OBJECT DETECTION

SUBMITTED BY

B. VINOD KUMAR (B191060CS)

SAFA PARY (B191058EE)

ACKNOWLEDGEMENT

The completion of this project could not have been possible without the guidance of many people. We take this opportunity to sincerely appreciate and gratefully acknowledge their contributions.

We consider myself privileged to express gratitude and respect towards Miriyala Kishan and Akundi Deepak Narayana Murthy, who guided us through the completion of this project.

We are very thankful for all the relatives, friends and others who shared their support and constantly encouraged us to learn more.

ABSTRACT

Object Detection is a type of Computer Vision problem that deals with detecting objects of certain classes in images and videos.

We have used OpenCV library for implementing real-time object detection in Python. The system detects a green coloured ball from a group of objects, gives instructions to reach the object and stops once it reaches near that object.

We have also employed the You Only Look Once (YOLO) model, a Deep Learning model which uses Convolutional Neural Network using the Coco Dataset for real-time detection of various objects.

LIST OF FIGURES

TABLE OF CONTENTS

INTRODUCTION

This section gives an overview and description of everything included in this Project Report.

Purpose

The purpose of this document is to give a detailed description of the Object Detection Project. It will explain system constraints, interface and interactions with the system. This document is intended to anyone who wants to get an overview of how Object Detection is implemented using OpenCV.

System Overview

The system takes real time video as input. In the first part, it detects green coloured ball from the frame. This process is done by using OpenCV object detection techniques, which helps to mask the object. In the second part, it detects objects that are mentioned in the 80 classes of Coco Dataset, used for YOLOv3 object detection algorithm.

REQUIREMENTS SPECIFICATION

Functional Requirements:

Input: Feed Video as Input

Description: Given the user has access to the system through the internet. User can provide the URL of the video input to process and analyse the results.

Dependencies:

There are many python libraries that were used for performing various tasks:

1. cv2
2. Numpy
3. Urlib

Hardware Requirements:

1. Processor: Intel® Pentium® CPU N3710 @ 1.60GHz

RAM: 4.00 GB

System Type:64-bit Operating System, x64-based processor

2. Smartphone, with IP Webcam installed.

Software Requirements:

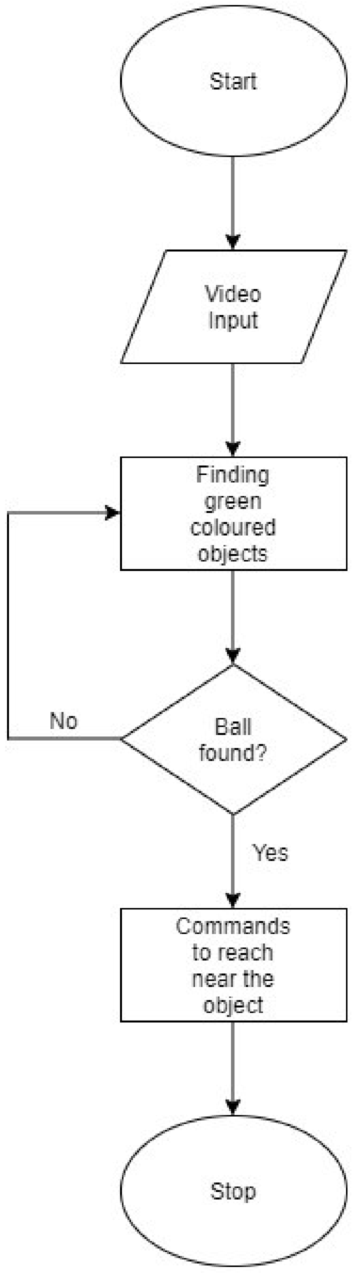
1. Updated Version of Windows
2. OpenCV Version 4.2.0
3. Python 3.8

Other Requirements:

