet to surpass the high benchmark set by the baseline model, which boasts an mAP of 89%. . This suggests that although there is some improvement, there is still a significant difference to overcome in order to reach the highest level of performance as compared to the standard.

Given that we have only completed ten iterations so far, it is fair to assume that conducting more iterations could potentially result in better mAP scores.

Graph Analysis for Hyperparameter Tuning using Mutation:

Fitness vs Itteration Graph:

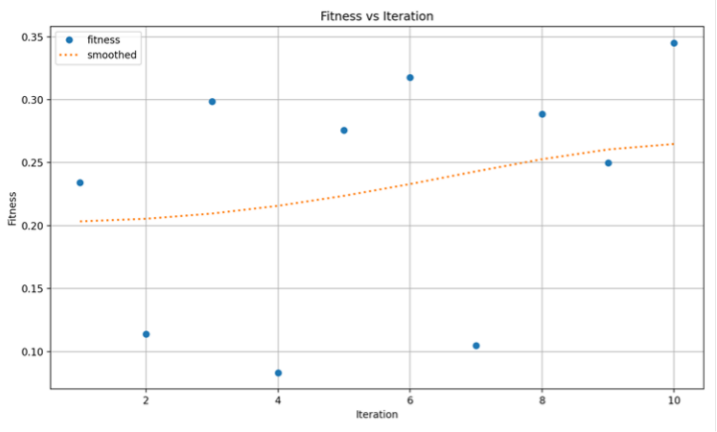

Figure 5.2:

The "Fitness vs. Iteration" graph elucidates the progression of our model's fitness over successive iterations. It begins with a baseline fitness score, which then improves significantly at iteration 2, illustrating the positive impact of our tuning efforts. Despite some variability, the overall trend is upward, culminating in the highest fitness score in the last iteration. This graphical representation affirms the success of our mutation-based tuning approach, setting a robust foundation for future optimization through crossover techniques focused on the most influential hyperparameters: learning rate, momentum, and data augmentation.

Scatter Plots:

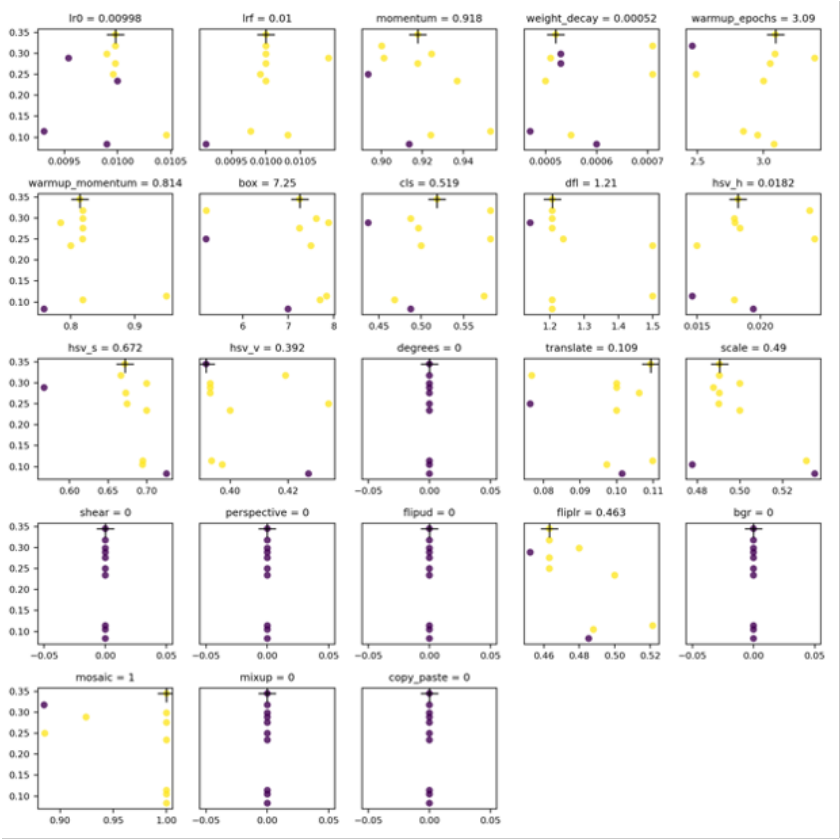

Figure 5.3:

The second graph presents a collection of scatter plots for various hyperparameters against the fitness score.

- Concentration of Fitness Scores: Yellow points indicate higher fitness concentrations. Hyperparameters corresponding to these points are potential candidates for optimal settings.
- Stability vs. Variability: Hyperparameters with vertical distributions of purple points denote fixed parameters with no mutation. Conversely, parameters with a

widespread of yellow and purple points reflect greater variability and sensitivity to tuning.

- Hyperparameter Influence: Prominent yellow vertical lines show that certain values for hyperparameters such as lr0, momentum, and warmup_momentum lead to better fitness scores.

Analyzing the scatter plots reveals certain hyperparameters are more influential in augmenting the model's accuracy. Consistently higher fitness scores associated with specific values of lr0 and momentum underscore their critical role in the model's detection capability. The best accuracy i.e mAP score we achieved is 56%.

In conclusion, our mutation-based hyperparameter tuning method over ten iterations has led to a promising yet steady mAP score of 56%, demonstrating the potential of the approach in refining model accuracy. Although this marks an improvement, it falls short of the 89% mAP of the baseline model. While additional iterations could bridge this gap, they would also entail significant computational investments. Based on the knowledge gained from mutation, we aimed to enhance our hyperparameter optimization even further. While mutation successfully guided smaller adjustments, we realized that exploring broader combinations of the most impactful hyperparameters through crossover could potentially enhance our model's performance even further.

APPROACH 3

5.1.2 Hyperparameter Tuning using Crossover :

Building on the solid basis provided by mutation-based hyperparameter optimization, we set out to improve the performance of our YOLO model by introducing a crossover method into the tuning process. The crossover mechanism, which draws inspiration from genetic algorithms, mimics biological reproduction by inheriting qualities from both parents. Its goal is to combine the best hyperparameters from each parent to produce children with optimal characteristics.

