Class ID:ML06B1-1

Problem statement:
Object Detection: Take an image and get the number of the objects inside the image i.e, each object name.

Introduction:
Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class in digital images and videos. Well-researched domains of object detection include face detection and pedestrian detection.

Libraries used:
- OpenCV (https://opencv.org/)
  OpenCV is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then Itseez. The library is cross-platform and free for use under the open-source BSD license.
- Numpy (https://numpy.org/)
  NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- Matplotlib (https://matplotlib.org/)
  Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+.

Modules used:
- OS (https://docs.python.org/3.0/library/os.html)
  This module provides a portable way of using operating system dependent functionality.

Neural Network used:
- Mobilenet SSD (https://github.com/chuanqi305/MobileNet-SSD)
  A Caffe implementation of MobileNet-SSD detection network, with pre-trained weights on VOC0712 and mAP=0.727.

Algorithm used:
- YOLO (https://pjreddie.com/darknet/yolo/)
  YOLO is a state of the art real-time object detection system.

Virtual notebook handler used:
- Jupyter Notebook (https://jupyter.org/)
  Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

Working:

Overview of the system:
We plan to use MobileNet SSD and YOLO in order to solve the object detection problem. SSD is more accurate compared to YOLO but the problem with them is that the number of objects detectable by the pre-trained model of the MobileNet SSD we use is low compared to the objects that are detectable by using the YOLO algorithm. YOLO algorithm is fast compared to SSD but it lacks the accuracy that we get with using the SSDs. So we plan to use the SSD for the initial round of object detection and if it fails to detect any object or if the user is not satisfied with the output then the user can opt to do a deeper search using YOLO algorithm.

Importing the prerequisites that are required:
First, we import the most necessary packages we need to read, resize and process the image. Rest of the required libraries can be imported as required later in the code.

```
#Importing the necessary packages
import numpy as np
import cv2
import os
```

The information about the libraries that are imported is given above under the sub-heading Libraries used

Taking the input from the user and setting the proper resource paths:
There are many ways available through which we can take the input which include local image directory, inputting image URL and so on. But this might cause inconvenience for testing and the target user of this code will be fairly knowledgeable with handling local image directories. Thus we have already stored some example images under the directory images. The user can add the images to this folder to test the system.
We ask the user to input the image path after the user has inserted the image into the folder images. Then we read the image using the methods available through OpenCV.

```
#Setting up proper resource paths
image_path=input('Path of valid input image=')
proto_path= "MobileNetSSD_deploy.prototxt.txt"
model_path= "MobileNetSSD_deploy.caffemodel"
confidence_m= 0
```

Creating variable and list to store the number of objects identified and their names:

```
#creating a variable to store the number of objects detected
num_of_objects = 0

#creating a list to store the detected objects
objects_list = []
```

Then we build a list called CLASSES which contain labels. This is followed by list colours containing corresponding random colours for bounding boxes. After all this, we load our model by printing a message and loading the model.

```
# initialize the list of class labels MobileNet SSD was trained to
# detect, then generate a set of bounding box colors for each class
CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
 "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
 "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
 "sofa", "train", "tvmonitor"]
COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))
```

```
# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
```

Next, we will load our query image and prepare our blob which we will feed-forward through the network.Taking note of the comment in this block, we load our image, extract the height and width , and calculate a 300 by 300-pixel blob from our image.

```
# load the input image and construct an input blob for the image
# by resizing to a fixed 300x300 pixels and then normalizing it
# (note: normalization is done via the authors of the MobileNet SSD
# implementation)
image = cv2.imread(args["image"])
(h, w) = image.shape[:2]
blob = cv2.dnn.blobFromImage(cv2.resize(image, (300, 300)), 0.007843,
	(300, 300), 127.5)
```

Now we’re ready to do the heavy lifting we’ll pass this blob through the neural network.

```
# pass the blob through the network and obtain the detections and
# predictions
print("[INFO] computing object detections…”)
net.setInput(blob)
detections = net.forward()
```

We set the input to the network and compute the forward pass for the input, storing the result as detections. Computing the forward pass and associated detections could take a while depending on your model and input size.

Let’s loop through our detections and determine what and where the objects are in the image. We start by looping over our detections, keeping in mind that multiple objects can be detected in a single image.

```
# loop over the detections
for i in np.arange(0, detections.shape[2]):
        # extract the confidence (i.e., probability) associated with the
        # prediction
        confidence = detections[0, 0, i, 2]
        # filter out weak detections by ensuring the `confidence` is
        # greater than the minimum confidence
        if confidence > confidence_m:
        # extract the index of the class label from the `detections`,
        # then compute the (x, y)-coordinates of the bounding box for
        # the object
        idx = int(detections[0, 0, i, 1])
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")
        #Counting the number of objects detected
        num_of_objects = num_of_objects + 1

        #Appending the objects discovered into the list containing the objects
        objects_list.append(CLASSES[idx])
        # display the prediction
        label = "{}: {:.2f}%".format(CLASSES[idx], confidence * 100)
        print("[INFO] {}".format(label))
        cv2.rectangle(image, (startX, startY), (endX, endY),
        COLORS[idx], 2)
        y = startY - 15 if startY - 15 > 15 else startY + 15
        cv2.putText(image, label, (startX, y),
        cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)
```

