FSM Online Internship Completion Report
on

INTP2022-ML-3: Computer Vision to detect defects in PCB

In

Machine Learning

Submitted by

Keivalya Pandya
Birla Vishvakarma Mahavidyalaya

Under Mentorship of
Mr. Devesh Tarasia

FSM
Smart Manufacturing Simplified

IITD-AIA Foundation for Smart Manufacturing

1 June, 2022 to 31 July, 2022

Computer Vision to detect defects in PCB

Abstract

In the PCB manufacturing industry, one of the most important aspects of production is quality checking. PCB boards go through lots of production processes from panel cutting to laminating and one single defect in a board will make the entire board become obsolete. With the rise of electronic appliance demands every day, the demand for higher quality components is rising. With the high price tag of commercial automated optical inspection (AOI), lots of manufacturers are not able to do an automated inspection. With the continuous development of object detection technology, the YOLO series of algorithms with very high precision and speed has been used in various fault detection tasks. To establish a PCB fault detection system, we propose a fault detection method based on YOLOv5 and annotate the 1500 data sets.

Keywords: Neural Networks, Deep learning, YOLO, Industry automation

Table of Content

Contents

Introduction to the problem

In the PCB manufacturing industry, one of the most important aspects of production is quality checking. PCB boards go through lots of production processes from panel cutting to laminating and one single defect in a board will make the entire board become obsolete. With the rise of electronic appliance demands every day, the demand for higher quality components is rising. With the high price tag of commercial automated optical inspection (AOI), lots of manufacturers are not able to do an automated inspection. This is where Computer Vision and Machine Learning can provide an alternative for commercial AOI to assist small-scale manufacturers to do automated inspections. Image Subtraction enable users to easily find numerous visual defects in PCBs, especially with a complex method. By dissecting pictures and pointing out the differences in the output image, PCBs can be inspected quickly to find out the defective parts.

Essentially, deploying it to the web for remote access can ensure accessibility to a larger group of industry people.

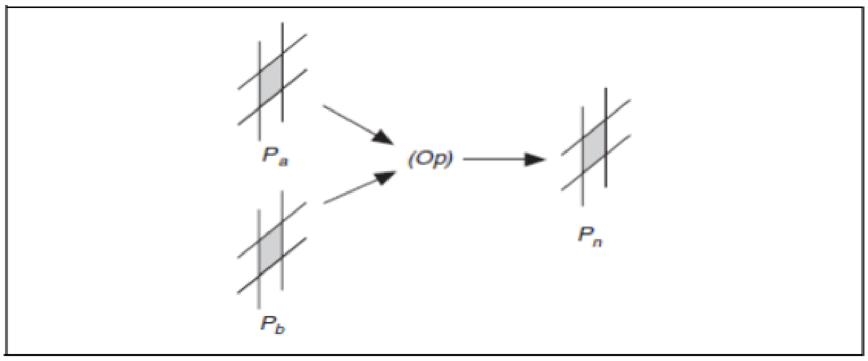
With the continuous development of object detection technology, the YOLO series of algorithms with very high precision and speed has been used in various fault detection tasks. To establish a PCB fault detection system, we propose a fault detection method based on YOLOv5 and annotate the 1500 data sets.

Literature Review

Logical operators for image

As we are provided with the template image (perfectly good PCB) and the test image (image with the faults corresponding to the template image), an initial approach towards detecting defects could be to subtract the test image pixels from the template image pixels. Pixel-by-pixel transformation is a procedure between pictures that applies logic or arithmetic. It creates a picture where each pixel's value is derived from pixels in other images that have the same coordinates.

If *A and B* are the images with a resolution *XY*, and *Op* is the operator, then the image *N* resulting from the combination of *A* and *B* through the operator *Op* is such that each pixel *P* of the resulting image *N* is assigned the value *pn = (pa)(Op)(pb)*; where *pa* is the value of pixel *P* in image *A*, and *pb* is the value of pixel *P* in image *B*.



Object Detection

Classification can be understood in 3 distinguished steps:

1. Classification, in which we extract a certain type of information, with a pre-defined category or instance ID to describe the entire image.

2. Detection, in which we detect bounding boxes within which the pixels of the classified objects are observed. Compared with classification, detection gives the understanding of the picture's foreground and background. The output of the detection model is a list, and each item of the list used a data group to give the category and position of the detected target (commonly used coordinate representation of rectangular detection box)
3. Segmentation is a pixel-level description of an image, which gives meaning to each pixel category (instance) and is suitable for scenes requiring high understanding.

We are going to use a detection-based model (instead of segmentation) in order to compensate for classification's lack of localization information, as well as segmentation's pixel lever complexity.

One-Stage and Two-Stage

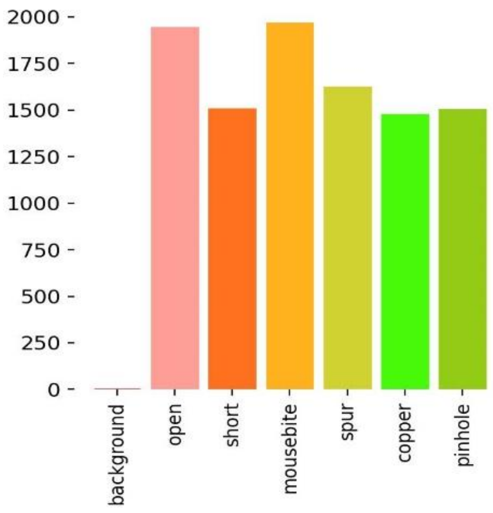
The one-stage network is represented by the YOLO series network, while the two-stage network is represented by faster-RCNN. The RPN is trained end-to-end to generate high-quality region proposals, which are used by Fast R-CNN for detection.