The custom crossover process was meticulously designed to merge hyperparameters from two distinct sets to form a new set, referred to as the "child." This child has the potential to embody the optimal characteristics of both parent sets, thereby fostering a more extensive exploration of the hyperparameter space.

Experimentation and Results

Crossover Working:

1. Hyperparameter Optimization Setup

Table 5.1: Key Hyperparameters, Their Importance, and Tested Values in Model Tuning

Hyperparameter	Importance	Tested Values
Initial Learning Rate (lr0)	The initial learning rate sets the starting speed at which the model learns. It is crucial as too high a rate can cause the model to converge too quickly to a suboptimal solution, and too low a rate can slow down the training process unnecessarily.	0.05, 0.01, 0.1
Final Learning Rate (lrf)	The final learning rate dictates the learning speed as the training progresses towards completion. It is important for fine-tuning the model, ensuring it makes smaller adjustments as it approaches optimal performance.	0.05, 0.01, 0.1
Epoch	The number of epochs determines how many times the model will work through the entire training dataset. This affects how well the model learns from the data, with too few epochs leading to underfitting and too many possibly causing overfitting.	20, 50, 100
Image Size	Image size impacts the level of detail the model can learn from each image. Larger images provide more detail but require more computational power, while smaller images are less demanding but might miss finer details crucial for accurate predictions.	640, 450, 500

- **Population Size (POPULATION_SIZE = 10):** We decided to use a population size of 10, meaning our algorithm would explore 10 different combinations of hyperparameters at a time. This provides a good balance between exploring diversity and computational efficiency.
- **Number of Generations (NUM_GENERATIONS = 2):** We set the evolutionary process to run for two generations. While more generations could lead to further refinement, it was important to strike a balance between thorough exploration and practical time constraints.
- **Mutation Rate (MUTATION_RATE = 0.1):** we set the mutation rate to 0.1. This means during the crossover process, there's a slight chance of random changes to the new 'child' hyperparameter sets, ensuring the exploration doesn't get stuck in a single area.

2. Initializing the Search The initialization phase randomly combines hyperparameters to form 10 distinct sets. Each set is termed an 'individual', and collectively, they

constitute our initial population. Within the generational loop, we assess and evolve our population of hyperparameters:

- Fitness Evaluation: For each individual, we define and apply a calculate_fitness function. This function conducts model training using the individual hyperparameters and evaluates performance, producing a 'fitness' score based on a weighted sum of performance metrics like mAP, precision, and recall.
- Selection Process: Once we've assessed the entire population, we implement a 'truncation selection' method, also known as elitism. This method selects the top-performing individuals—in our case, the four sets of hyperparameters with the highest fitness scores.
- Crossover Mechanism: We pair these elite individuals to perform crossover, creating 'children' that inherit hyperparameters from both 'parents'. The crossover process aims to combine the strengths of two individuals to potentially produce an even more fit offspring. In our two-generation model, each crossover yields two children, forming the new population for the next generation.
- Population Update: The new generation of offspring now becomes the current population, subject to the next iteration of fitness evaluation and selection. This evolutionary process is repeated until we reach our predetermined number of generations.

Our analysis utilizes various metrics to evaluate the model's detection capabilities. The 'fitness' score provides a holistic view, while metrics like mean Average Precision (mAP) offer deeper insights into specific aspects of performance.

Results :

```
Results saved to runs/detect/train22222222222
Results are printed below:
{'fitness': 0.5565991295454545,
 'metrics/mAP50(B)': 0.9266166666666668,
 'metrics/mAP50-95(B)': 0.5154860698653199,
 'metrics/precision(B)': 0.8858666058054272,
 'metrics/recall(B)': 0.904593453088003}
2
[100, 640, 0.05, 0.1]
[[100, 640, 0.05, 0.1], [100, 640, 0.05, 0.1]]
```

Figure 5.4:

Key Performance Metrics:

Mean Average Precision (mAP@50): Our model's mAP@50 has risen to an impressive 0.9266, indicative of its keen ability to accurately detect and classify objects at the 50% IoU benchmark.

Precision: A precision value of 0.8859 highlights the model's strong ability to minimize false positives.A desirable trait, particularly in critical application areas.

Recall: The recall rate, at 0.9046, confirms that the model successfully captures the vast majority of relevant instances within the dataset, a critical attribute for applications with little room for error.

Hyperparameter Selection and Its Efficacy: The chosen hyperparameter set—consisting of 100 epochs, an image size of 640, an initial learning rate of 0.05, and a final learning rate of 0.1—has proven to be exceptionally effective. This configuration has allowed our model to undergo thorough learning across numerous iterations, without rushing through or stalling in the optimization process. The crossover approach has had a significant positive influence, increasing the mAP@50 from an already impressive 89% to a remarkable 92%.

Graphs Analysis for Hyperparameter Tuning using Crossover

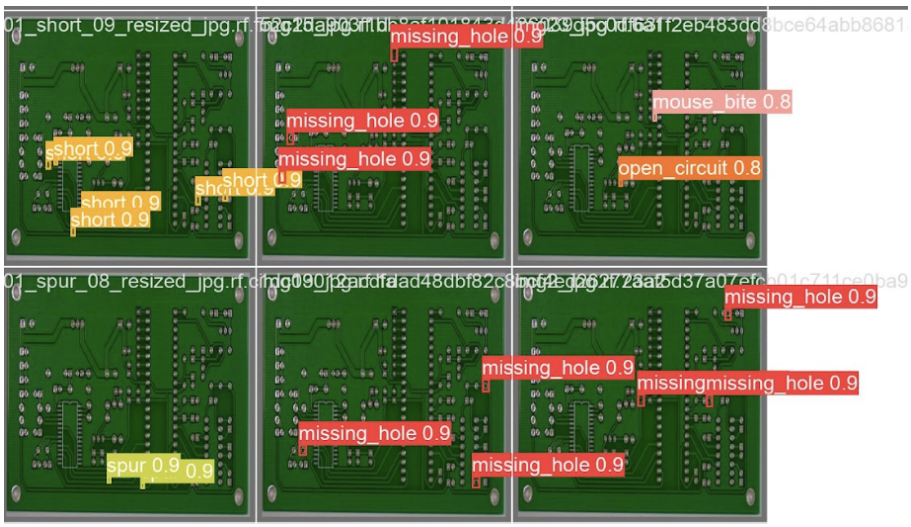

Figure 5.5: PCB defect detection with conf. scores

The crossover process resulted in high confidence scores e.i 0.9 for defect detection, particularly in the 'short', 'spurious copper', and 'missing hole' classifications.These high confidence scores show that the model is detecting these flaws with high accuracy. No classes were recognized with a confidence score of less than 0.8, indicating that the crossover strategy improved the performance of our models.