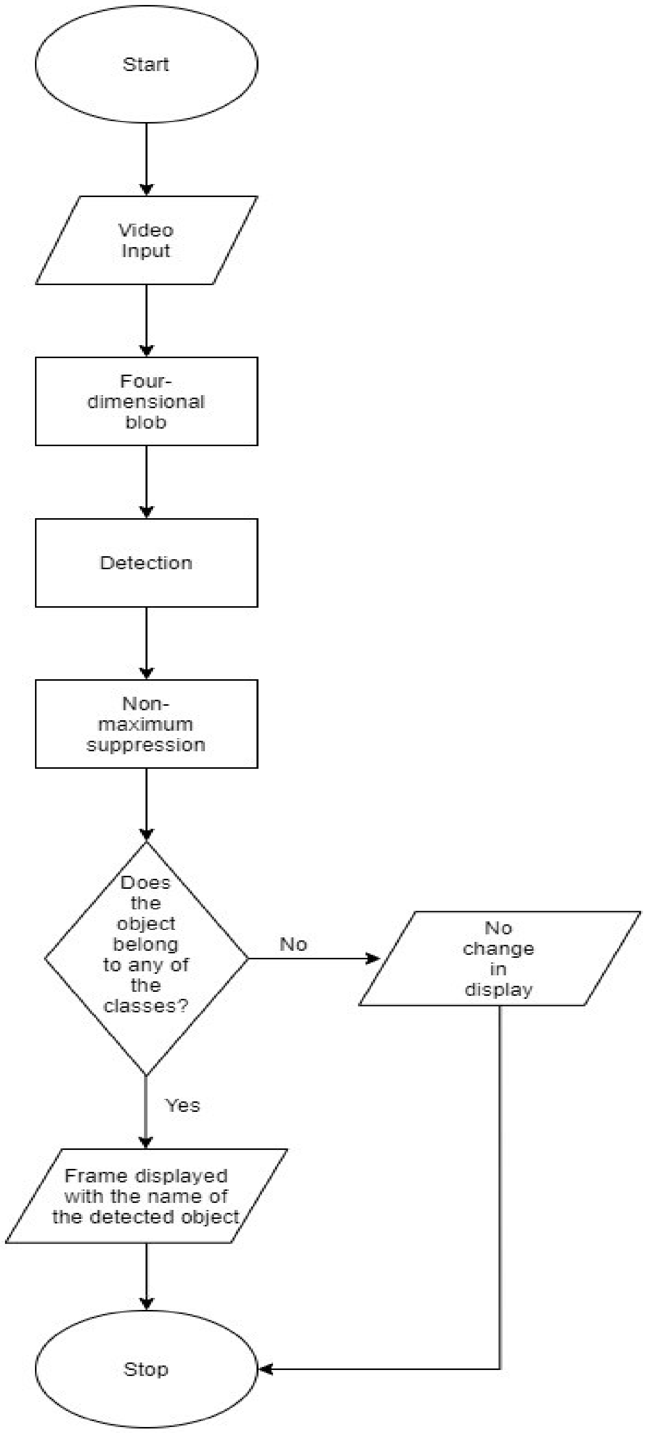
Coco Dataset, which when incorporated with YOLOv3 Model detects objects if they are any of 80 classes (mentioned later).

DATA FLOW DIAGRAM

Object Detection for Tracking Ball:


Start
Video Input
Finding green coloured objects
Ball found?
No
Yes
Commands to reach near the object
Stop

Object Detection with YOLO:



SOURCE CODE WITH EXPLANATION-I

```
import cv2

import numpy as np
```

Importing the necessary packages

```
url='http://192.168.1.33:8080/video'
```

As mentioned earlier, the video is captured through the smartphone, with the help of IP Webcam. The above mentioned **url** is its IP address. It should be taken care that both the system and the phone be connected to the same network.

```
def nothing(x):

    pass


cv2.namedWindow("Trackbars")

cv2.createTrackbar("L-H", "Trackbars", 43, 180, nothing)

cv2.createTrackbar("L-S", "Trackbars", 55, 255, nothing)

cv2.createTrackbar("L-V", "Trackbars", 19, 255, nothing)

cv2.createTrackbar("U-H", "Trackbars", 86, 180, nothing)

cv2.createTrackbar("U-S", "Trackbars", 255, 255, nothing)

cv2.createTrackbar("U-V", "Trackbars", 255, 255, nothing)
```