Understanding the Data and Exploratory Data Analysis (EDA)

About the Dataset

The collection comprises 1,500 picture pairs, each consisting of a template image devoid of defects and a tested image that has been aligned and annotated with the positions of the six most prevalent PCB defects: open, short, mouse bite, spur, pinhole, and spurious copper.

There are a total of 10013 labels in the dataset whose distribution is as follows:



1. Background - 0 (not used in the entire dataset)
2. Open - 1942
3. Short - 1506
4. Mouse bite - 1965
5. Spur - 1625
6. Pinhole - 1474
7. Spurious copper - 1501

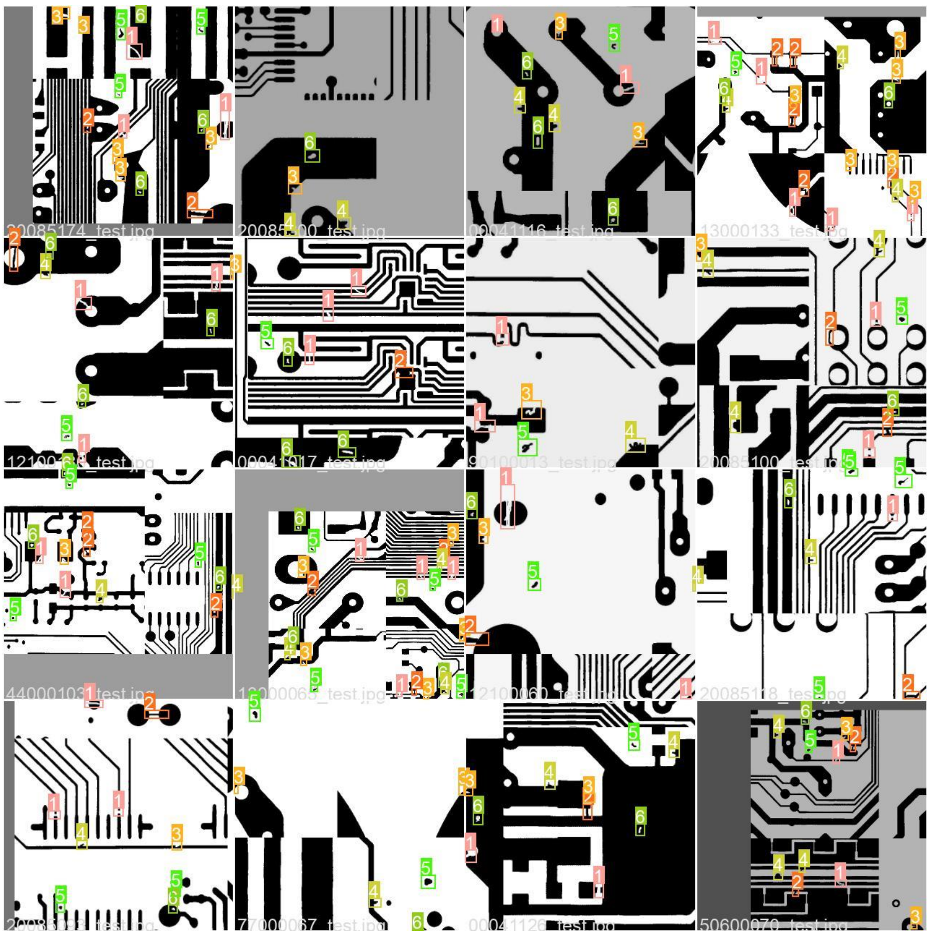
About the image

The linear scan CCD used to capture each image in this collection has a resolution of about 48 pixels per millimetre. The original size of the template and the tested image is around 16k x 16k pixels. They are then divided into several 640 × 640 pixels sub-images using a cropping process, then aligned using template matching methods. However, the 1500 defective PCB images and their annotation files will be primary resources for training our Deep Learning Model.

About the image annotations

We utilize an axis-aligned bounding box with a class ID for each flaw in the tested photos. Each annotated image owns an annotation file with the same filename, e.g.00041000_test.jpg, 00041000_temp.jpg and 00041000.txt are the tested image, template image and the corresponding annotation file. Each defect on the tested image is annotated as the format: x1, y1, x2, y2, type, where (x1,y1) and (x2,y2) are the top-left and the bottom-right corner of the bounding box of the defect and type is an integer ID that follows the matches:

0-background (not used), 1-open, 2-short, 3-mouse-bite, 4-spur, 5-copper, 6-pin-hole.