Precision-Recall Curve (Figure 3.19)

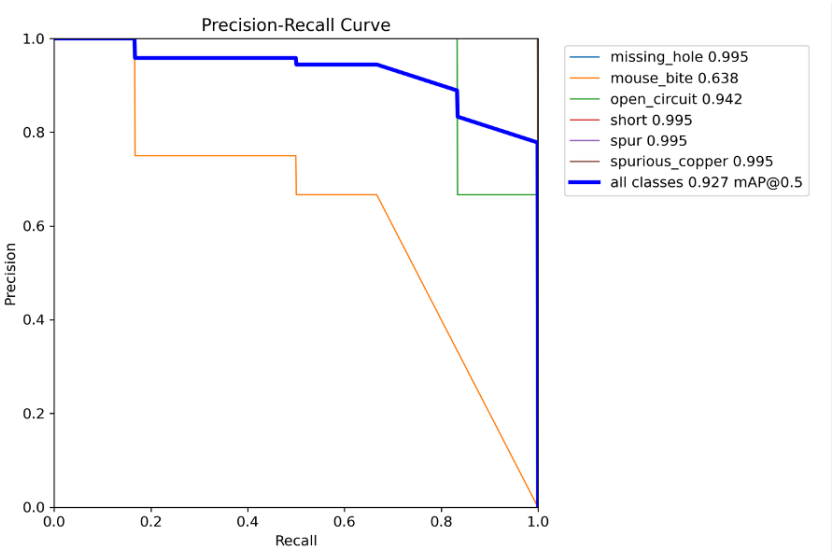

Figure 5.6: Precision-Recall Curve

The enhancements from crossover tuning are clearly reflected in the Precision-Recall Curve graph. We observed exceptional precision across most defect classes, particularly 'missing hole' and 'short', which maintained high precision across all recall levels. While 'mouse bite' showed lower precision, indicating a space for further improvement, the overall mean Average Precision (mAP) of 0.927 at an IoU threshold of 0.5 signifies a strong performance across all classes.This is an improvement considering the the mAP of pre-crossover technique (manual tuning) was 0.89. This improvement in the precision-recall balance demonstrates the crossover method's effectiveness.

Insights from the Normalized Confusion Matrix (Figure 3.20):

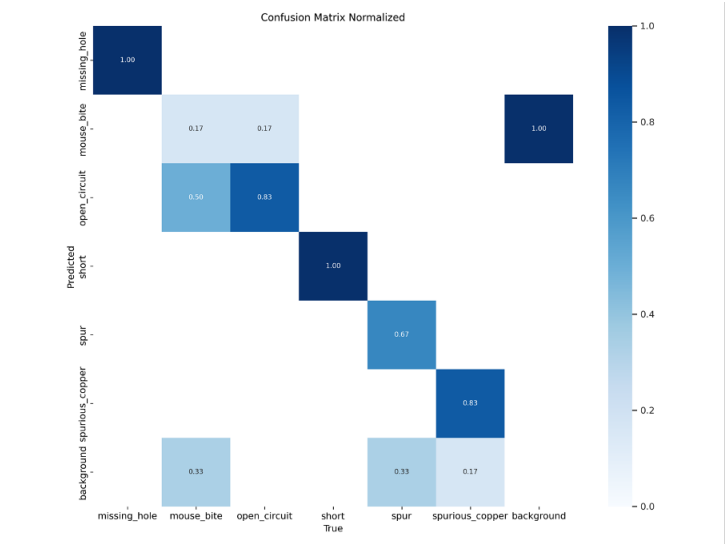

Figure 5.7: Normalized Confusion Matrix

Accuracy by Class: The 'missing hole' class shows a normalized value of 1.00, indicating a 100% true positive rate in detection, which is an improvement if previously the detection rate was lower(around 90%). The 'short' defect type also shows a perfect score, pointing to a significant enhancement from the pre-crossover model. The matrix indicates a clear overall improvement in accuracy for detecting various defects post-crossover, with many classes showing high values along the diagonal

Insights from the Raw Count Confusion Matrix (Figure 3.21):

