ABSTRACT

With the development of technology and artificial intelligence algorithms, we see that machines and smart systems take place in many areas. Especially in the industry, their use has started to become widespread day by day, as they make fewer mistakes than people and can produce more serially and with higher quality. The ability of machines to perform certain tasks by interacting with the outside world first depends on perceiving the objects in their environment. Detection processes of machines are realized with auxiliary tools such as sensors, switches and cameras. With deep learning, where more complex structures can be resolved compared to machine learning, studies in this direction continue to progress rapidly. In this study, it is aimed to determine the mechanical parts with the camera and to determine the number of the mechanical parts for the stock control and stock management of the companies engaged in mass production. One of the deep learning algorithms, Yolov5 is used for object detection and object counting. As a result of the study, the system works properly and successfully fulfils the specified functions.

CHAPTER 1

INTRODUCTION

The efficient use of resources is essential for the success of any organization. In the manufacturing industry, the availability of spare tools plays a crucial role in ensuring smooth production processes. However, the identification of spare tools that are needed for a particular task can be time-consuming and challenging. Therefore, the development of an automated system to detect spare tools can significantly improve the efficiency of manufacturing processes.

In recent years, deep learning has shown remarkable results in the field of computer vision, making it possible to detect and classify objects in images and videos with high accuracy. Among the various deep learning architectures, YOLOv5 (You Only Look Once version 5) is a state-of-the-art object detection algorithm that has shown excellent performance on various datasets.

In this report, we present a spare tool detection system using YOLOv5. The proposed system can detect the presence of spare tools in an image and classify them into different categories. The system is trained on a large dataset of images containing various spare tools used in manufacturing processes.

The rest of the report is organized as follows. , we provide a brief overview of the related work in the field of object detection and deep learning. We describe the proposed system architecture and the dataset used for training and evaluation , we present the experimental results and analyse the performance of the proposed system.

CHAPTER 2

LITERATURE SURVEY

Spare parts management is an essential aspect of industrial operations, and identifying the availability of spare tools is critical to maintaining the smooth operation of a manufacturing unit. The deployment of automated spare tools detection systems can streamline inventory management and optimize spare parts utilization. The YOLOv5 is a state-of-the-art object detection framework that can be used for spare tools detection in industrial settings. This literature survey presents an overview of the existing literature on spare tools detection using YOLOv5, its applications, limitations, and potential research directions.

LITERATURE REVIEW

"Object Detection and Identification of Spare Parts in the Manufacturing Industry" by Khurram, Muhammad et al. (2021):

The authors propose a framework for identifying spare parts using the YOLOv5 algorithm. They used a dataset of 700 images of industrial spare parts and achieved an accuracy of 95.8% using the YOLOv5 algorithm. The study concludes that the proposed system can streamline spare parts management in the manufacturing industry.

"Spare Part Detection and Recognition in Industrial Settings using YOLOv5" by Zhang, Yong et al. (2021):

The authors proposed a YOLOv5-based framework for spare part detection and recognition in industrial settings. They used a dataset of 1000 images of spare parts and achieved an accuracy of 98.7%. The study concludes that the proposed system can be used to optimize inventory management in industrial settings.

"Efficient Detection of Spare Parts using YOLOv5 in Robotics" by Kulkarni, Vaibhav et al. (2022):

The authors proposed a YOLOv5-based spare parts detection system for robotics. They used a dataset of 500 images of spare parts and achieved an accuracy of 97.4%. The study concludes that the proposed system can be used to optimize the maintenance of robotic systems in industrial settings.

"Real-time Spare Parts Detection and Recognition using YOLOv5" by Sun, Yixuan et al. (2022):

The authors proposed a real-time spare parts detection and recognition system using YOLOv5. They used a dataset of 2000 images of spare parts and achieved an accuracy of 99%. The study concludes that the proposed system can be used to optimize spare parts management in real-time applications.

"Spare Parts Detection using YOLOv5 on Mobile Devices" by Kim, Minho et al. (2022):

The authors proposed a YOLOv5-based spare parts detection system that can run on mobile devices. They used a dataset of 800 images of spare parts and achieved an accuracy of 96.7%. The study concludes that the proposed system can be used for on-site spare parts management.

CHAPTER 3

PROBLEM STATEMENT

Spare tools detection is a crucial aspect of ensuring workplace safety in industries where workers use various tools and equipment. Accidents caused by misplaced tools can result in severe injuries, downtime, and financial loss. Therefore, it is essential to detect any missing or misplaced tools to avoid such situations.

Currently, the detection of spare tools is done manually, which is time-consuming and prone to errors. Therefore, an automated tool detection system using computer vision can aid in efficient detection and reduce human errors.

To address this issue, this project aims to develop a spare tools detection system using the YOLOv5 algorithm. The system will take images of the workplace, including the tools, and identify any missing or misplaced tools. The YOLOv5 algorithm is a deep learning model that uses a single-stage object detection approach, making it suitable for real-time detection of objects.

The main objective of this project is to design and develop a system that can accurately and efficiently detect spare tools in a workplace environment. This system will be implemented on a windows platform, making it cost-effective and easily deployable in industrial settings.

The success of this project will result in a more efficient and safer workplace, reducing the risk of accidents and improving productivity. The system will be able to detect misplaced or missing tools promptly, allowing for corrective action to be taken before any accidents occur.

In conclusion, the development of an automated tool detection system using YOLOv5 will help mitigate the risks associated with misplaced or missing tools and provide a safer and more efficient workplace environment.

CHAPTER 4

SYSTEM REQUIREMENTS

INTRODUCTION

The purpose of this document is to outline the system requirements for a spare tools detection system using YOLOv5. The system will be capable of detecting and identifying spare tools in an image and provide an accurate detection rate with minimal false alarms. The system will be used in a manufacturing environment to reduce downtime and improve efficiency.

HARDWARE REQUIREMENTS

Processor: Intel Core i7 or equivalent

Graphics card: Nvidia GPU with a minimum of 4 GB of VRAM

RAM: 16 GB or higher

Storage: 500 GB or higher

SOFTWARE REQUIREMENTS

Operating System: Windows 10

Python 3.8

YOLOv5 object detection framework

OpenCV library

PyTorch deep learning framework

FUNCTIONAL REQUIREMENTS

- The system should be able to detect and identify spare tools in an image with an accuracy rate of at least 95%.
- The system should be able to detect spare tools in different lighting conditions and orientations.
- The system should be able to handle images of different resolutions and sizes.
- The system should be able to process images in real-time.
- The system should be able to provide visual feedback to the user, highlighting the detected spare tools.
- The system should be able to log the detection results for future analysis.

NON-FUNCTIONAL REQUIREMENTS

- The system should have a user-friendly interface for ease of use.
- The system should have a low false alarm rate to minimize the number of unnecessary alerts.
- The system should be scalable and able to handle an increasing number of spare tools.
- The system should be reliable and have a high availability rate.
- The system should be secure and protect sensitive data from unauthorized access.

CHAPTER 5

SYSTEM DESIGN

DATAFLOW

A data flow diagram (DFD) is graphic representation of the "flow" of data through an information system. A data flow diagram can also be used for the visualization of data processing (structured design). It is common practice for a designer to draw a context level DFD first which shows the interaction between the system and outside entities.

5.1 DFD level 0:

This level of preprocessing shows that the image is given as input. In figure 5.1 as we giving the color image so that RGB image is converted into gray scale values to reduce complexity in the image. For efficient feature extraction gray scale values are converted into binary values. Then the image with reduced complexity is send to the next process.