Encoded annotations of the training data (representation ID as mentioned above)

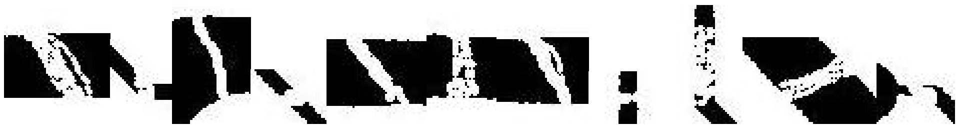
Cropping and extracting the defects

I have created an image cropping tool in order to crop the defects from the dataset

```
notation_file_path =
r"Dataset\PCBData\group{f_name}\{f_name}_not//".format(f_name=12000)
notation_file_dirs = os.listdir(notation_file_path)

saving_folder =
r"Dataset\PCBData\group{f_name}\{f_name}_defects//".format(f_name=12000)
saving_folder_dirs = os.listdir(saving_folder)

image_file_path =
r"Dataset\PCBData\group{f_name}\{f_name}//".format(f_name=12000)
image_file_dirs = os.listdir(image_file_path)
```

```
for imag in image_file_dirs:
    if os.path.isfile(image_file_path+imag):
        im = Image.open(image_file_path+imag)
        f,e = os.path.splitext(image_file_path+imag)
        if f.endswith("test"):
            f = f.split("//")[-1].split("_")[0]
            text_file = open(notation_file_path+f+".txt")
            text_lines = text_file.readlines()
            count = 0
            for line in text_lines:
                nf,ne = os.path.splitext(saving_folder)
                x1,y1,x2,y2,anno = [int(s) for s in line.strip().split()]
                cropim = im.crop((x1,y1,x2,y2))
                count = count+1

print(nf+str(anno)+"_"+f+"_"+str(annotate(anno))+"_"+str(count))

cropim.save(nf+str(anno)+"_"+f+"_"+str(annotate(anno))+"_"+str(count)+".jpeg
", "jpeg")
```

The output for open circuit defect extraction from the first group are as follows:



Similar insights can be observed when iterating through other groups of dataset. This helps in understanding the shape and size of the bounding boxes from the annotation files.

Image Preprocessing

It is essential to pre-process the image in order to delete any kind of noise or unwanted detailing in the dataset. This helps in better evaluation as well as conducting mathematical operations or image functions using OpenCV to the images. However, in our case, we require our Machine learning (or deep learning) algorithm to learn every single detail about the data that has been fed to it. If we process the data into something that is not distinguishable for the algorithm, it will not learn all the relevant features that are necessary to output a valid and confident prediction. In other words, here preprocessing the data for smoothening the surface, or reducing size, or even applying gaussian filter (let's say) will act as an unwanted noise, instead of helping us for better results.
So, what can be done?

Preparing images for training

Current folder structure:

```
└───PCBData
    ├───group00041
    │   ├───00041
    │   ├───00041_not
    ├───group12000
    │   ├───12000
    │   └───12000_not
    ├───group12100
```

```
     │     ├───12100
     │     └───12100_not
     ├───group12300
     │     ├───12300
     │     └───12300_not
     ├───group13000
     │     ├───13000
     │     └───13000_not
     ├───group20085
     │     ├───20085
     │     └───20085_not
     ├───group44000
     │     ├───44000
     │     └───44000_not
     ├───group50600
     │     ├───50600
     │     └───50600_not
     ├───group77000
     │     ├───77000
     │     └───77000_not
     ├───group90100
     │     ├───90100
     │     └───90100_not
     └───group92000
           ├───92000
           └───92000_not
```

The desired folder structure which is an entire repository of all the training images and validation-cross-validation images:

```
└───Dataset
        └───PCBData
              ├───images
              ├───labels
              └───validation
```

Where `/images` contain 1200 test images, `/labels` contain 1500 annotation files, and `/validation` contains 300 test images which are not taken for training the dataset. That means, `/labels` will have all the annotation files corresponding to images present in `/images` and `/validation` directories.

Processing the folder structure

This has been obtained by writing the following python script:

```python
import glob
import shutil
import os

folder_names = [00041, 12000, 12100, 12300, 13000, 20085, 44000, 50600,
77000, 90100, 92000]

for folder_name in folder_names:
    src_dir = f"Dataset/PCBData/group{folder_name}/{folder_name}"
    dst_dir = f"Dataset/PCBData/images"
    for jpgfile in glob.iglob(os.path.join(src_dir, "*_test.jpg")):
        shutil.copy(jpgfile, dst_dir)
    for txtfile in glob.iglob(os.path.join(src_dir, "*.txt")):
        shutil.copy(txtfile, dst_dir)
```

Creating the `/validation` directory is not mandatory, however, is often recommended to avoid testing the model on the trained dataset itself whilst training.

Why change the folder structure?

In order to answer this question, let us understand how were the folders grouped in the first place. The images were grouped according to the contradiction in the novel module, termed Group Pyramid Pooling by the author. This is assumed to be categorized into the number of pixels that are being operated while feature scaling the neural network model.
However, this distinction is redundant for us because the model architecture we are going to use will be a lot deeper than the one the author of DeepPCB has used, hence we will be able to extract finer features enabling us to carry out experiments regarding the efficiency and effectiveness of our model.
Therefore, it will be much more efficient for us to just make two folders namely `/images` and `/labels`. This will help our server/local computer to recognize where all the images and their corresponding labels are located onto which we need to train our model. And just like that, we have prepared our dataset which can now be trained further.