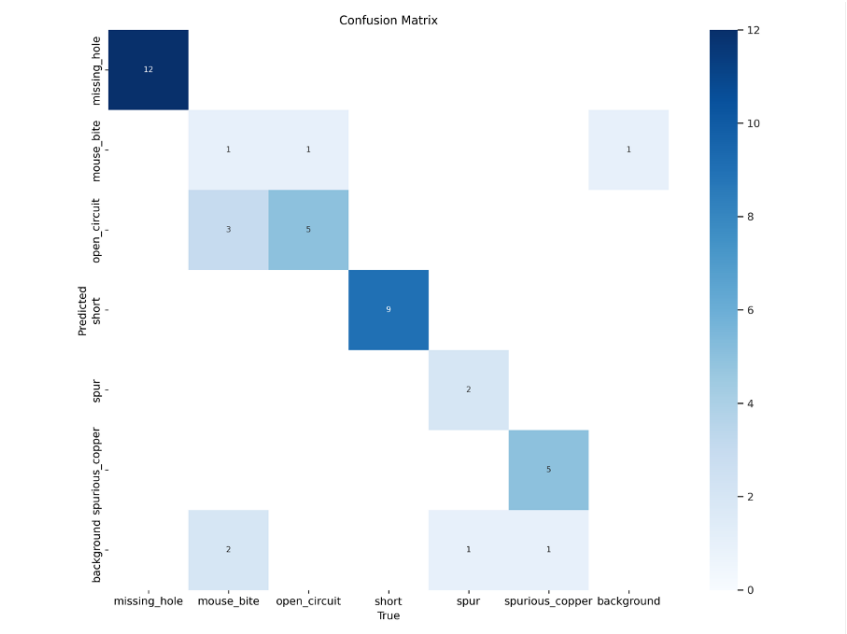

Figure 5.8: Confusion Matrix

- This matrix demonstrates the model's strength in identifying'missing hole' defects, with a high count of 12 true positives, which, if higher than the pre-crossover counts, indicates that the model is more capable of effectively detecting frequent defects.
- A decrease in misclassifications, particularly between'mouse bite' and 'open circuit' is seen, which previously exhibited a high frequency of errors with our baseline model. The counts show a more balanced distribution of mistakes across classes, implying that the crossover tuning, resulted in a more balanced approach to identifying various defect kinds, which is likely an improvement over a pre-crossover model with disproportionate errors.

Performance Metrics Graphs:

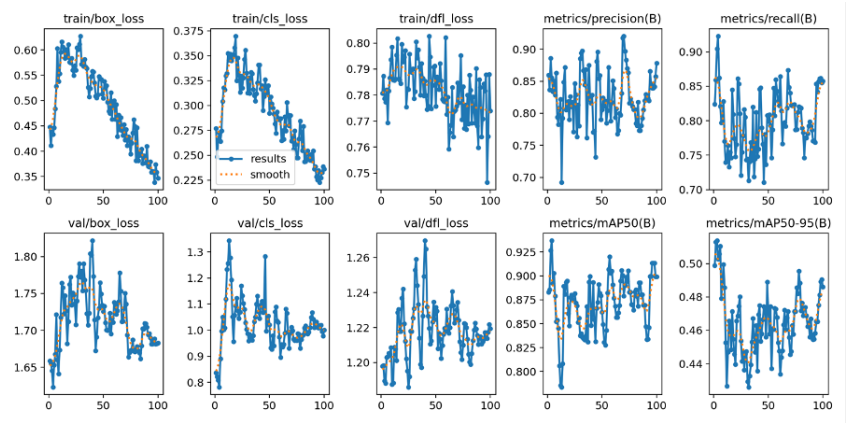

Figure 5.9: Performance Metrics Graphs

- In the validation loss graphs, there's noticeable noise; however, this can be interpreted as the model's exploration of the solution space, potentially leading to better generalization when confronted with new, unseen data.
- The training loss metrics—box, class, and direction focal loss (dfl)—exhibit a more refined descent, settling at lower values as compared to our baseline model. This suggests that the model has become more adept at minimizing these errors through the course of its training.
- The training loss metrics—box, class, and direction focal loss (dfl)—exhibit a more refined descent, settling at lower values as compared to our baseline model. This suggests that the model has become more adept at minimizing these errors through the course of its training.
- When examining the precision and recall metrics, we observe that they attain higher levels, with precision reaching a peak of 0.9 and recall regularly approaching 0.8. This demonstrates our model's ability to accurately detect and categorize items has been greatly improved
- Turning our attention to the mAP scores, we see an upward trajectory in both mAP50 and mAP50-95, with mAP50 showing scores consistently close to the 0.9 mark

In summary, crossover hyperparameter tuning resulted in significant gains in confidence scores, precision-recall balance, and detection accuracy, indicating a more robust model. The consistent performance across various metrics suggests that the crossover method has helped in navigating the hyperparameter space more effectively, leading to a model that outperforms its previous iteration. It successfully outperforms both the baseline model and the Mutation Approach, achieving an impressive mAP score of 92%.

5.2 Comparison and Analysis

In our search for the best hyperparameter settings for our YOLO model, we thoroughly compared three different approaches:

- the Baseline model
- the Mutation-based approach
- the Crossover approach.

The baseline model set a high standard with a mAP of 89%, acting as an impressive benchmark for our further tuning efforts. It gave a credible measure of the model's inherent capabilities before any complex tuning was performed.

After switching to mutation-based hyperparameter tweaking, we achieved a noteworthy mAP of 56% across ten iterations. This strategy resulted in modest improvements in model performance, demonstrating the method's capacity to discover and keep advantageous hyperparameter changes. However, this technique did not outperform the baseline model, indicating that while mutation tuning progressed, there was still room for major improvement.

Our final and most advanced strategy utilized the crossover method. Drawing on genetic algorithms, this approach extended the concept of mutation by combining the best hyperparameters from high-performing sets to generate a new, potentially superior configuration. The use of crossover hyperparameter adjustment resulted in a significant boost in our model's performance, eventually outperforming the baseline model with a mAP score of 92%. The crossover mechanism was particularly effective at encouraging a thorough exploration of the hyperparameter space, resulting in a more robust model with high precision and recall rates, as well as high confidence scores in defect identification.

Each strategy yielded unique insights into the hyperparameter optimization process. The baseline model provided a solid foundation, the mutation technique refined this by iteratively altering parameters, and the crossover strategy took these adjustments to the next level by merging the strengths of several sets to create a superior hyperparameter configuration. The findings clearly show that the crossover technique, with its subtle hyperparameter combinations, is the most promising approach for improving the YOLO model's performance, ultimately leading to a complex, finely-tuned detection system.

