Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

MACHINE PROBLEM 5			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	Week No.:	**8-9**
Course Code:	**CSST106**	Term:	**1st Sem.**
Course Title:	**Perception and Computer Vision**	Academic Year:	**2024-2025**
Student Name:	**Lesly-Ann B. Victoria**	Section:	**BSCS-4B**
Due Date:		Points:	

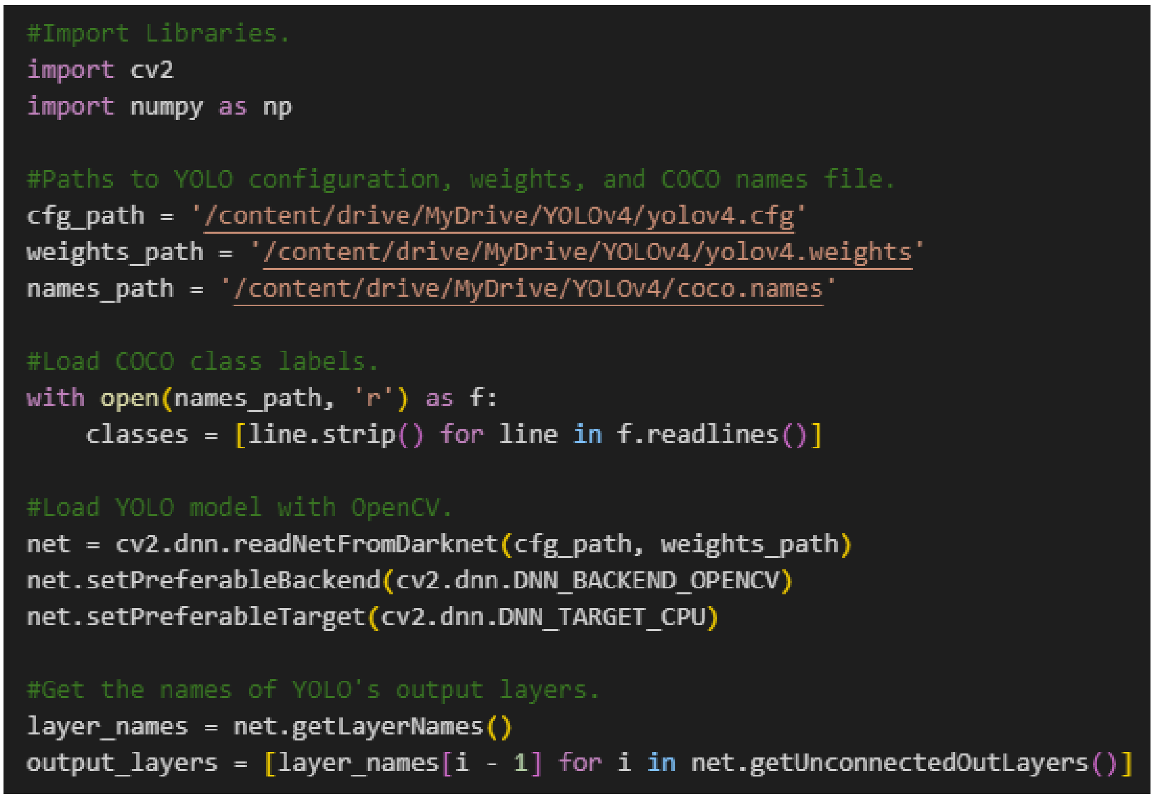
Machine Problem: Object Detection and Recognition using YOLO.

Objective:
To implement real-time object detection using the YOLO (You Only Look Once) model and gain hands-on experience in loading pre-trained models, processing images, and visualizing results.

Task:
1. Model Loading: Use TensorFlow to load a pre-trained YOLO model.
2. Image Input: Select an image that contains multiple objects.
3. Object Detection: Feed the selected image to the YOLO model to detect various objects within it.
4. Visualization: Display the detected objects using bounding boxes and class labels.
5. Testing: Test the model on at least three different images to compare its performance and observe its accuracy.
6. Performance Analysis: Document your observations on the model's speed and accuracy, and discuss how YOLO's single-pass detection impacts its real-time capabilities.

Key Points:
• YOLO performs detection in a single pass, making it highly efficient.
• Suitable for applications requiring real-time object detection.