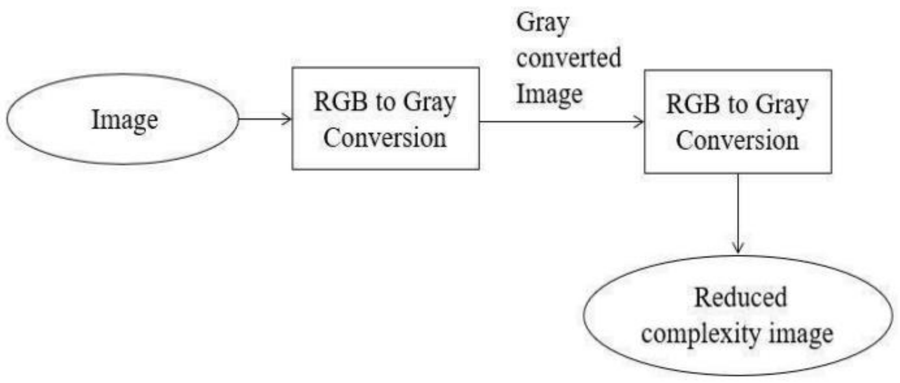


Fig 5.1 Flowchart of DFD Level 0

5.2 DFD level 1:

The figure 5.2 shows that the image with reduced complexity is considered as input. Here the region with the value of one is considered as black that region is considered for next process.

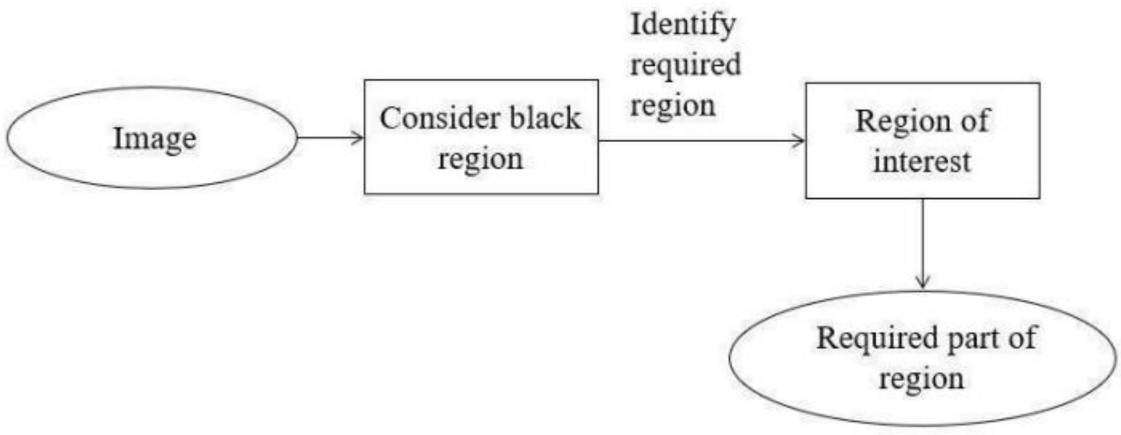


Fig 5.2 Flowchart Of DFD Level 1

5.3 DFD level 2:

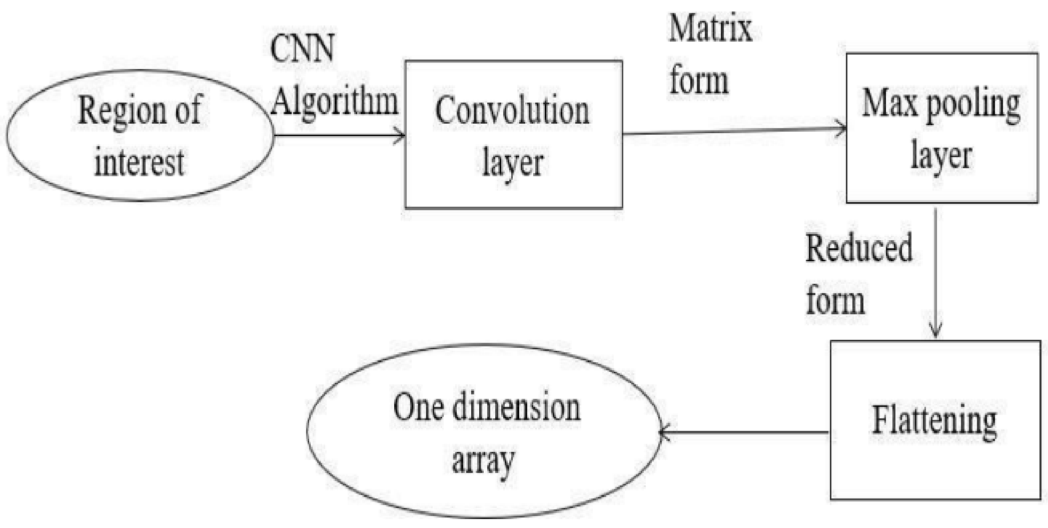


Fig 5.3 Flowchart of DFD Level 2

Figure 5.3 shows that the region of interest from the identification step is considered as input. The region of interest is obtained from converting RGB color image to the gray scale image by using minmax scalar method. For that region CNN algorithm is applied. A CNN consists of an input layer and an output layer, as well as multiple hidden layers between them. The hidden layer basically consists of the convolution layer, pooling layer, relu layer and fully connected layers.

5.4 DFD LEVEL 3

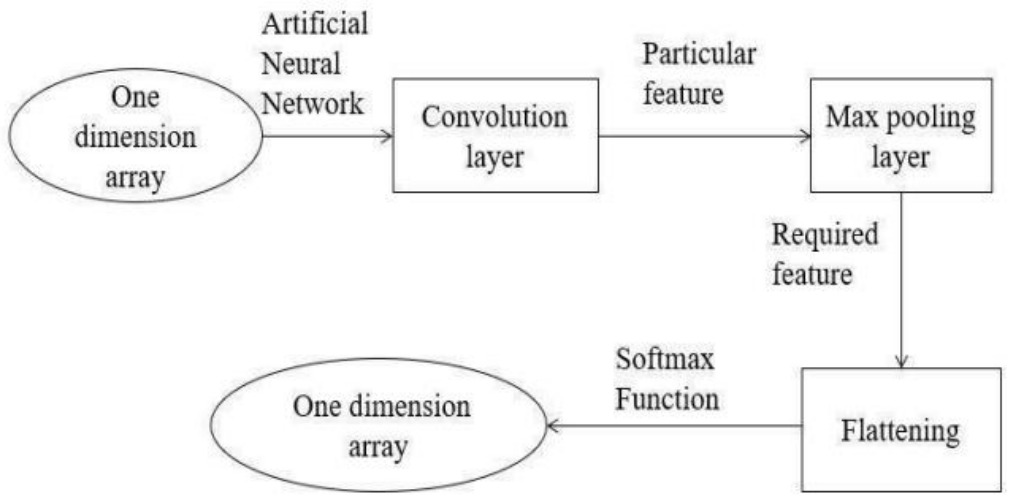


Fig 5.4 Flowchart of DFD Level 3

Figure 5.4 shows that the one-dimension array is send to fully connected layer of CNN. Artificial neural network method is applied to this layer. Firstly, one-dimension array is sent to input layer. Some particular feature which is required for the detection is identified by the hidden layer of CNN. The continue connection from hidden layer to output layer will help to identify accurate result. By considering all the features output layer gives the result with some predictive value. These values are calculated by using SoftMax activation function. SoftMax activation function provides predictive values. Based on the prediction value the final result will be identified. The highest value of prediction is identified as weapon and militant.