Tuning Method	**Map Achieved**	**Precision**	**Recall**	**Training Time**
Baseline Model	89%	89.5%	85%	30 mins
Mutation Approach	56%	49%	63%	2 hours
Crossover Approach	92%	92%	92.2%	3 hours

Table 5.2: Comparison

The YOLOv9 model required robust GPU support due to its computational complexity, and was run on Tesla T4 GPUs for efficient processing. The manual tuning phase was relatively quick, taking about 30 minutes per training cycle, which, while needing considerable human oversight, kept GPU usage to a minimum. Mutation tuning took around 2 hours, which allowed for several hyperparameter adjustments without significantly increasing computational costs. The most resource-intensive approach was crossover tuning,

employing a genetic algorithm that extended processing times to 2-3 hours. Although this method increased GPU usage, it enabled a deeper and more automated exploration of hyperparameter space, leading to a notable improvement in model performance with a maximum mAP of 92%.

Chapter 6

Model Inference

6.1 Web Application Development for Inference Engine

The interface of the web application was meticulously designed to prioritize user-friendliness and functionality. Key features of the interface include:

- **Image Upload Capability:** Users can upload images of PCBs directly through the interface, simplifying the process for non-technical users.
- **Webcam Feature for Real-Time Deployment:** Integrates real-time video capturing capabilities, enabling instant on-the-spot inspections and immediate results for PCB defect detection, which is crucial for quality control in manufacturing environments.
- **Adjustable IOU and Confidence Threshold Sliders:** These sliders provide users with the flexibility to tailor the sensitivity of the defect detection process according to specific requirements, ensuring that the application can be adapted to various operational contexts with precision.

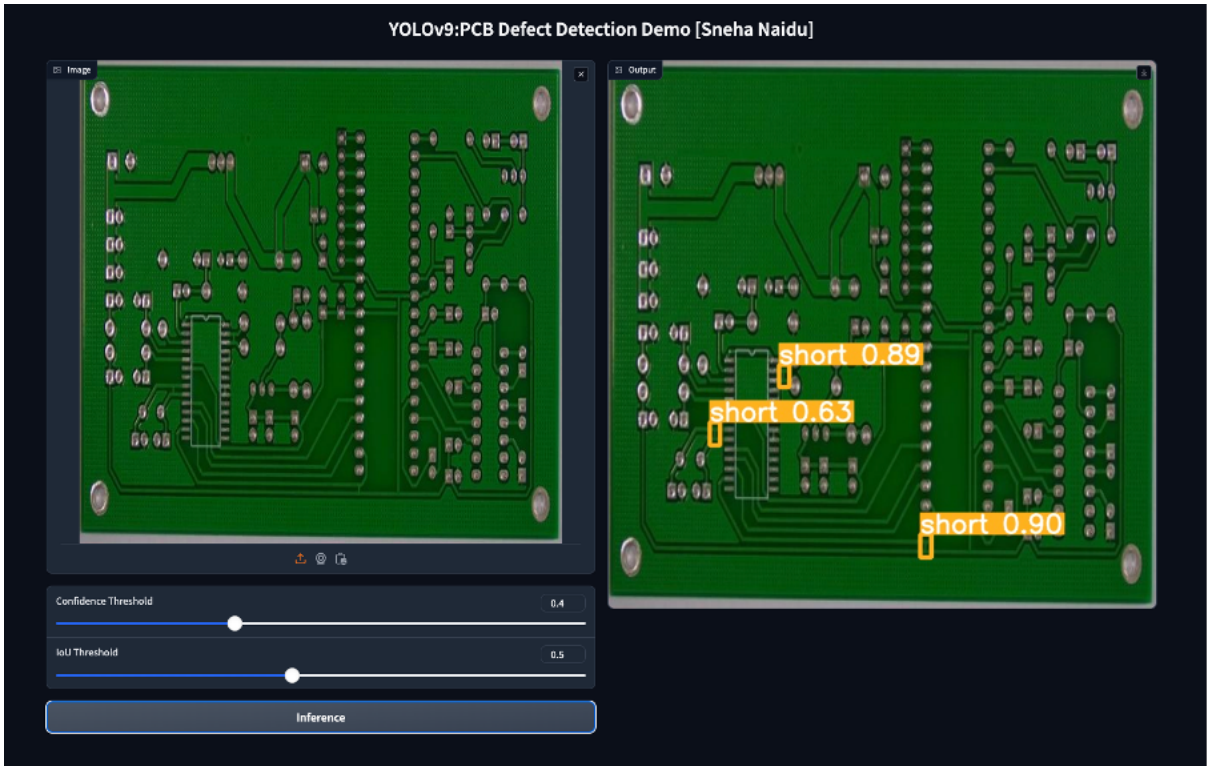

Figure 6.1: Web App

6.2 Integration of the YOLOv9 Model

The integration of the YOLOv9 model into the web application was a crucial component designed to leverage its powerful object detection capabilities. This process included several essential steps:

Environment Setup We used Google Colab as a hosting environment to leverage its GPU support, essential for running the computationally intensive YOLOv9 model. The Google Drive is mounted to access the project directory where the YOLOv9 model and associated files are stored, ensuring seamless file management and accessibility. All necessary Python dependencies are installed from the requirements.txt file of the YOLOv9 project, setting up a consistent environment for model execution.

Function for Running Inference (run_inference):

This function is written to manages outputs efficiently and avoid data overwriting. It generates a unique directory for each inference session using a function - generate_random_code(). This ensures that all the output files (results) are stored in an isolated manner.

Model Invocation: The model is invoked with the path to the input image, along with user-defined IOU and confidence thresholds.

Processing and Output: After running the detection, the function updates the user with the path of the processed image, which displays the detected defects.

Real-Time Object Detection Capabilities

The real-time detection capabilities is am important part of the application, providing immediate and accurate defect assessments:

Live Image Processing: The app processes live images feeds through the webcam, applying the YOLOv9 model in real-time to detect and classify PCB defects accurately.

Defect Highlighting and Annotations: Detected defects are visually highlighted on the PCB images shown to the user, with annotations that clearly label the type of defect detected, enhancing the interpretability of results.