Experiment Setup

Deployment using FastAPI

Install the FastAPI module and requirements

Run the following command to install all the requirements and application-specific dependencies to build a Restful API using FastAPI.

```
pip install pip-tools
pip-compile requirements.in
```

This will generate a requirements.txt file in the same directory.

```
pip install requirements.txt
```

Folder structure

```
├───main.py
├───requirements.in
├───requirements.txt
├───pcb-fault-detection
│   ├───model
│   └───yolov5
│       ├───models
│       │   └───hub
│       └───utils
│           ├───aws
│           ├───flask_rest_api
│           ├───google_app_engine
│           └───loggers
│               └───wandb
├───training data
└───yolov5
    ├───.github
    │   ├───ISSUE_TEMPLATE
    │   └───workflows
    ├───data
    │   ├───hyps
    │   ├───images
    │   └───scripts
    ├───models
    │   ├───hub
    │   └───__pycache__
    ├───runs
    │   └───train
    │       ├───exp
    │           └───weights
    ├───utils
    │   ├───aws
    │   ├───docker
    │   ├───flask_rest_api
    │   ├───google_app_engine
    │   ├───loggers
    │   │   ├───wandb
    │   │   │   └───__pycache__
    │   │   └───__pycache__
    │   └───__pycache__
    └───__pycache__
```

Modifications based on our requirements

This folder structure can be duplicated using the following commands:

```
git clone https://github.com/keivalya/FSM-INT-2022.git
```

Main change that needs to be carried out is in the `Codes/pcb-fault-detection/model` folder.
Update the PyTorch model (`best.pt`) present inside the directory with your created model.

Deployment

We use `gunicorn` in order to deploy our FastAPI, as it is easy to use, has tons of support and is highly compatible cross-platform.

```
gunicorn -w 4 -k uvicorn.workers.UvicornWorker main:app
```

If all the requirements are met properly, our application will be successfully deployed locally.

Deployed web application

https://pcb-fault-detection.herokuapp.com/docs#

Algorithm Explanation

In this section, we first introduce the overall structure of the network. Then we introduce in detail the details of our modified classifier and evaluation metrics of the dataset.

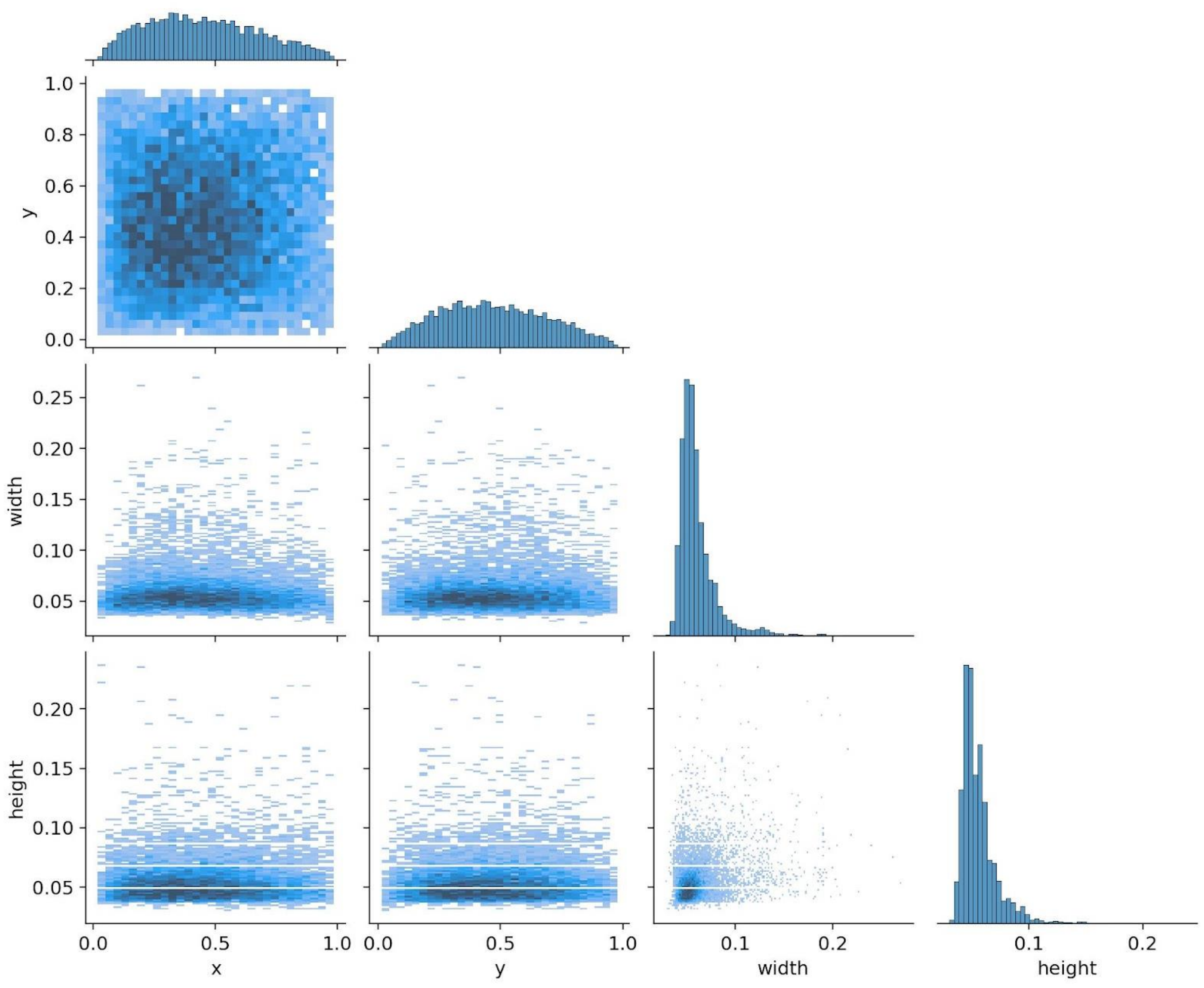
YOLOv5

YOLOv5 has inherited the advantages of YOLOv4, namely adding SPP-NET, modifying the SOTA method, and putting forward new data enhancement methods, such as Mosaic training, Self Adversary training (SAT), and multi-channel feature replacing FPN fusion with PANet.