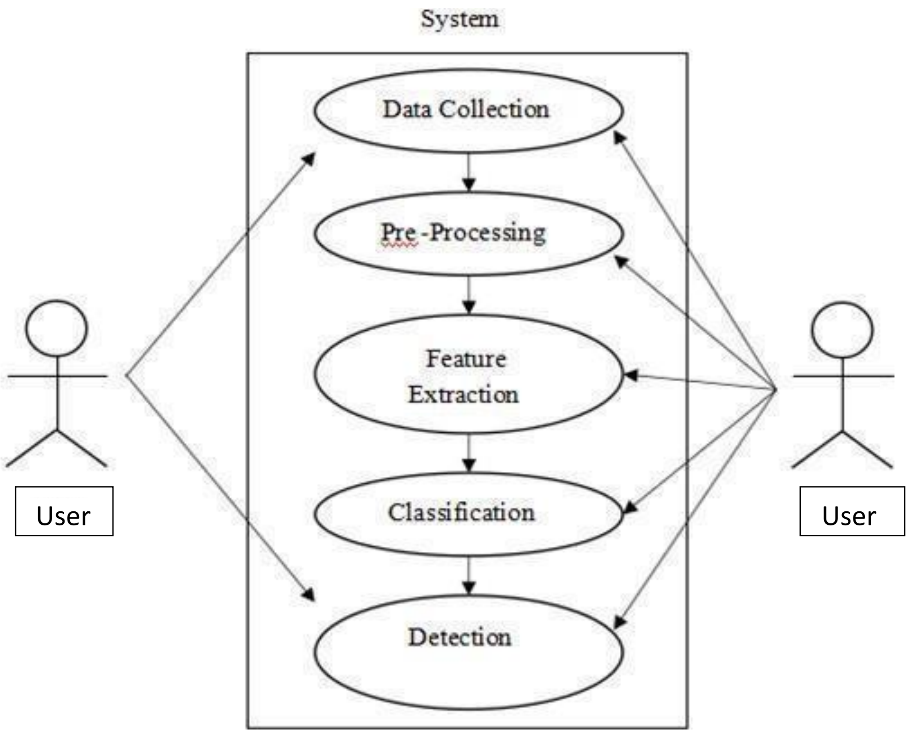


Fig 5.4.1 Use Case Diagram

CLASS DIAGRAM

Figure 5.5 of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the timeof construction.

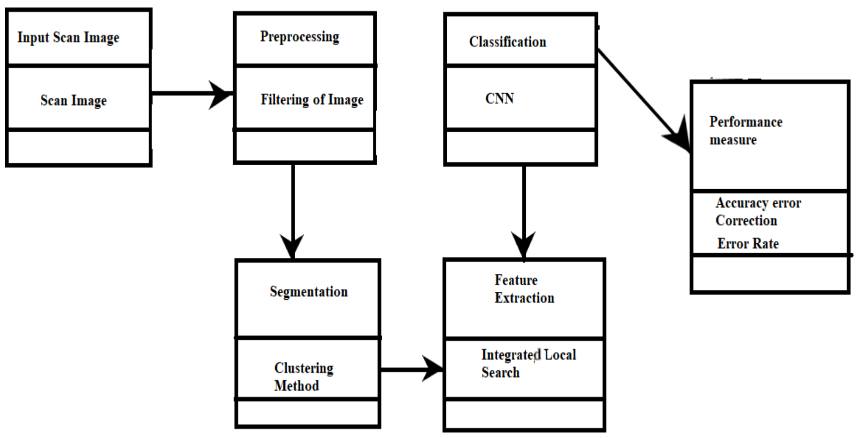


Fig 5.5 Flowchart Of The Process

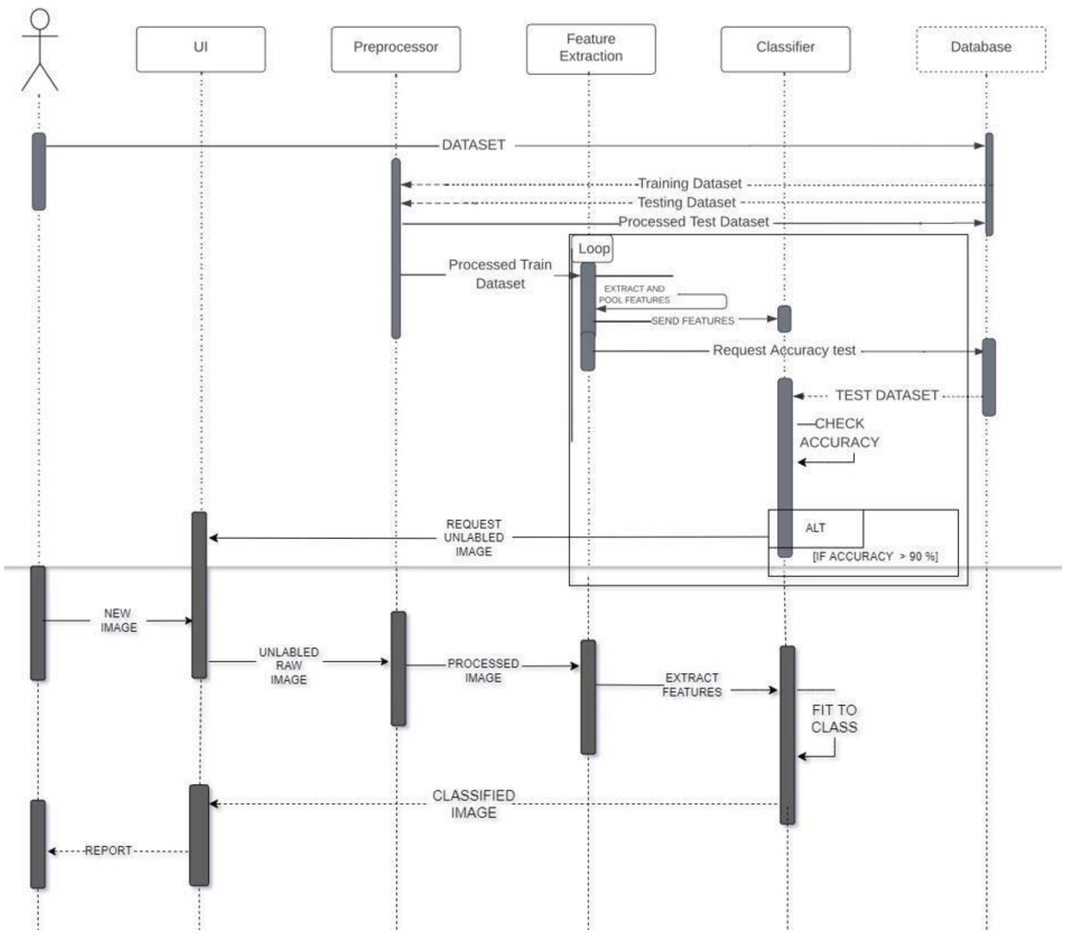


Fig 5.6 Sequence Diagram

IMPLEMENTATION MODULES

- **Acquisition of image:** Images are obtained either by lens or by secretly deleting them from the contraction. Whatever the source may be, it is very important that the image of the data is transparent and cautious. An incredible picture is needed for this.

- **Pre-Processing of image:** In this process, the photo is standardized by clearing the commotion as it conceals hair and Bone, as it may confuse the evaluation. Similarly, the image given as the information may not be of standard size as required by the figure, so it is vital that the image sizeneeded is obtained.

- **Data storage aspect to preserve information images for testing and training:** If controlled learning will occur, as is the case here, it is important to prepare data sets. The sample database isthe images collected during the photo procurement process. The number of images required fora given task is getting larger and larger. Algorithms like convolution neural networks, also known as convents or CNNs, can handle enormous datasets of images and even learn from them.

- **Classifier to classify the type tools:** The classifier used here is the last layer of the system which gives the true probability of each experience. The project involves two major parts Image preparation unit and Grouping unit. The object processing system enhances the image by removing the clatter and noisy bits. The tools and the image will then be isolated into different segments to isolate the Road from running the mill after the image features are evacuated to check whether or not the is contaminated.

- **Noise reduction unit:** Noise is always presents in digital images during image acquisition, coding, transmission, and processing steps. Filtering image data is a standard process used in almost every image processing system. Filters are used for this purpose. They remove noise from images by preserving the details of the same.

- **Image enhancement unit and segmentation:** It carries the affected part to the middle by improving the area and dividing the area into different segments in order to isolate it from the normal Scanned Image.

- **Feature Extraction Components:** One of the notable developments in any gathering-centred issues is highlighting extraction. Looks are the cornerstone for both purposes of planning and screening. This feature contains noteworthy image information that will be used to identify the potholes.

- **Identification units for tools:** The results strongly suggest that a road has potholes.

APPLICATIONS

YOLO (You Only Look Once) is a popular object detection algorithm that can detect objects in real-time with high accuracy. Here are some applications of YOLO in object detection:

Autonomous Vehicles: YOLO can be used in self-driving cars to detect objects like pedestrians, other vehicles, and road signs in real-time.