Conclusion

The development of this web application not only leverages the advanced capabilities of the YOLOv9 model for effective PCB defect detection but also embodies a user-centric approach, ensuring that the tool is accessible and valuable across various industrial settings. Each time the application is launched it generates a publicly accessible URL. This URL can be shared and used by anyone, enhancing the collaborative potential and ease of access to the tool.The application's intuitive design, combined with real-time processing capabilities and adjustable detection parameters, makes it a robust solution for industries requiring precise and efficient defect detection.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

In conclusion we successufully implemented the YOLOv9 model for PCB Defect Detection along with our unique approch of Hyperparameter Tuning using GA to enhance the model's performance. The exploration of hyperparameter tuning for our YOLO model was done across three distinct approaches—baseline, mutation, and crossover. This research has yielded a comprehensive understanding of the methods available to enhance model performance.

The baseline model, with an mAP of 89%, served as a solid control in our experiments, reflecting the model's performance without advanced tuning techniques. It established a target for the other methods to aspire to and potentially exceed.

The mutation-based approach, which fine-tunes hyperparameters through selective and incremental changes, showed promise with a final mAP of 56%. This technique's strength lies in its simplicity and its ability to iteratively learn and improve model accuracy. Despite its potential, it fell short of the baseline model's performance, indicating that while useful, mutation alone might be insufficient for reaching the pinnacle of our model's capabilities.

Building upon the mutation method, we implemented the crossover technique, inspired by genetic algorithms, to combine the best traits of selected hyperparameter sets. This strategy outperformed both the baseline and mutation approaches, achieving a remarkable mAP of 92%. The crossover method's efficacy in navigating the hyperparameter space was evident, leading to substantial improvements in precision, recall, and confidence scores for defect detection.

Overall, the progression from baseline to mutation, and finally to crossover, illustrates a journey of escalating refinement and complexity in tuning strategies. Each method contributed to a deeper understanding of the model's learning dynamics and how different hyperparameters influence performance. The ultimate conclusion is that while the baseline provides a dependable foundation, and mutation offers incremental advancements,

the crossover method unlocks the full potential of hyperparameter tuning, delivering a powerful and precise model that stands as a testament to the efficacy of sophisticated optimization strategies in machine learning. One of the key findings are the best set of hyperparameters found in the Crossover Technique – epoch size = 100, image_size = 640*640pixels, initial learning rate of = 0.05 and final learning rate = 0.1.

Furthermore, the project addressed real-time and industrial environment needs. The YOLOv9 Compressed version was selected to ensure lightweight deployment on edge devices, balancing computational demands with portability and low power consumption. This ensures effective operation within factory settings without compromising speed or accuracy.

Lastly, we created a web application interface to showcase the trained YOLOv9 model for real-time PCB defect detection. Its image upload and live webcam features make it applicable to real world scenarios. Its user-friendly interface allows users to upload images or utilize a live webcam feed for defect detection. Adjustable Intersection over Union (IoU) and confidence threshold settings allow users to adapt the findings to their unique requirements, balancing the number of detections with the confidence of those detections. The application delivers results in seconds, accelerating the quality control process.

7.2 Future Work

Crossover Technique While the custom crossover technique significantly improves model performance, it is computationally intensive. With adequate resources and computational power, further refinement and optimization of the crossover technique hold promise for achieving even better results. Exploring parallelization methods, distributed computing frameworks, or specialized hardware accelerators could unlock new avenues for enhancing the efficiency and scalability of the crossover approach.

Exploration of Alternative Industries: Future research could investigate the use of custom-trained models for defect detection in diverse industries beyond PCB manufacturing. Examples include automotive components, pharmaceutical products, agricultural produce, and consumer electronics. Each industry presents unique challenges and requirements for defect detection, ranging from complex surface anomalies in automotive parts to microscopic defects in pharmaceutical tablets. By extending the application of defect detection technologies to these domains, we can address a broader spectrum of quality control needs and contribute to improved product reliability and safety across various sectors.

In conclusion, our findings emphasize the significance of real-time defect detection capabilities as well as the possibility of novel methodologies such as the crossover method. Looking ahead, investigating new sectors and perfecting defect detection technologies will promote further breakthroughs in quality control practices and boost innovation across industries.

Bibliography

[1] Bani-Hani, D. [2020], 'Genetic algorithm: A simple and intuitive guide', `https://towardsdatascience.com/genetic-algorithm-a-simple-and-intuitive-guide-51c04cc1f9ed`. Accessed: 2024-04-20.

[2] Chaudhary, V., Dave, I. R. and Upla, K. P. [2017], Automatic visual inspection of printed circuit board for defect detection and classification, *in* '2017 international conference on wireless communications, signal processing and networking (WiSPNET)', IEEE, pp. 732–737.

[3] DataCamp [2023], 'Yolo object detection explained', `https://www.datacamp.com/blog/yolo-object-detection-explained`.

[4] Du, B., Wan, F., Lei, G., Xu, L., Xu, C. and Xiong, Y. [2023], 'Yolo-mbbi: Pcb surface defect detection method based on enhanced yolov5', *Electronics* **12**(13), 2821.

[5] Feng, H., Mu, G., Zhong, S., Zhang, P. and Yuan, T. [2022], 'Benchmark analysis of yolo performance on edge intelligence devices', *Cryptography* **6**(2), 16.

[6] Hu, B. and Wang, J. [2020], 'Detection of pcb surface defects with improved faster-rcnn and feature pyramid network', *Ieee Access* **8**, 108335–108345.

[7] Huang, W., Wei, P., Zhang, M. and Liu, H. [2020], 'Hripcb: a challenging dataset for pcb defects detection and classification', *The Journal of Engineering* **2020**(13), 303–309.

[8] Jain, S. [2017], 'Introduction to genetic algorithm & their application in data science', *Analytics Vadhya* .

[9] Khan, M. A., Park, H. and Chae, J. [2023], 'A lightweight convolutional neural network (cnn) architecture for traffic sign recognition in urban road networks', *Electronics* **12**(8), 1802.