Dataset Introduction

Our data set is 1500 images, each of which contains annotated PCB defects including positions of the 6 most common types of PCB Defects (open, short, mouse bite, spur, pinhole and spurious copper). The dataset has been adopted for research purposes from https://github.com/tangsanli5201/DeepPCB.

The original dimensions of each image are around 16k × 16k pixels. They are then divided into several 640 × 640 pixels sub-images using a cropping process, then aligned using template matching methods.



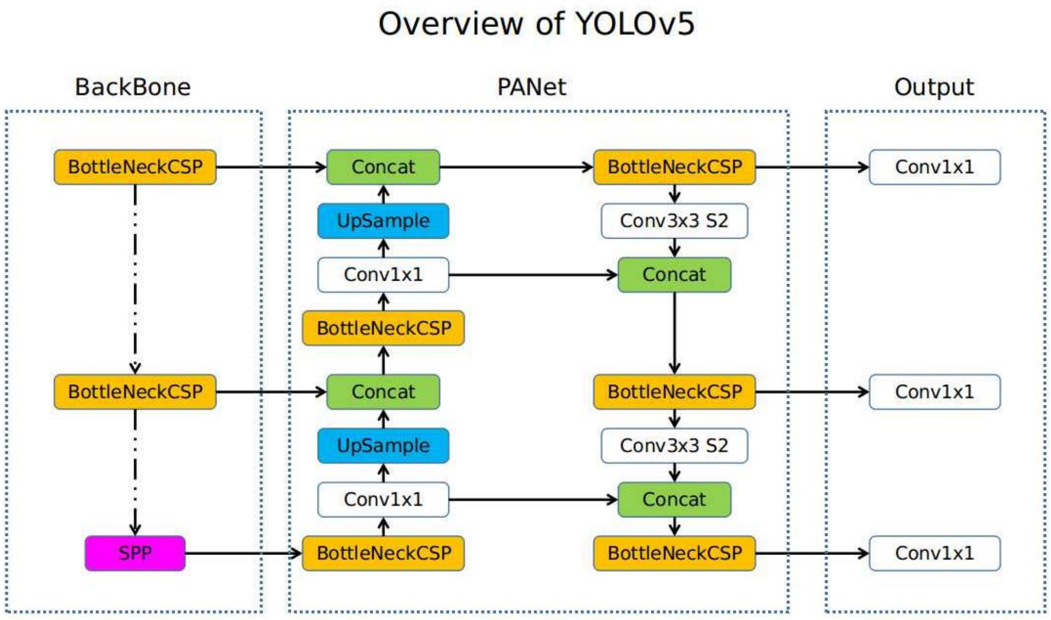
System and Hyperparameter details

Our operating system is ubuntu20.04, using the Pytorch framework, and training and testing on an i5 8.0 GB RAM, 4GB Graphics card and CUDA CPU.

Parameter	Value
Learning rate	0.01
Learning rate decay	0.999
Learning rate decay step	1
Weight rate decay	5e-4
Momentum	0.937
Batch size	16
Number of iterations	100

Model Structure

```
                 from  n    params  module                                  arguments
  0                -1  1      3520  models.common.Conv                      [3, 32, 6, 2, 2]
  1                -1  1     18560  models.common.Conv                      [32, 64, 3, 2]
  2                -1  1     18816  models.common.C3                        [64, 64, 1]
  3                -1  1     73984  models.common.Conv                      [64, 128, 3, 2]
  4                -1  2    115712  models.common.C3                        [128, 128, 2]
  5                -1  1    295424  models.common.Conv                      [128, 256, 3, 2]
  6                -1  3    625152  models.common.C3                        [256, 256, 3]
  7                -1  1   1180672  models.common.Conv                      [256, 512, 3, 2]
  8                -1  1   1182720  models.common.C3                        [512, 512, 1]
  9                -1  1    656896  models.common.SPPF                      [512, 512, 5]
 10                -1  1    131584  models.common.Conv                      [512, 256, 1, 1]
 11                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 12           [-1, 6]  1         0  models.common.Concat                    [1]
 13                -1  1    361984  models.common.C3                        [512, 256, 1, False]
 14                -1  1     33024  models.common.Conv                      [256, 128, 1, 1]
 15                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 16           [-1, 4]  1         0  models.common.Concat                    [1]
 17                -1  1     90880  models.common.C3                        [256, 128, 1, False]
 18                -1  1    147712  models.common.Conv                      [128, 128, 3, 2]
 19          [-1, 14]  1         0  models.common.Concat                    [1]
 20                -1  1    296448  models.common.C3                        [256, 256, 1, False]
 21                -1  1    590336  models.common.Conv                      [256, 256, 3, 2]
 22          [-1, 10]  1         0  models.common.Concat                    [1]
 23                -1  1   1182720  models.common.C3                        [512, 512, 1, False]
 24      [17, 20, 23]  1    229245  models.yolo.Detect                      [80, [[10, 13, 16,
30, 33, 23], [30, 61, 62, 45, 59, 119], [116, 90, 156, 198, 373, 326]], [128, 256, 512]]
                  Model summary: 270 layers, 7235389 parameters, 7235389 gradients
```