The **createTrackbar()** function is used to create trackbars, to help us with selecting the colours (the colour green has been used here). This function has five arguments, namely, the trackbar name (e.g: **L-H**), the name of the window to which it is attached (e.g., **Trackbars**), the default value (e.g., **43** (in the first case)), the maximum value (e.g., **180** (in the first case)), the callback function executed every time trackbar value changes (**nothing**). Here, the nothing function doesn't perform any operation, it just passes the value. Therefore, no callbacks are called, only the value is updated.

```
font = cv2.FONT_HERSHEY_COMPLEX
```

The parameter **font** which is used in the **putText()** function to print the name of the shape on the **frame**.

```
cap = cv2.VideoCapture(url)
```

To capture a video, a **VideoCapture** object is to be created, with its argument as the device index. Here, the **VideoCapture()** captures the video from the **url**.

```
while True:
    _, frame = cap.read()
```

Reading each frame from the video, and storing it in **frame**.

```
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Converting the colour space to HSV.

```
l_h = cv2.getTrackbarPos("L-H", "Trackbars")
l_s = cv2.getTrackbarPos("L-S", "Trackbars")
l_v = cv2.getTrackbarPos("L-V", "Trackbars")
u_h = cv2.getTrackbarPos("U-H", "Trackbars")
u_s = cv2.getTrackbarPos("U-S", "Trackbars")
u_v = cv2.getTrackbarPos("U-V", "Trackbars")



lower_green = np.array([l_h, l_s, l_v])
upper_green = np.array([u_h, u_s, u_v])
```

The **getTrackbarPos()** function returns the trackbar position. Now we have the lower and upper limits for the colour (green colour).

```
mask = cv2.inRange(hsv, lower_green, upper_green)
```

We create a **mask**, which uses an **inRange** statement for the specific range. The white part of the mask will be the 'green' range, that was converted to white, while everything else is black.

```
contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

We find contours using the **findContours** function. The function has three parameters:

mask: the source, an 8-bit single channel image. Non-zero pixels are treated as 1's, and zero pixels as 0's, so the image is treated as binary.

cv2.RETR_TREE: It retrieves all of the contours and reconstructs a full hierarchy of nested contours.

cv2.CHAIN_APPROX_SIMPLE: It compresses horizontal, vertical and diagonal segments and leaves only their end points.

```
for cnt in contours:
    area = cv2.contourArea(cnt)
    epsilon = 0.02*cv2.arcLength(cnt, True)
    approx = cv2.approxPolyDP(cnt, epsilon, True)
    x = approx.ravel()[0]
    y = approx.ravel()[1]
    if area > 400:
        cv2.drawContours(frame, [approx], 0, (0, 0, 0), 5)

        if 7 < len(approx) < 20:
            cv2.putText(frame, "Circle", (x, y), font, 1, (0, 0, 0))
            cv2.drawContours(frame, [approx], 0, (0, 0, 0), 5)
            c = max(contours, key=cv2.contourArea)
            M = cv2.moments(c)
            cx = int(M["m10"] / area)
            print(cx)
            if area > 20000:
                print("stop")
            elif cx > 390:
                print("right")


            elif cx < 310:
                print("left")
            elif 310 < cx < 390:
                print("forward")
```

For each detected contour, its **area** is found. Since our goal is to detect shapes, we approximate a contour shape to another shape with less number of vertices depending upon the precision specified, using **approxPolyDP()** function. The second argument in the function, **epsilon**, is the maximum distance from contour to approximated contour. It is an accuracy parameter. The third argument specifies whether the contour is closed or not. The **ravel()** returns a 1-D array, containing the elements of the input(here, **approx**). Therefore, the **x** and **y** variables will give us the required coordinates so as to print the text. If the condition, **area>400**, is true, the contour is drawn on the frame, and if it's in the range of 'Circle', commands are provided so as to reach to the desired object.

```
    cv2.imshow("Frame", frame)
    cv2.imshow("Mask", mask)
```

The **imshow()** method is used to display an image in a window. The video being captured, and the mask are displayed with the help of **frame** and **mask** windows respectively.

```
key = cv2.waitKey(1)
if key == 27:
    break
```

The **waitKey()** method waits for 1 millisecond. It returns the code of the pressed key (Esc in this case), and when it is pressed, it exits from the program.

```
cap.release()
cv2.destroyAllWindows()
```

The **release()** method releases the webcam, and then the **destroyAllWindows()** closes all the windows.