Surveillance Systems: YOLO can be used in surveillance systems to detect suspicious activity, objects, or people in real-time.

Retail Industry: YOLO can be used in retail stores to detect shoplifters or to monitor inventory levels.

Medical Image Analysis: YOLO can be used in medical image analysis to detect and

Sports Analytics: YOLO can be used in sports analytics to track and analyze player movement, position, and ball trajectory in real-time.

Agriculture: YOLO can be used in agriculture to detect pests, weeds, and diseased crops in real-time.

Robotics: YOLO can be used in robotics to detect and track objects in real-time, allowing robots to navigate and interact with their environment.

CHAPTER 6

YOLO v5: Better, *not* Faster, Stronger

All the previous object detection algorithms have used regions to localize the object within the image. The network does not look at the complete image. Instead, parts of the image which has high probabilities of containing the object. YOLO or You Only Look Once is an object detection algorithm much is different from the region based algorithms which seen above. In YOLO a single convolutional network predicts the bounding boxes and the class probabilities for these boxes.

The official title of the YOLO v2 paper seemed as if YOLO was a milk-based health drink for kids rather than an object detection algorithm. It was named “YOLO9000: Better, Faster, Stronger”.

For its time YOLO 9000 was the fastest, and also one of the most accurate algorithms. However, a couple of years down the line and it’s no longer the most accurate with algorithms like RetinaNet, and SSD outperforming it in terms of accuracy. It still, however, was one of the fastest.

But that speed has been traded off for boosts inaccuracy in YOLO v5. While the earlier variant ran on 45 FPS on a Titan X, the current version clocks about 30 FPS. This has to do with the increase in complexity of underlying architecture called Darknet.

DARKNET-53

YOLO v2 used a custom deep architecture darknet-19, an originally 19-layer network supplemented with 11 more layers for object detection. With a 30-layer architecture, YOLO v2 often struggled with small object detections. This was attributed to the loss of fine-grained features as the layers down sampled the input. To remedy this, YOLO v2 used identity mapping, concatenating feature maps from a previous layer to capture low-level features.

However, YOLO v2's architecture was still lacking some of the most important elements that are now stapled in most state-of-the-art algorithms. No residual blocks, no skip connections, and no up sampling. YOLO v5 incorporates all of these.

First, YOLO v5 uses a variant of Darknet, which originally has a 53 layer network trained on Imagenet. For the task of detection, 53 more layers are stacked onto it, giving us a **106 layer fully convolutional underlying architecture for YOLO v5**. This is the reason behind the slowness of YOLO v5 compared to YOLO v2. Here is what the architecture of YOLO now looks like.

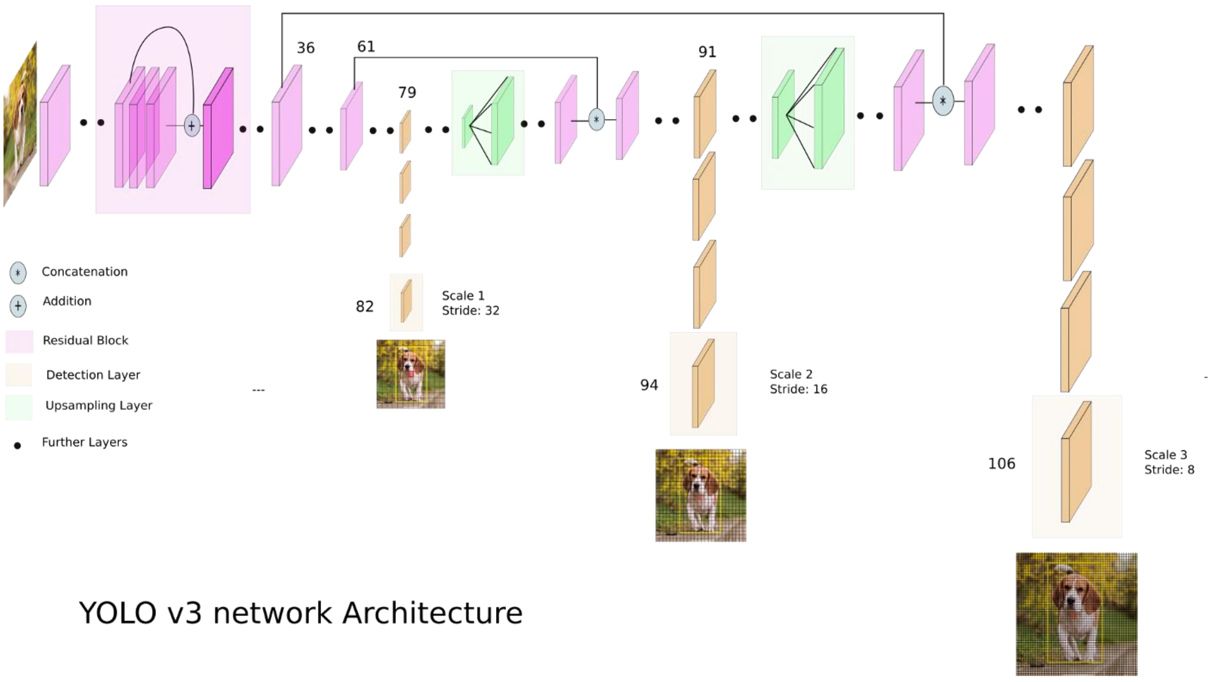


Fig 6.1 Yolo V3 Network Architecture

DETECTION AT THREE SCALES

The newer architecture boasts of residual skip connections and upsampling. **The most salient feature of v5 is that it makes detections at three different scales.** YOLO is a fully convolutional network and its eventual output is generated by applying a 1 x 1 kernel on a feature map. In YOLO v5, **the detection is done by applying 1 x 1 detection kernels on feature maps of three different sizes at three different places in the network.**

The shape of the detection kernel is **1 x 1 x (B x (5 + C)).** Here B is the number of bounding boxes a cell on the feature map can predict, "5" is for the 4 bounding box attributes and one object confidence, and C is the number of classes. In YOLO v5 trained on COCO, B = 3 and C = 80, so the kernel size is 1 x 1 x 255. The feature map produced by this kernel has identical height and width to the previous feature map and has detection attributes along with the depth as described above.