[10] Newton, E. [2023], 'Why visual ai inspection is essential for detecting pcb defects'. **URL:** *https://www.manufacturingtomorrow.com/story/2023/09/why-visual-ai-inspection-is-essential-for-detecting-pcb-defects/21326/*

[11] *PCB Dataset* [2024], `https://t.ly/2_anp`. Accessed: 2024-04-20.

[12] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. [2016], You only look once: Unified, real-time object detection, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 779–788.

[13] Ren, S., He, K., Girshick, R. and Sun, J. [2016], 'Faster r-cnn: Towards real-time object detection with region proposal networks', *IEEE transactions on pattern analysis and machine intelligence* **39**(6), 1137–1149.

[14] Tang, J., Liu, S., Zhao, D., Tang, L., Zou, W. and Zheng, B. [2023], 'Pcb-yolo: An improved detection algorithm of pcb surface defects based on yolov5', *Sustainability* **15**(7), 5963.

[15] Vaiciukynas, E., Gelzinis, A., Verikas, A. and Bacauskiene, M. [2018], Parkinson's disease detection from speech using convolutional neural networks, *in* 'Smart Objects and Technologies for Social Good: Third International Conference, GOODTECHS 2017, Pisa, Italy, November 29-30, 2017, Proceedings 3', Springer, pp. 206–215.

[16] WongKinYiu [2024], 'Implementation of YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information', `https://github.com/WongKinYiu/yolov9`. Accessed: 2024-04-20.

Chapter 8

Appendix

8.1 Code: Model Training

```
# Mounting Google Drive
from google.colab import drive
drive.mount('/content/drive')

#Change the current working directory to the specified path where the YOLOv9 project resides.
%cd /content/drive/MyDrive/Machine_Learning_CW/yolov9

# Install all the dependencies required for YOLOv9 from the requirements.txt file.
!pip install -r requirements.txt -q


from ultralytics import YOLO

# Build a YOLOv9c model from pretrained weight
model = YOLO('yolov9c.pt')

# Display model information
model.info()

results = model.train(data='/content/drive/MyDrive/Machine_Learning_CW/PCB_datasets/PCB_2_D.v1i.yolov9/data.yaml',
                      epochs=20, imgsz=640)


results = model.train(data='/content/drive/MyDrive/Machine_Learning_CW/PCB_datasets/PCB_2_D.v1i.yolov9/data.yaml',
                      epochs=50, imgsz=640)


# Train the model
# The training process is configured with specific parameters including the dataset path, number of epochs, and input image size.
# Note: The dataset specified by 'data.yaml' contains paths to training and validation images along with class labels.
```

```
31
32 results = model.train(data='/content/drive/MyDrive/Machine_Learning_CW/
       PCB_datasets/PCB_2_D.v1i.yolov9/data.yaml',
33                        epochs=100, imgsz=640)
34
35
36 import os
37 from IPython.display import Image, display
38
39
40 # Set the directory path ofr images
41 directory_path = '/content/drive/MyDrive/Machine_Learning_CW/yolov9/
       runs/detect/100_epoch_manual'
42
43 # List all files in the directory
44 all_files = os.listdir(directory_path)
45
46
47 # Loop through all files, and display them if they are images
48 for file_name in all_files:
49     if file_name.lower().endswith(('.png', '.jpg', '.jpeg')):
50         print(f"Displaying {file_name}")
51         display(Image(filename=os.path.join(directory_path, file_name),
       width=600))
52
53
54 import pandas as pd
55
56 # Set the path to your CSV file
57 csv_file_path = '/content/drive/MyDrive/Machine_Learning_CW/yolov9/runs
       /detect/100_epoch_manual/results.csv'
58
59 # Read the CSV file into a DataFrame
60 df = pd.read_csv(csv_file_path)
61
62 # Adjust pandas settings to display all rows
63 pd.set_option('display.max_rows', 100)
64
65 # print(df.columns.tolist())
66
67
68 # # Select only the 'epoch' column and the 'metrics/mAP_0.5(B)' column
69 selected_columns_df = df[['                  epoch', '       metrics/
       mAP50(B)', '    metrics/mAP50-95(B)']]
70
71 # # Print the selected columns of the DataFrame
72 print(selected_columns_df)
73
74
75 model.tune(data='/content/drive/MyDrive/Machine_Learning_CW/
       PCB_datasets/PCB_2_D.v1i.yolov9/data.yaml',
76            epochs=100, iterations=10, optimizer='AdamW', plots=False,
       save=True, val=True)
77
78
79
80 import os
81 from IPython.display import Image, display
```

```

# Set the directory path ofr images
directory_path = '/content/drive/MyDrive/Machine_Learning_CW/yolov9/runs/detect/tune7'

# List all files in the directory
all_files = os.listdir(directory_path)


# Loop through all files, and display them if they are images
for file_name in all_files:
    if file_name.lower().endswith(('.png', '.jpg', '.jpeg')):
        print(f"Displaying {file_name}")
        display(Image(filename=os.path.join(directory_path, file_name), width=600))



import pandas as pd

# Set the path to your CSV file
csv_file_path = '/content/drive/MyDrive/Machine_Learning_CW/yolov9/runs/detect/tune7/tune_results.csv'

# Read the CSV file into a DataFrame
df = pd.read_csv(csv_file_path)

# Adjust pandas settings to display all rows
# pd.set_option('display.max_rows', 100)

# print(df.columns.tolist())


# # # Select only the 'epoch' column and the 'metrics/mAP_0.5(B)' column
selected_columns_df = df[['fitness']]

# # # Print the selected columns of the DataFrame
print(selected_columns_df)
```

Listing 8.1: code

8.2 Code:Custom Crossover Method

8.2.1 Importing Libraries and Defining Fitness Function

```


from pprint import pprint

# Documentation of all parameters
# https://docs.ultralytics.com/usage/cfg/#train-settings