We also apply a check to the confidence (i.e., probability) associated with each detection. If the confidence is high enough (i.e. above the threshold), then we’ll display the prediction in the terminal as well as draw the prediction on the image with text and a coloured bounding box. Looping through our detections first we extract the confidence value If the confidence is above our minimum threshold we extract the class label index and compute the bounding box around the detected object.Then, we extract the *(x, y)*-coordinates of the box which we will use shortly for drawing a rectangle and displaying text. Next, we build a text label containing the class name and confidence. Using the label, we print it to the terminal, followed by drawing a coloured rectangle around the object using our previously extracted *(x, y)*-coordinates.In general, we want the label to be displayed above the rectangle, but if there isn’t room, we’ll display it just below the top of the rectangle. Finally, we overlay the coloured text onto the image using the *y-values* that we just calculated.

Now we print the number of objects detected and names of the objects:

```
print("Number of objects detected by the MobileNet SSD are:",num_of_objects)
```

Now as Opencv shows the image in BGR we convert the image to RGB format. And then, at last, we show the image. The image will be like the object bounded in a coloured box with its name written.

```
# show the output image
import matplotlib.pyplot as plt
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
# as opencv loads in BGR format by default, we want to show it in RGB.
plt.show()
```

This was using Mobilenet SSD.

Now we display a message whether the user wants to do a deep search using YOLO. If yes then it will work as follows:-

As we have imported all the libraries in the beginning and also we have created a list of labels and colours we don't need to do that here. We just need to load the YOLO algorithm.

```
response = input("Do you want to do a deeper search using YOLO algorithm? Type Yes or No")
```

```
if response.lower() == "yes":
        # Load Yolo
        net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")
        classes = []
        with open("coco.names", "r") as f:
        classes = [line.strip() for line in f.readlines()]
        layer_names = net.getLayerNames()
        output_layers = [layer_names[i[0] - 1] for i in net.getUnconnectedOutLayers()]
        colors = np.random.uniform(0, 255, size=(len(classes), 3))
```

After the loading of the algorithm, we will load the image to be used to detect the objects.

```
# Loading image
        img = cv2.imread(image_path)
        img = cv2.resize(img, None, fx=0.4, fy=0.4)
        height, width, channels = img.shape
```

Then we create a variable to store the number of objects detected and also we create a list to store the objects detected.

```
#creating a variable to store the number of objects detected using YOLO
        num_of_objects_yolo = 0

#creating a list to store the names of the objects detected using YOLO algorithm
        yolo_object_list = []
```

Then we load our image, extract the height and width, and calculate a 416 by 416-pixel blob from our image.

```
 # Detecting objects
        blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0), True, crop=False)
net.setInput(blob)
        outs = net.forward(output_layers)
```

We set the input to the network and compute the forward pass for the input, storing the result as outs. Computing the forward pass and associated detections could take a while depending on your model and input size. At this point, the detection is done, and we only need to show the result on the screen. We then loop through the outs array, we calculate the confidence and we choose a confidence threshold.

We set threshold confidence of 0.5, if it's greater we consider the object correctly detected, otherwise we skip it. The threshold goes from 0 to 1. The closer to 1 the greater is the accuracy of the detection, while the closer to 0 the less is the accuracy but also it's greater the number of objects detected.

```
# Showing information on the screen
        class_ids = []
        confidences = []
        boxes = []
        for out in outs:
        for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        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)
                boxes.append([x, y, w, h])
                confidences.append(float(confidence))
                class_ids.append(class_id)
```

When we perform the detection, it happens that we have more boxes for the same object, so we should use another function to remove this “noise”. Its called Non-Maximum Suspension.

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

We finally extract all the information and show them on the screen.

- **Box**: contain the coordinates of the rectangle surrounding the object detected.
- **Label**: it’s the name of the object detected
- **Confidence**: the confidence about the detection from 0 to 1.

```
font = cv2.FONT_HERSHEY_SIMPLEX
        for i in range(len(boxes)):
        if i in indexes:
        x, y, w, h = boxes[i]
        label = str(classes[class_ids[i]])
        num_of_objects_yolo = num_of_objects_yolo + 1
        yolo_object_list.append(label)
        color = colors[i]
        cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
```

Now we print the number of objects detected and the names of the objects that are detected:

```
print("Number of objects detected using YOLO algorithm are:",num_of_objects_yolo)
        print("\nThe names of the objects detected are:\n",yolo_object_list)
```

As by default image will be in BGR format so we convert it to RGB format for displaying the image and the objects will be displayed in a bounded coloured box.

```
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        # as opencv loads in BGR format by default, we want to show it in RGB.
        plt.show()
```