source: https://github.com/ultralytics/yolov5/issues/280

The network structure of YOLOv5 is divided into three parts, backbone, neck, and output. In the backbone, the input image with 640×640×3 resolution goes through the Focus structure. Using the slicing operation, it first becomes a 320×320×12 feature map, and then after a convolution operation of 32 convolution kernels, it finally becomes a 320×320×32 feature map. The CBL module is a basic convolution module. A CBL module represents Conv2D + BatchNormal + LeakyRELU.

The BottleneckCSP module mainly performs feature extraction on the feature map, extracting rich information from the image. Compared with other large-scale convolutional neural networks, the BottleneckCSP structure can reduce gradient information duplication in the convolutional neural networks' optimization process. Its parameter quantity occupies most of the parameter quantity of the entire network. By adjusting the width and depth of the BottleneckCSP module, four models with different parameters can be obtained, namely YOLOv5s, YOLOv5m, YOLOv5l, and YOLOv5x. The SPP module mainly increases the receptive field of the network and acquires features of different scales.
YOLOv5 also adds a bottom-up feature pyramid structure based on the FPN structure. With this combination operation, the FPN layer conveys strong semantic features from top to bottom, and the feature pyramid conveys robust positioning features from the bottom up. Combine feature aggregation from different feature layers to improve the network's ability to detect different scales' targets. At the end of the figure, output the classification results and object coordinates.

Metric Explanation

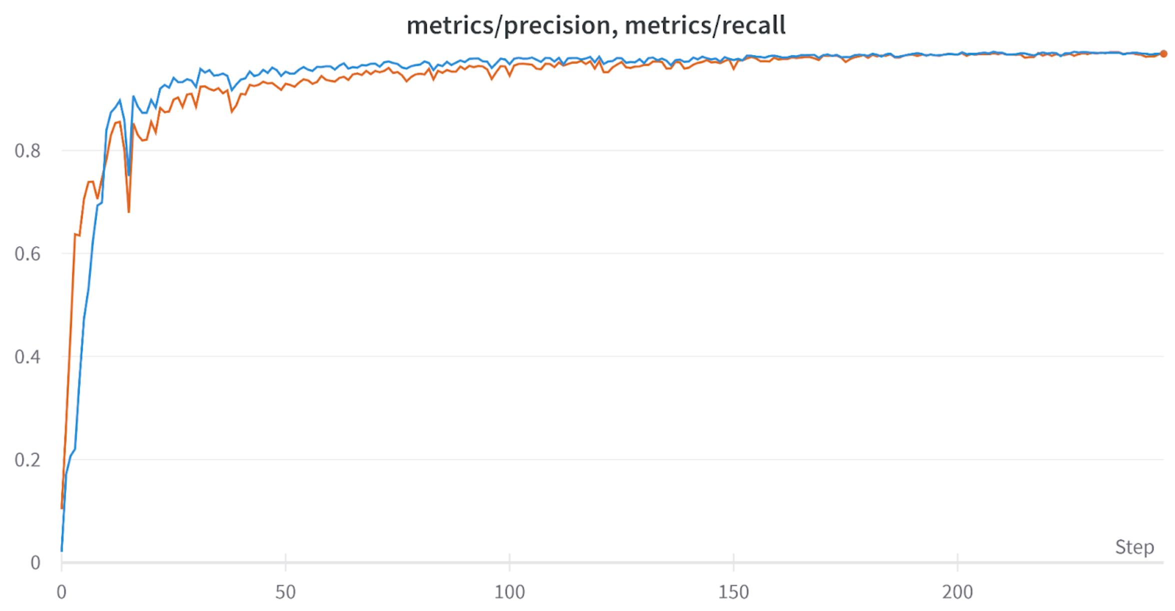
The average precision mean mAP, precision and recall are used to describe the experimental outcomes. To begin, we determine the precision rate and recall rate for each category of an object using the following formula.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

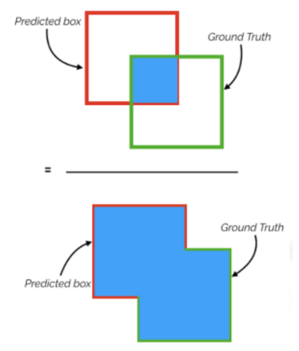
$$AP = \int_0^1 precision(recall)$$

Where TP stands for true positive and FP for false positive; FN stands for false-negative; mAP is the average of all categories' APs, while AP is the average accuracy for a particular category.