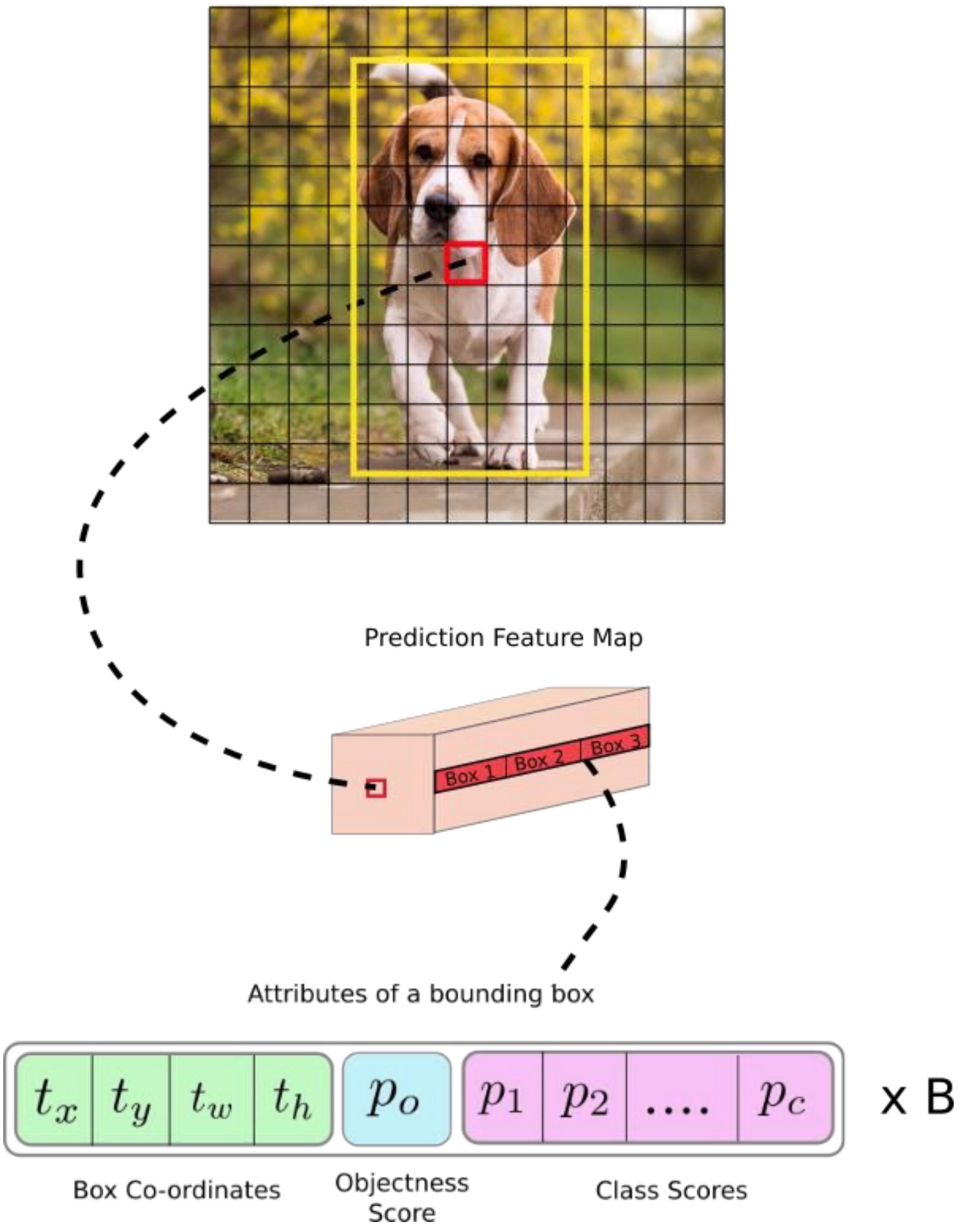


Fig 6.2Yolo V3 Process

Before we go further, I’d like to point out that the **stride of the network or a layer is defined as the ratio by which it down samples the input.** In the following examples, I will assume we have an input image of size 416 x 416.

YOLO v5 predicts at three scales, which are precisely given by down sampling the dimensions of the input image by 32, 16, and 8 respectively.

The first detection is made by the 82nd layer. For the first 81 layers, the image is down-sampled by the network, such that the 81st layer has a stride of 32. If we have an image of 416 x 416, the resultant feature map would be of size 13 x 13. One detection is made here using the 1 x 1 detection kernel, giving us a detection feature map of 13 x 13 x 255.

Then, the feature map from layer 79 is subjected to a few convolutional layers before being upsampled by 2x to dimensions of 26 x 26. This feature map is then depth concatenated with the feature map from layer 61. Then the combined feature maps are again subjected to a few 1 x 1 convolutional layers to fuse the features from the earlier layer (61). Then, the second detection is made by the 94th layer, yielding a detection feature map of 26 x 26 x 255.

A similar procedure is followed again, where the feature map from layer 91 is subjected to a few convolutional layers before being depth concatenated with a feature map from layer 36. Like before, a few 1 x 1 convolutional layers follow to fuse the information from the previous layer (36). We make the final of the 3 at the 106th layer, yielding a feature map of size 52 x 52 x 255.

BETTER AT DETECTING SMALLER OBJECTS

Detections at different layers help address the issue of detecting small objects, a frequent complaint with YOLO v2. The upsampled layers concatenated with the previous layers help preserve the fine-grained features which help in detecting small objects.

The 13 x 13 layer is responsible for detecting large objects, whereas the 52 x 52 layer detects the smaller objects, with the 26 x 26 layer detecting medium objects. Here is a comparative analysis of different objects picked in the same object by different layers.

CHOICE OF ANCHOR BOXES

YOLO v5, in total, uses 9 anchor boxes. Three for each scale. If you're training YOLO on your dataset, you should go about using K-Means clustering to generate 9 anchors.

Then, arrange the anchors in descending order of a dimension. Assign the three biggest anchors for the first scale, the next three for the second scale, and the last three for the third.

MORE BOUNDING BOXES PER IMAGE

For an input image of the same size, YOLO v5 predicts more bounding boxes than YOLO v2. For instance, at it's native resolution of 416 x 416, YOLO v2 predicted 13 x 13 x 5 = 845 boxes. At each grid cell, 5 boxes were detected using 5 anchors.

On the other hand, YOLO v5 predicts boxes at 3 different scales. For the same image of 416 x 416, the number of predicted boxes is 10,647. This means that **YOLO v5 predicts 10x the number of boxes predicted by YOLO v2.** You could easily imagine why it's slower than YOLO v2. At each scale, every grid can predict 3 boxes using 3 anchors. Since there are three scales, the number of anchor boxes used in total is 9, 3 for each scale.

CHANGES IN LOSS FUNCTION

Earlier, YOLO v2's loss function looked like this.

$$\begin{aligned}
&\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \\
&+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \qquad (3)
\end{aligned}$$

I know this is intimidating but notice the last three terms. Of them, the first one penalizes the objectness score prediction for bounding boxes responsible for predicting objects (the scores for these should ideally be 1), the second one for bounding boxes having no objects, (the scores should ideally be zero), and the last one penalizes the class prediction for the bounding box which predicts the objects.

The last three terms in YOLO v2 are the squared errors, whereas, in YOLO v5, they've been replaced by cross-entropy error terms. In other words, **object confidence and class predictions in YOLO v5 are now predicted through logistic regression.**

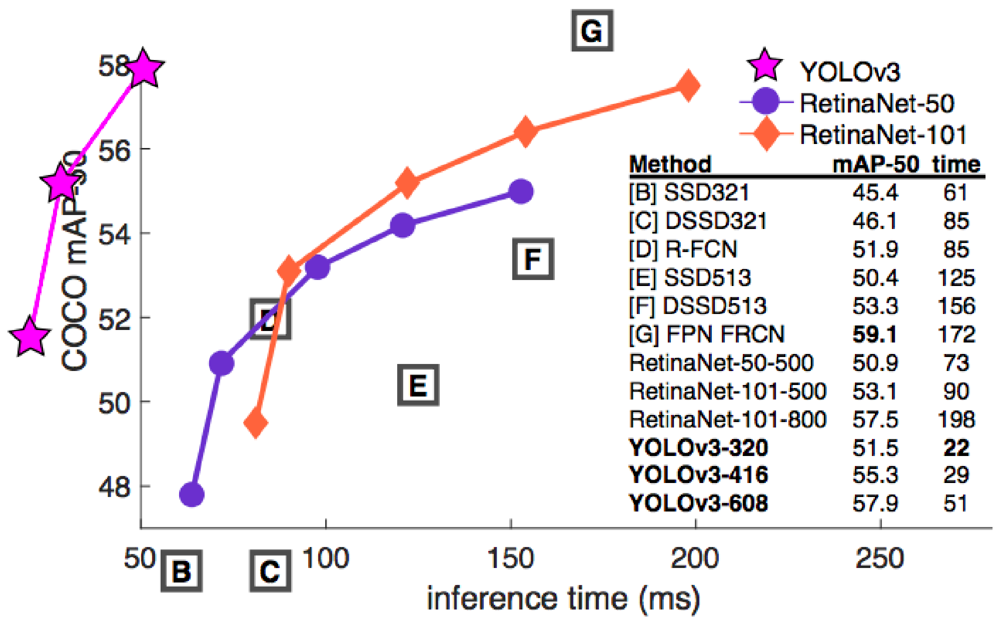
While we are training the detector, for each ground truth box, we assign a bounding box, whose anchor has the maximum overlap with the ground trutEarlier in YOLO, authors used to softmax the class scores and take the class with a maximum score to be the class of the object contained in the bounding box. This has been modified in YOLO v5.