```

```
8
9  # Define the fitness function to evaluate model performance
10 def calculate_fitness(individual):
11   # Training the model with given individual hyperparameters
12     results = model.train(data='/content/drive/MyDrive/
       Machine_Learning_CW/PCB_datasets/PCB_2_D.v1i.yolov9/data.yaml',
13                           epochs=individual[0], imgsz=individual[1], lr0=
       individual[2], lrf=individual[3])
14
15
16     # Printing and returning the training results
17     print("Results are printed below: ")
18     results_dict = results.results_dict
19     pprint(results_dict)
20
21     return results_dict
```

Listing 8.2: code

8.2.2 Initializing Hyperparameters and Population

```
1  # @title
2  import random
3  import numpy as np
4
5  # Defining hyperparameters for optimization
6  hyperparameters = {
7      'num_epochs': [20, 50, 100],
8      'image_size': [640, 640, 640],
9      'lr0':[0.05, 0.01, 0.1],
10     'lrf':[0.05, 0.01, 0.1]
11 }
12
13 # Setting up the population size and generations
14 POPULATION_SIZE = 10
15 NUM_GENERATIONS = 2
16
17 # Mutation rate for genetic algorithm
18 MUTATION_RATE = 0.1
19
20 # Initializing the population with random hyperparameters
21 population = []
22 for i in range(POPULATION_SIZE):
23     individual = []
24     for param in hyperparameters.values():
25         individual.append(random.choice(param))
26     population.append(individual)
27
28
29 # Display initial population
30 print(population)
```

Listing 8.3: code

8.2.3 Evolutionary Algorithm: Selection, Crossover, and Generation Update

```
1
2  import random
3
4  # Main loop for evolving the population
5  for generation in range(NUM_GENERATIONS):
6
7      print("New Generation:")
8
9  # Calculating fitness scores for each individual
10     fitness_scores = []
11     for individual in population:
12         print("Starting the fitness evaluation for individual: " + str(
       individual))
13         # SN: temp
14         fitness_dict = calculate_fitness(individual)
15         fitness_score = fitness_dict['fitness']
16
17         fitness_scores.append(fitness_score)
18
19 # print(fitness_scores)
20
21
22     # Selecting parents based on fitness scores using 'trunction
       selection' selection
23     parents = []
24     fitness_sort = sorted(range(len(fitness_scores)), key=lambda i:
       fitness_scores[i], reverse=True)
25     top_two_indices = fitness_sort[:2]
26
27     for index in top_two_indices:
28         parents.append(population[index])
29
30 # print(fitness_scores)
31 # print(len(parents))
32
33
34    # Generating new population through crossover between selected
       parents
35     new_population = []
36     for i in range(len(parents)):
37         parent1, parent2 = random.sample(parents, 2)
38         child = []
39         for j in range(len(parent1)):
40             if random.random() < 0.5:
41                 child.append(parent1[j])
42             else:
43                 child.append(parent2[j])
44
45         new_population.append(child)
46
47
48     # Updating the population with the new generation
49     population = new_population
50
```

```

# Final population output
print(len(population))
print(child)
print(population)
```

Listing 8.4: code

8.3 Code: Web App for Inference

8.3.1 Random Code Generation for Output Directory:

```
import random
import string

def generate_random_code(length=3):
  """Generates a random code of the specified length using letters."""
  letters = string.ascii_letters  # All uppercase and lowercase letters
  code = ''.join(random.choice(letters) for i in range(length))
  return code
```

Listing 8.5: code

8.3.2 Real-time Image Processing

```
from detect import run
import os

model_path = "/content/drive/MyDrive/Machine_Learning_CW/yolov9/runs/
    detect/train22222222222/weights/best.pt"

def run_inference(source_img_path, conf_threshold, iou_threshold):
  # Generate a 3 letter random directory for output
  name_of_exported_folder = generate_random_code()

  print("[Source path:]" + source_img_path)
  print("[Exported folder:]" + name_of_exported_folder)

  run(weights=model_path,
      source=source_img_path,
      device=0,
      name=name_of_exported_folder,
      conf_thres=conf_threshold,
      iou_thres=iou_threshold)

  file_name = os.path.basename(source_img_path)
  predicted_img_path = str("runs/detect/" + name_of_exported_folder + "
    /" + file_name)

  print("[predicted img path:]" + predicted_img_path)

```

```
25   return predicted_img_path
```

Listing 8.6: code

8.3.3 Run Inference

```
1 def runInference_DisplayImage(source_img_path, conf_threshold,
    iou_threshold):
2   predicted_img_path = run_inference(source_img_path, conf_threshold,
    iou_threshold)
3   image = gr.Image(predicted_img_path)
4   return image
```

Listing 8.7: code

8.3.4 GUI

```
1 def app():
2     with gr.Blocks():
3         with gr.Row():
4             with gr.Column():
5                 img_path = gr.Image(type="filepath", label="Image")
6                 conf_threshold = gr.Slider(
7                     label="Confidence Threshold",
8                     minimum=0.1,
9                     maximum=1.0,
10                    step=0.1,
11                    value=0.4,
12                )
13                iou_threshold = gr.Slider(
14                    label="IoU Threshold",
15                    minimum=0.1,
16                    maximum=1.0,
17                    step=0.1,
18                    value=0.5,
19                )
20                yolov9_infer = gr.Button(value="Inference")
21
22            with gr.Column():
23                output_numpy = gr.Image(type="numpy",label="Output")
24
25        yolov9_infer.click(
26            fn=runInference_DisplayImage,
27            inputs=[img_path, conf_threshold, iou_threshold],
28            outputs=[output_numpy],
29        )
30
31
32 gradio_app = gr.Blocks()
33 with gradio_app:
34     gr.HTML(
35         """
36     <h1 style='text-align: center'>
```

```
    YOLOv9:PCB Defect Detection Demo [Sneha Naidu]
    </h1>
    """)
    # gr.HTML(
    #     """
    #     <h3 style='text-align: center'>
    # Legend: [0 missing_hole][1 mouse_bite][2 open_circuit][3 short][4
    spur][5 spurious_copper]

    #     </h3>
    #     """)
    with gr.Row():
        with gr.Column():
            app()

gradio_app.launch(debug=True, share=True)
```

Listing 8.8: code