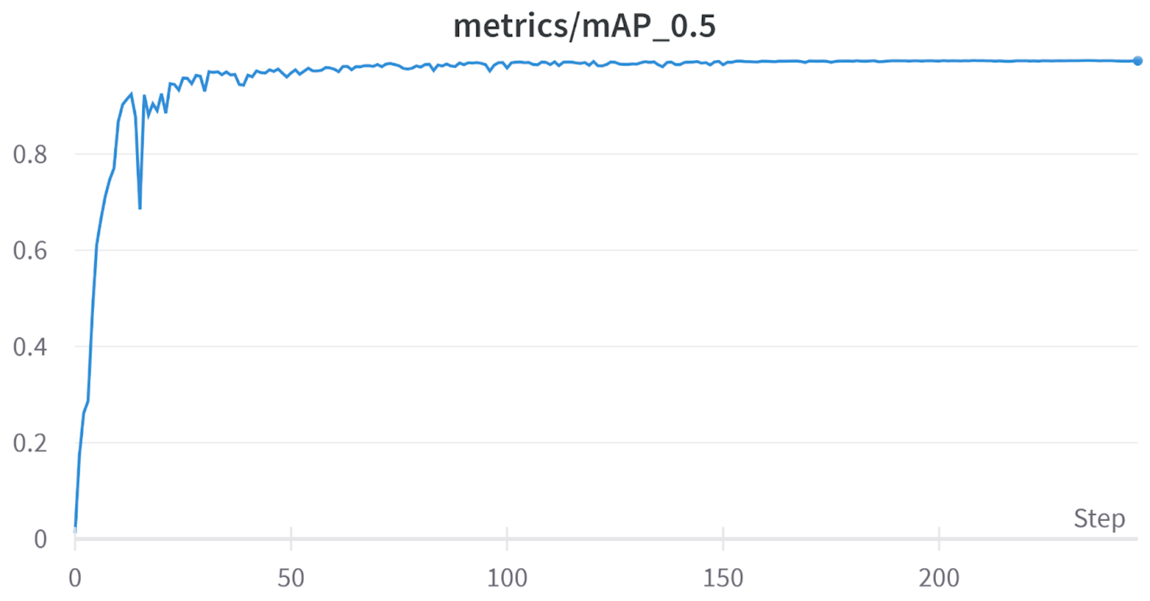


We evaluate the mAP averaged for IoU ∈ [0.5 : 0.05 : 0.95].For each bounding box, we measure the overlap between the predicted bounding box and the ground truth bounding box. This is measured by IoU (intersection over union). For object detection tasks, we calculate Precision and Recall using the IoU value for a given IoU threshold.

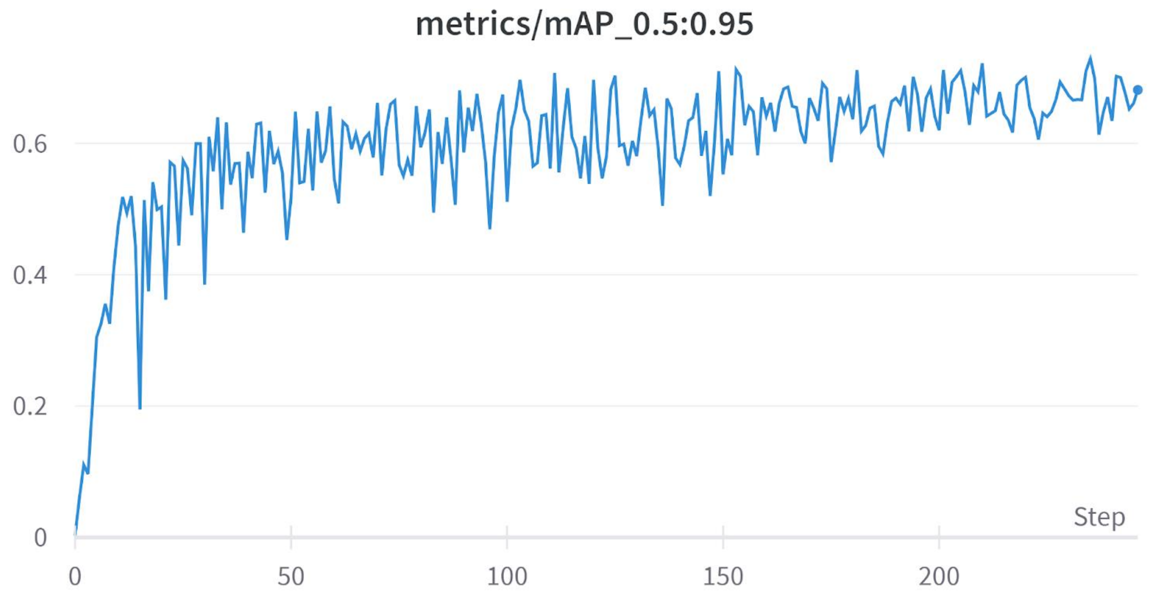
$$Intersection\ of\ \ Union\ (\ IoU) \ = \frac{Area\ of\ \ Overlap}{Area\ of\ \ Union}$$



For instance, if the IoU threshold is 0.5, and the IoU value for a prediction is 0.7, then we classify the prediction as True Positive (TF). On the other hand, if IoU is 0.3, we classify it as a False Positive (FP).
In our detection model, we assume that the confidence score threshold is 0.5 and the IoU threshold is also 0.5. So we calculate the AP at the IoU threshold of 0.5.



Graphical representation of mAP:0.5 across various epochs/steps
It can be found that in the initial training phase, as the training time increases, the model increases rapidly. We can even see a significant downfall and a steep shortcoming in precision as well as recall at the 15th epoch.