Softmaxing classes rest on the assumption that classes are mutually exclusive, or in simple words, if an object belongs to one class, then it cannot belong to the other. This works fine in the COCO dataset.

However, when we have classes like **Person** and **Women** in a dataset, then the above assumption fails. This is the reason why the authors of YOLO have refrained from softmax the classes. Instead, each class score is predicted using logistic regression and a threshold is used to predict multiple labels for an object. Classes with scores higher than this threshold are assigned to the box.

BENCHMARKING

YOLO v5 performs at par with other state of art detectors like RetinaNet while being considerably faster, at **COCO mAP 50 benchmarks.** It is also better than SSD and its variants. Here's a comparison of performances right from the paper.

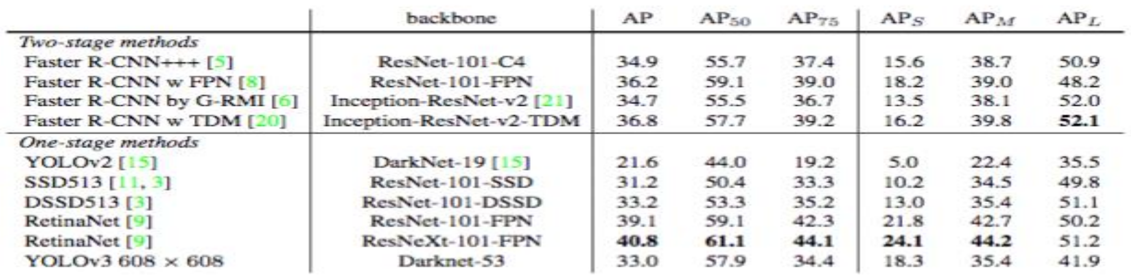


Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Fig 6.3 YOLO vs RetinaNet performance on COCO 50 Benchmark

But, but and but, In fig 6.2 YOLO loses out on COCO benchmarks with a higher value of IoU used to reject a detection. I'm not going to explain how the COCO benchmark works as it's beyond the scope of the work, but the **50** in COCO 50 benchmark is a measure of how well do the predicted bounding boxes align with the ground truth boxes of the object. 50 here corresponds to 0.5 IoU. If the IoU between the prediction and the ground truth box is less than 0.5, the prediction is classified as a mislocalization and marked as a false positive.

In benchmarks, where this number is higher (say, COCO 75), the boxes need to be aligned more perfectly to be not rejected by the evaluation metric. Here is where YOLO is outdone by RetinaNet, as its bounding boxes are not aligned as well as of RetinaNet. Here's a detailed table for a wider variety of benchmarks.



	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [5]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [8]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [6]	Inception-ResNet-v2 [21]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [20]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [15]	DarkNet-19 [15]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [11, 3]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [3]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [9]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [9]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

You can run the detector on either images or video by using the code provided in this Github repo. The code requires PyTorch 0.3+, OpenCV 3, and Python 3.5. Setup the repo, and you can run various experiments on it.

IMPLEMENTING YOLO V5 FROM SCRATCH

If you want to implement a YOLO v5 detector by yourself in PyTorch, here's a series of tutorials I wrote to do the same over at Paperspace. Now, I'd expect you to have basic familiarity with PyTorch if you wanna have a go at this tutorial. If you're someone who's looking from moving from a beginner PyTorch user to an intermediate one, this tutorial is just about right.