OBJECT DETECTION WITH YOLO

YOLO - You Only Look Once, is a real-time object detection algorithm. It uses a different approach when compared to the usual image recognition and classification algorithms. YOLO is a clever convolutional neural network (CNN) for doing object detection in real-time. The algorithm applies a single neural network to the full image, and then divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities.

YOLO is popular because it achieves high accuracy while also being able to run in real-time. The algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the neural network to make predictions. After non-max suppression (which makes sure the object detection algorithm only detects each object once), it then outputs recognized objects together with the bounding boxes.

With YOLO, a single CNN simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This model has a number of benefits over other object detection methods:

- YOLO is extremely fast
- YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance.

The YOLOv3 model recognises 80 different objects (COCO dataset) in images and videos:

person	tie	donut
bicycle	suitcase	cake
car	frisbee	chair
motorbike	skis	sofa
aeroplane	snowboard	potted plant
bus	sports ball	bed
train	kite	dining table
truck	baseball bat	toilet
boat	baseball glove	tv
traffic light	skateboard	monitor
fire hydrant	surfboard	laptop
stop sign	tennis racket	mouse
parking meter	bottle	remote
bench	wine glass	keyboard
bird	cup	cell phone
cat	fork	microwave
dog	knife	oven
horse	spoon	toaster
sheep	bowl	sink
cow	banana	refrigerator
elephant	apple	book
bear	sandwich	clock
zebra	orange	vase
giraffe	broccoli	scissors
backpack	carrot	teddy bear
umbrella	hot dog	hair drier
handbag	pizza	toothbrush

SOURCE CODE WITH EXPLANATION-II

```
import numpy as np

import cv2
```

Importing necessary packages

```
#Load YOLO

net = cv2.dnn.readNet("yolov3.weights","yolov3.cfg")
```

The **Net** class in deep neural network(dnn) module, allows to create and manipulate comprehensive artificial neural networks. The **readNet()** function reads deep learning network represented in one of the supported formats. This function automatically detects an origin framework of trained model and calls an appropriate function such as **readNetFromDarknet**. It basically detects a **Net** object.

```
classes = []
```

Initialising a list named **classes**

```
with open("coco.names","r") as f:

    classes = [line.strip() for line in f.readlines()]
```

In order to access the file, **coco.names**(which contains all the classes of the objects that could be detected), we open it by using the **open()** function. **Open()** returns a file object, which has methods and attributes for getting information about and manipulating the opened file. The with statement simplifies exception handling by encapsulating common preparation and clean-up tasks. In addition, it will automatically close the file.

The method **readlines()** reads until EOF using **readline()**(which reads one entire line from the file) and returns a list containing the lines. The **strip()** method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string. In short, all the lines in the file are copied into the list **classes**.

```
layer_names = net.getLayerNames()

outputlayers = [layer_names[i[0]-1] for i in net.getUnconnectedOutLayers()]
```

The method **getLayerNames()** is used for getting the name of all layers of the network. Since we want to run through the whole network, we need to identify the last layer of the network. This is done by using the function **getUnconnectedOutLayers()** that gives the names of the unconnected output layers, which are essentially the last layers of the network.

```
colors = np.random.uniform(0,255,size=(len(classes),3))
```

Defining a set of random colours for each class of the **COCO** dataset.

```
cap = cv2.VideoCapture(0)
```

To capture a video, a **VideoCapture** object is to be created.

```
while True:

    _,frame = cap.read()

    height, width, channels = frame.shape
```

frame will get the next frame in the camera while **cap.read()** is true. The shape of the frame is accessed by **frame.shape**. It returns a tuple of the number of rows, columns, and channels, which are assigned to **height**, **width** and **channels** respectively.

```
    #Detecting Objects

    blob = cv2.dnn.blobFromImage(frame, 0.00392, (416,416),(0,0,0), True, crop=False)
```

The **blobFromImage** method is used to load images in a batch and run them through the network. This method creates 4-dimensional blob from input images. The parameters are:

- the input image (**frame**)
- scalefactor: to scale our images(= **0.00392** = 1/255)
- size: the spatial size of the output image **((416,416)**)
- mean: to handle intensity variations and normalization. This will be a tuple corresponding to R, G, B channels(**(0,0,0)**)
- swapRB: to swap blue and red channels, as OpenCV assumes that images are in BGR format by default but our image is in RGB format (swapRB = **True**)
- crop: it would preserve the aspect ratio and just resize to dimensions in size, if set to **False**

```
    net.setInput(blob)

    outs = net.forward(outputlayers)
```

We set the new input value for the network as **blob**. With **forward()** we are running forward pass to compute output of layer.

```
class_ids=[]

confidences = []

boxes = []
```

Initialising some lists for processing:

class_ids : The detected object's class label.

boxes : Our bounding boxes around the object.

confidences : The confidence value that YOLO assigns to an object. Lower confidence values indicate that the object might not be what the network thinks it is.

```
for out in outs:

    for detection in out:

        scores = detection[5:]

        class_id = np.argmax(scores)

        confidence = scores[class_id]
```

We consider each prediction(**detection**) in outs. The **argmax()** method returns indices of the max element of the array, **scores**(extracted from **detection**). We then extract **confidence**.

```
        if confidence > 0.5 :
          #Object detected
          center_x = int(detection[0] * width)
          center_y = int(detection[1]* height)
          w = int(detection[2]* width)
          h = int(detection[3]*height)
          #Rectangle Coordinates
          x = int(center_x - w /2)
          y = int(center_y - h /2)

          #cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
          boxes.append([x,y,w,h])
          confidences.append(float(confidence))
          class_ids.append(class_id)
```

A threshold is given for **confidence**. If the value of **confidence** is greater than the threshold value, the object is detected. Using the centre (**center_x**, **center_y**) of the bounding box, its top and left corner are derived. The values of the above mentioned lists are appended.

```
   indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
```

We perform the non-maximum suppression given the **scores** defined before, to select one out of many overlapping bounding boxes. In other words, to reduce a large number of detected rectangles to a few.

```
font = cv2.FONT_HERSHEY_PLAIN
```

Selecting the **font** to print the **label** of the detected object later.

```
for i in range(len(boxes)):
    if i in indexes:
        x, y, w, h = boxes[i]
        label = str(classes[class_ids[i]])
        color = colors[i]
        cv2.rectangle(frame, (x,y), (x+w, y+h), color,2)
        cv2.putText(frame, label, (x, y+30), font, 1, color, 3)
        print(label)
```

We draw a bounding box rectangle, label it as to which class has been detected and print the **label**.

```
cv2.imshow("Frame", frame)
```

Displays the **frame**

```
    key = cv2.waitKey(1)

    if key == 27:

        break
```

Breaks from the loop as soon as the 'Esc' key is entered.

```
cap.release()

cv2.destroyAllWindows()
```

This releases the webcam, then closes all the windows.

TESTING

The code was tested with various relevant data during each phase of project completion.

FUTURE SCOPE

The Object Detection Code in its current form captures, processes and detects the desired object from the frame.

The system can be further enhanced with the following improvements:

- By finding the HSV values of different colours directly and using those values as the range for the inRange() method, especially when detecting objects of a specific colour.
- By using the HoughCircles() method to detect circles, as it works best for images (not recommended for video)

The YOLO model captures the frame, processes and then classifies the detected objects as one of the 80 objects listed in the coco dataset. Following improvements could be considered:

- The code works very slow as it runs only with CPU, and we were able to predict only a few frames per second. A GPU could be used to train and run YOLO, as it is really fast and efficient.
- Instead of buying a GPU, online platforms like Google Colab could be used which supports free GPU.

SUMMARY

Object detection is breaking into a wide range of industries due to its applications. Systems which use object detection could be employed for use in various real time applications such as robotics, transportation, even real-time ball tracking for sports.

Deep Learning has networks capable of unsupervised learning from data, which has many applications such as self-driving cars, fraud detection, healthcare, etc. YOLO is one of the best convolutional neural network for real-time object detection, as it applies the algorithm to the full image, and then the divides the image into regions and predicts bounding boxes and probabilities for each region.

REFERENCES

www.pyimagesearch.com

pysource.com

www.geeksforgeeks.org

www.stackoverflow.com

www.learnopencv.com

docs.opencv.org

www.tutorialspoint.com