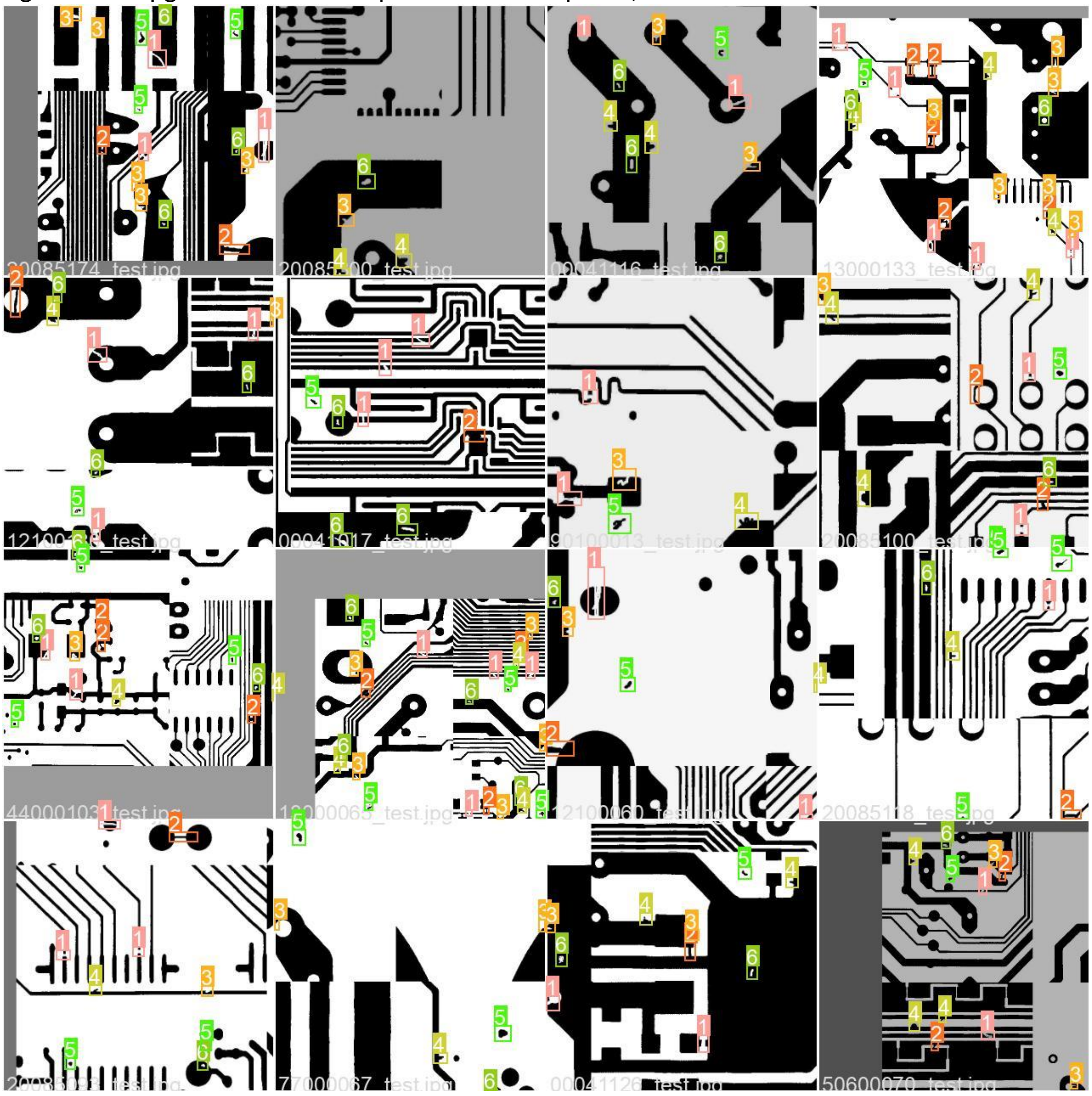


Graphical representation of mAP_0.5:0.95 across all the epochs

Here it is observed that initially, mAP0.5:0.95 is extremely inconsistent, however as the epochs increase, we can see narrower variations in the metrics. This represents the platonic convergence of the criterion and increase in its stability.

Results

After training the model for 24.058 hours and stopping at 336 epochs due to receiving no significant upgrades over the previous 100 epochs, results can be seen as follows.



Future Work

Running on the test dataset, results show that the algorithm needs to be improved in the following aspects:

1. Increase detection threshold

   It can be observed that "spur" is detected even when there is an extremely low or negligible possibility of it in that segment. It may be due to resolution error wherein a little discrepancy in the pixels because of quality reduction in the preprocessing stage. This can be resolved by increasing the threshold for detection.

2. Develop robust algorithms

   Using a hybrid approach, instead of a purely YOLO approach may help us acquire better results in lower processing time. For example, preprocessing the image to reduce the pixel complexity will not only improve speed but also the overall accuracy by performing only on specifically identified segments of the image.

Conclusion

The PCB Fault Detection algorithm is able to detect defective PCBs rapidly using the YOLOv5 method. Using FastAPI's SwaggerUI, we can generate an elegant user interface where the algorithm can be deployed. For real-time object detection, a constant image feed from the camera can be set-up pointing towards the PCB that needs to be analysed.