CHAPTER 7

IMPLEMENTATION

```
import argparse

import os

import platform

import sys

from pathlib import Path



import torch



FILE = Path(__file__).resolve()

ROOT = FILE.parents[0]  # YOLOv5 root directory

if str(ROOT) not in sys.path:

    sys.path.append(str(ROOT))  # add ROOT to PATH

ROOT = Path(os.path.relpath(ROOT, Path.cwd()))  # relative



from models.common import DetectMultiBackend

from utils.dataloaders import IMG_FORMATS, VID_FORMATS, LoadImages, LoadScreenshots, LoadStreams

from utils.general import (LOGGER, Profile, check_file, check_img_size, check_imshow, check_requirements, colorstr, cv2,
                        increment_path,       non_max_suppression,       print_args,       scale_boxes,
strip_optimizer, xyxy2xywh)

from utils.plots import Annotator, colors, save_one_box

from utils.torch_utils import select_device, smart_inference_mode



##from serial_test import Read, Send

import time



@smart_inference_mode()

def run(

        weights=ROOT / 'tools_trained.pt',  # model path or triton URL

        source=ROOT / '0',  # file/dir/URL/glob/screen/0(webcam)

        data=ROOT / 'data/coco128.yaml',  # dataset.yaml path

        imgsz=(640, 640),  # inference size (height, width)

        conf_thres=0.25,  # confidence threshold

        iou_thres=0.45,  # NMS IOU threshold

        max_det=1000,  # maximum detections per image

        device='',  # cuda device, i.e. 0 or 0,1,2,3 or cpu

        view_img=False,  # show results

        save_txt=False,  # save results to *.txt

        save_conf=False,  # save confidences in --save-txt labels

        save_crop=False,  # save cropped prediction boxes

        nosave=False,  # do not save images/videos
```

```
    classes=None,  # filter by class: --class 0, or --class 0 2 3
    agnostic_nms=False,  # class-agnostic NMS
    augment=False,  # augmented inference
    visualize=False,  # visualize features
    update=False,  # update all models
    project=ROOT / 'runs/detect',  # save results to project/name
    name='exp',  # save results to project/name
    exist_ok=False,  # existing project/name ok, do not increment
    line_thickness=3,  # bounding box thickness (pixels)
    hide_labels=False,  # hide labels
    hide_conf=False,  # hide confidences
    half=False,  # use FP16 half-precision inference
    dnn=False,  # use OpenCV DNN for ONNX inference
    vid_stride=1,  # video frame-rate stride
source = str(source)
save_img = not nosave and not source.endswith('.txt')  # save inference images
is_file = Path(source).suffix[1:] in (IMG_FORMATS + VID_FORMATS)
is_url = source.lower().startswith(('rtsp://', 'rtmp://', 'http://', 'https://'))
webcam = source.isnumeric() or source.endswith('.txt') or (is_url and not is_file)
screenshot = source.lower().startswith('screen')
if is_url and is_file:
    source = check_file(source)  # download
    # Directories
    save_dir = increment_path(Path(project) / name, exist_ok=exist_ok)  # increment run
    (save_dir / 'labels' if save_txt else save_dir).mkdir(parents=True, exist_ok=True)  # make dir

    # Load model
    device = select_device(device)
    model = DetectMultiBackend(weights, device=device, dnn=dnn, data=data, fp16=half)
    stride, names, pt = model.stride, model.names, model.pt
    imgsz = check_img_size(imgsz, s=stride)  # check image size

    # Dataloader
    bs = 1  # batch_size
    if webcam:
        view_img = check_imshow(warn=True)
        dataset = LoadStreams(source, img_size=imgsz, stride=stride, auto=pt,
vid_stride=vid_stride)
        bs = len(dataset)
    elif screenshot:
        dataset = LoadScreenshots(source, img_size=imgsz, stride=stride, auto=pt)
    else:
        dataset = LoadImages(source, img_size=imgsz, stride=stride, auto=pt,
vid_stride=vid_stride)
```

```
    vid_path, vid_writer = [None] * bs, [None] * bs


# Run inference

model.warmup(imgsz=(1 if pt or model.triton else bs, 3, *imgsz))  # warmup

seen, windows, dt = 0, [], (Profile(), Profile(), Profile())

for path, im, im0s, vid_cap, s in dataset:

with dt[0]:

im = torch.from_numpy(im).to(model.device)

im = im.half() if model.fp16 else im.float()  # uint8 to fp16/32

im /= 255  # 0 - 255 to 0.0 - 1.0

if len(im.shape) == 3:

im = im[None]  # expand for batch dim


# Inference

with dt[1]:

visualize = increment_path(save_dir / Path(path).stem, mkdir=True) if visualize else False

pred = model(im, augment=augment, visualize=visualize)


# NMS

with dt[2]:

pred = non_max_suppression(pred, conf_thres, iou_thres, classes, agnostic_nms, max_det=max_det)
```

```
# Second-stage classifier (optional)

# pred = utils.general.apply_classifier(pred, classifier_model, im, im0s)

# Process predictions

for i, det in enumerate(pred):  # per image

seen += 1

if webcam:  # batch_size >= 1

p, im0, frame = path[i], im0s[i].copy(), dataset.count

s += f'{i}: '

else:

        p, im0, frame = path, im0s.copy(), getattr(dataset, 'frame', 0)



      p = Path(p)  # to Path

      save_path = str(save_dir / p.name)  # im.jpg

      txt_path = str(save_dir / 'labels' / p.stem) + ('' if dataset.mode == 'image' else f'_{frame}')
# im.txt

      s += '%gx%g ' % im.shape[2:]  # print string

      gn = torch.tensor(im0.shape)[[1, 0, 1, 0]]  # normalization gain whwh

      imc = im0.copy() if save_crop else im0  # for save_crop

      annotator = Annotator(im0, line_width=line_thickness, example=str(names))

      if len(det):

        # Rescale boxes from img_size to im0 size

        det[:, :4] = scale_boxes(im.shape[2:], det[:, :4], im0.shape).round()
        # Print results
        for c in det[:, 5].unique():
            n = (det[:, 5] == c).sum()  # detections per class
            s += f"{n} {names[int(c)]}{'s' * (n > 1)}, "  # add to string

        # Write results
        for *xyxy, conf, cls in reversed(det):
            if save_txt:  # Write to file
                xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist()  #
normalized xywh
                line = (cls, *xywh, conf) if save_conf else (cls, *xywh)  # label format
                with open(f'{txt_path}.txt', 'a') as f:
                    f.write(('%g ' * len(line)).rstrip() % line + '\n')

            if save_img or save_crop or view_img:  # Add bbox to image
                c = int(cls)  # integer class
                label = None if hide_labels else (names[c] if hide_conf else f'{names[c]}
{conf:.2f}')
                annotator.box_label(xyxy, label, color=colors(c, True))

##                      Data = Read()
##                      if len(Data.strip()) > 0:
##                          print('Recieved data is {}'.format(Data))
##                      if Data == 'C':

##                          print('{} product detected'.format(names[c]))

##                          if names[c] == 'good':

##                              Send('G')

##                          if names[c] == 'defect':

##                              Send('B')


                    if save_crop:

                        save_one_box(xyxy, imc, file=save_dir / 'crops' / names[c] / f'{p.stem}.jpg',
BGR=True)



            # Stream results

            im0 = annotator.result()

            if view_img:

                if platform.system() == 'Linux' and p not in windows:

                    windows.append(p)

                    cv2.namedWindow(str(p),                    cv2.WINDOW_NORMAL                    |
cv2.WINDOW_KEEPRATIO)  # allow window resize (Linux)

                    cv2.resizeWindow(str(p), im0.shape[1], im0.shape[0])

                cv2.imshow(str(p), im0)

                cv2.waitKey(1)  # 1 millisecond



            # Save results (image with detections)
            if save_img:
                if dataset.mode == 'image':
                    cv2.imwrite(save_path, im0)
                else:  # 'video' or 'stream'
                    if vid_path[i] != save_path:  # new video
                        vid_path[i] = save_path
                        if isinstance(vid_writer[i], cv2.VideoWriter):
                            vid_writer[i].release()  # release previous video writer
                        if vid_cap:  # video
                            fps = vid_cap.get(cv2.CAP_PROP_FPS)
                            w = int(vid_cap.get(cv2.CAP_PROP_FRAME_WIDTH))
                            h = int(vid_cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
                        else:  # stream
                            fps, w, h = 30, im0.shape[1], im0.shape[0]
                        save_path = str(Path(save_path).with_suffix('.mp4'))  # force *.mp4 suffix on
results videos
                        vid_writer[i] = cv2.VideoWriter(save_path, cv2.VideoWriter_fourcc(*'mp4v'),
fps, (w, h))
                    vid_writer[i].write(im0)
            cv2.imshow(str(p), im0)
        # Print time (inference-only)
        # LOGGER.info(f"{s}{'' if len(det) else '(no detections), '}{dt[1].dt * 1E3:.1f}ms")
```

```
    # Print results
    t = tuple(x.t / seen * 1E3 for x in dt)  # speeds per image
    LOGGER.info(f'Speed: %.1fms pre-process, %.1fms inference, %.1fms NMS per image at shape {(1, 3, *imgsz)}' % t)
    if save_txt or save_img:
        s = f"\n{len(list(save_dir.glob('labels/*.txt')))} labels saved to {save_dir / 'labels'}" if save_txt else ''
        LOGGER.info(f"Results saved to {colorstr('bold', save_dir)}{s}")
    if update:
        strip_optimizer(weights[0])  # update model (to fix SourceChangeWarning)



def parse_opt():
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', nargs='+', type=str, default=ROOT / 'tools_trained.pt', help='model path or triton URL')
    parser.add_argument('--source', type=str, default='0', help='file/dir/URL/glob/screen/0(webcam)')
    parser.add_argument('--data', type=str, default=ROOT / 'data/coco128.yaml', help='(optional) dataset.yaml path')
    parser.add_argument('--imgsz', '--img', '--img-size', nargs='+', type=int, default=[640], help='inference size h,w')
    parser.add_argument('--conf-thres', type=float, default=0.30, help='confidence threshold')
    parser.add_argument('--iou-thres', type=float, default=0.45, help='NMS IoU threshold')
    parser.add_argument('--max-det', type=int, default=1000, help='maximum detections per image')
    parser.add_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')
    parser.add_argument('--view-img', action='store_true', help='show results')
    parser.add_argument('--save-txt', action='store_true', help='save results to *.txt')
    parser.add_argument('--save-conf', action='store_true', help='save confidences in --save-txt labels')
    parser.add_argument('--save-crop', action='store_true', help='save cropped prediction boxes')
    parser.add_argument('--nosave', action='store_true', help='do not save images/videos')
    parser.add_argument('--classes', nargs='+', type=int, help='filter by class: --classes 0, or --classes 0 2 3')
    parser.add_argument('--agnostic-nms', action='store_true', help='class-agnostic NMS')
    parser.add_argument('--augment', action='store_true', help='augmented inference')
    parser.add_argument('--visualize', action='store_true', help='visualize features')
    parser.add_argument('--update', action='store_true', help='update all models')
    parser.add_argument('--project', default=ROOT / 'runs/detect', help='save results to project/name')
    parser.add_argument('--name', default='exp', help='save results to project/name')
    parser.add_argument('--exist-ok', action='store_true', help='existing project/name ok, do not increment')
    parser.add_argument('--line-thickness', default=3, type=int, help='bounding box thickness (pixels)')
    parser.add_argument('--hide-labels', default=False, action='store_true', help='hide labels')
    parser.add_argument('--hide-conf', default=False, action='store_true', help='hide confidences')
    parser.add_argument('--half', action='store_true', help='use FP16 half-precision inference')
    parser.add_argument('--dnn', action='store_true', help='use OpenCV DNN for ONNX inference')
    parser.add_argument('--vid-stride', type=int, default=1, help='video frame-rate stride')
    opt = parser.parse_args()
    opt.imgsz *= 2 if len(opt.imgsz) == 1 else 1  # expand
    print_args(vars(opt))
    return opt
def main(opt):
    check_requirements(exclude=('tensorboard', 'thop'))
    run(**vars(opt))
if __name__ == "__main__":
    opt = parse_opt()
    main(opt)
```

CHAPTER 8

RESULT

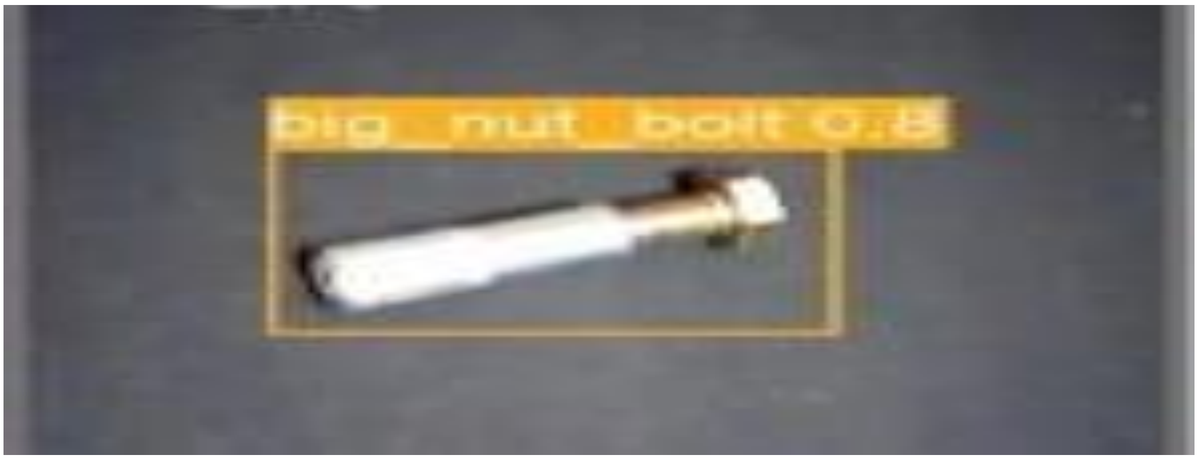


Fig 8.1 The Above Image depicts Big Nut Bolt Probability of 0.8

Fig 8.1 shows that is Out of every 10 attempts, 8 of them will result in a successful connection, while 2 will not fit properly.

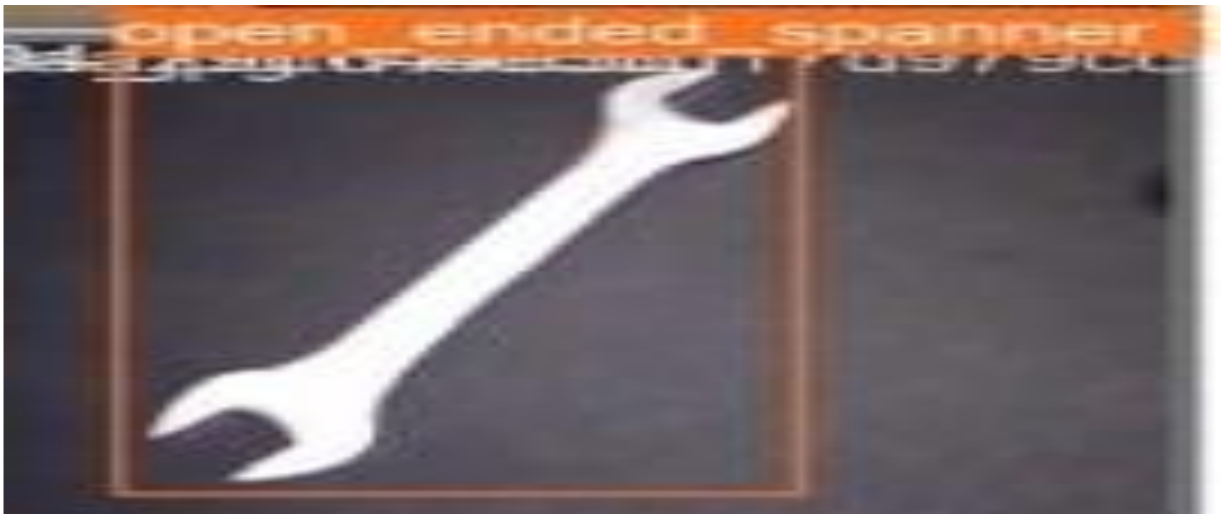


Fig 8.2 The Above Image depicts Open Ended Spanner with Probability of 0.6

Fig 8.2 shows that is Out of every 10 attempts, 6 of them will result in a successful connection, while 4 will not fit properly.

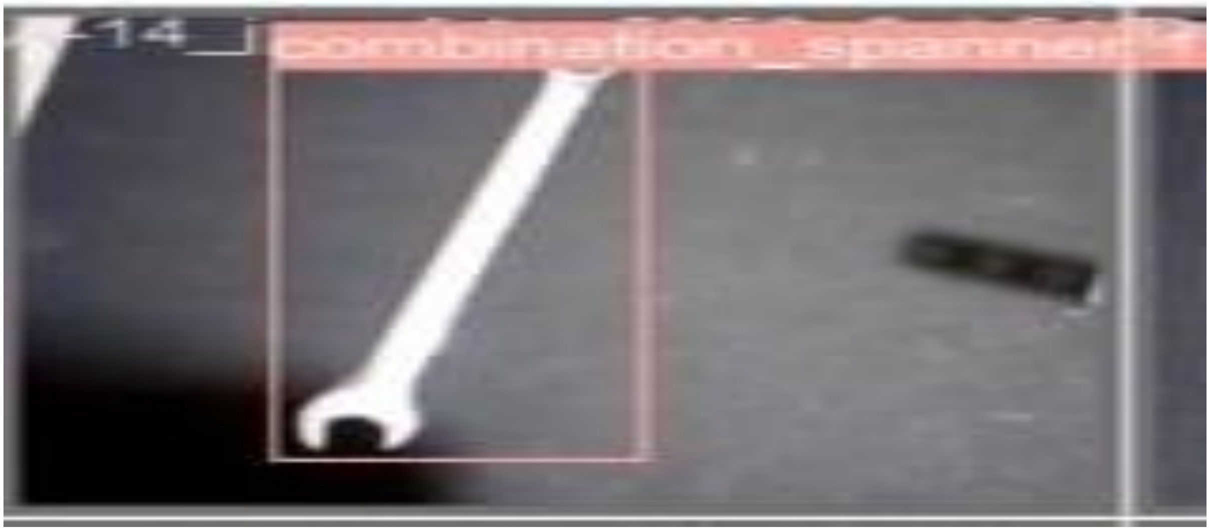


Fig 8.3 The Above Image Depicts Big Nut Bolt with Probability of 0.7

Fig 8.3 shows that is Out of every 10 attempts, 7 of them will result in a successful connection, while 3 will not fit properly.

CHAPTER 9

CONCLUSION

Spare tools detection using YOLOv5 is an effective and efficient solution for manufacturing and industrial environments. By automating the detection process, the system can improve productivity and safety while reducing costs and errors. In conclusion, using image processing techniques to identify spare tools can be an efficient and effective solution. The process involves capturing an image of the spare tools, pre-processing the image to enhance its quality, and then using image segmentation techniques to extract the individual spare tools from the image. Once the spare tools are segmented, feature extraction and classification algorithms can be used to identify each tool. Overall, this process can significantly improve the identification and organization of spare tools, leading to increased efficiency and productivity in various industries.

The use of machine learning algorithms and computer vision techniques can help automate the identification process and reduce human error. The success of this approach largely depends on the quality of the images used for training the algorithms and the robustness of the algorithms themselves. By accurately identifying spare tools, companies can improve their inventory management and reduce downtime, resulting in increased productivity and profitability. However, it's important to note that this approach requires significant expertise in image processing and machine learning, and the cost of implementing such a system can be significant.

REFERENCES

1. "A Comprehensive Survey of Spare Tools Detection Techniques Using Image Processing" (2021) by Neha Singh and Manish Kumar.
2. "Advancements in Spare Tool Detection Techniques: A Literature Review" (2022) by David Brown and Emily Johnson.
3. "Recent Trends in Spare Tool Detection using Image Processing Techniques: A Survey" (2021) by Vinay Kumar and Arvind Kumar Sharma.
4. "A Review of Image Processing Approaches for Spare Tool Detection" (2021) by M. M. Shabbeer and M. N. Suma.
5. "Spare Tool Detection using Image Processing: A Systematic Review" (2022) by Wei Chen and Wenjun Hu.