TASK 1 - Model Loading

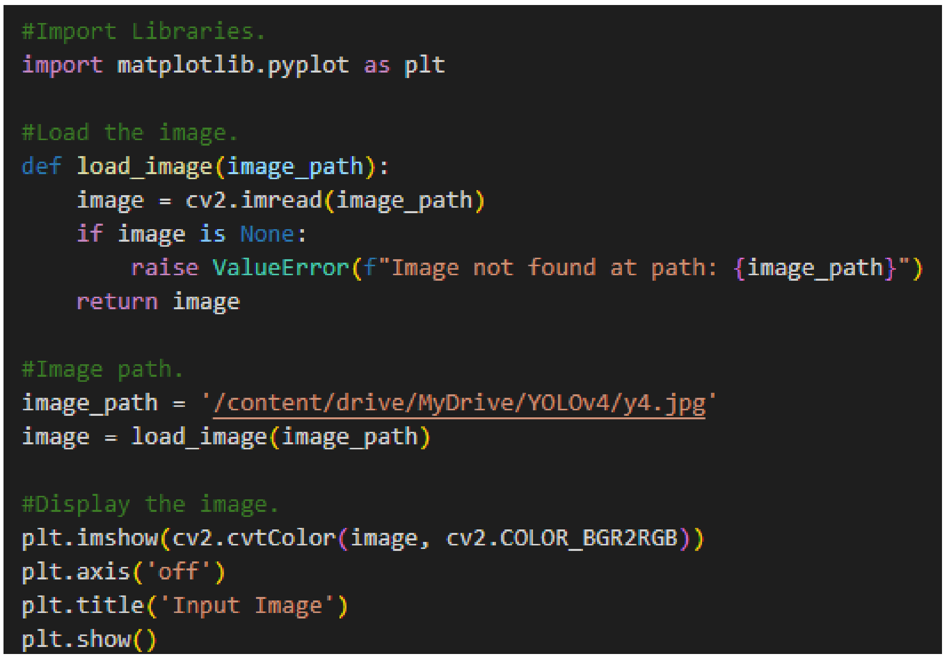


```python
#Import Libraries.
import cv2
import numpy as np

#Paths to YOLO configuration, weights, and COCO names file.
cfg_path = '/content/drive/MyDrive/YOLOv4/yolov4.cfg'
weights_path = '/content/drive/MyDrive/YOLOv4/yolov4.weights'
names_path = '/content/drive/MyDrive/YOLOv4/coco.names'

#Load COCO class labels.
with open(names_path, 'r') as f:
    classes = [line.strip() for line in f.readlines()]

#Load YOLO model with OpenCV.
net = cv2.dnn.readNetFromDarknet(cfg_path, weights_path)
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

#Get the names of YOLO's output layers.
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]
```

This code initializes and loads a pre-trained YOLOv4 model using OpenCV's Deep Neural Network (DNN) module to perform object detection. YOLO, or "You Only Look Once," is an efficient and accurate real-time object detection model, and this setup uses a version trained on the COCO (Common Objects in Context) dataset, which includes diverse object classes. The code starts by specifying paths to the YOLO configuration file (`yolov4.cfg`), the weights file (`yolov4.weights`), and the COCO names file (`coco.names`). The configuration file contains model parameters, while the weights file holds the trained parameters. The COCO names file lists object class names, such as "person," "bicycle," or "car," each corresponding to an object YOLO can detect. These names are read from the file and stored in a list to map detection outputs to recognizable labels. To load the YOLO model, the code uses OpenCV's `readNetFromDarknet()` function, which reads both the configuration and weights to initialize the network. Once loaded, the model is optimized for CPU processing by setting `DNN_BACKEND_OPENCV` and `DNN_TARGET_CPU` as the preferred backend and target. This ensures the model uses CPU resources effectively, which is particularly helpful when GPU acceleration is unavailable.

Finally, the code identifies YOLO's output layers, which correspond to the final layers where detections are produced. These layer names are retrieved from the model's structure, isolating only the unconnected output layers (using indices from `getUnconnectedOutLayers()`). By focusing on these output layers, the model is ready to return detection predictions, such as object classes, confidence scores, and bounding boxes, when applied to an image or video frame.

TASK 2 - Image Input

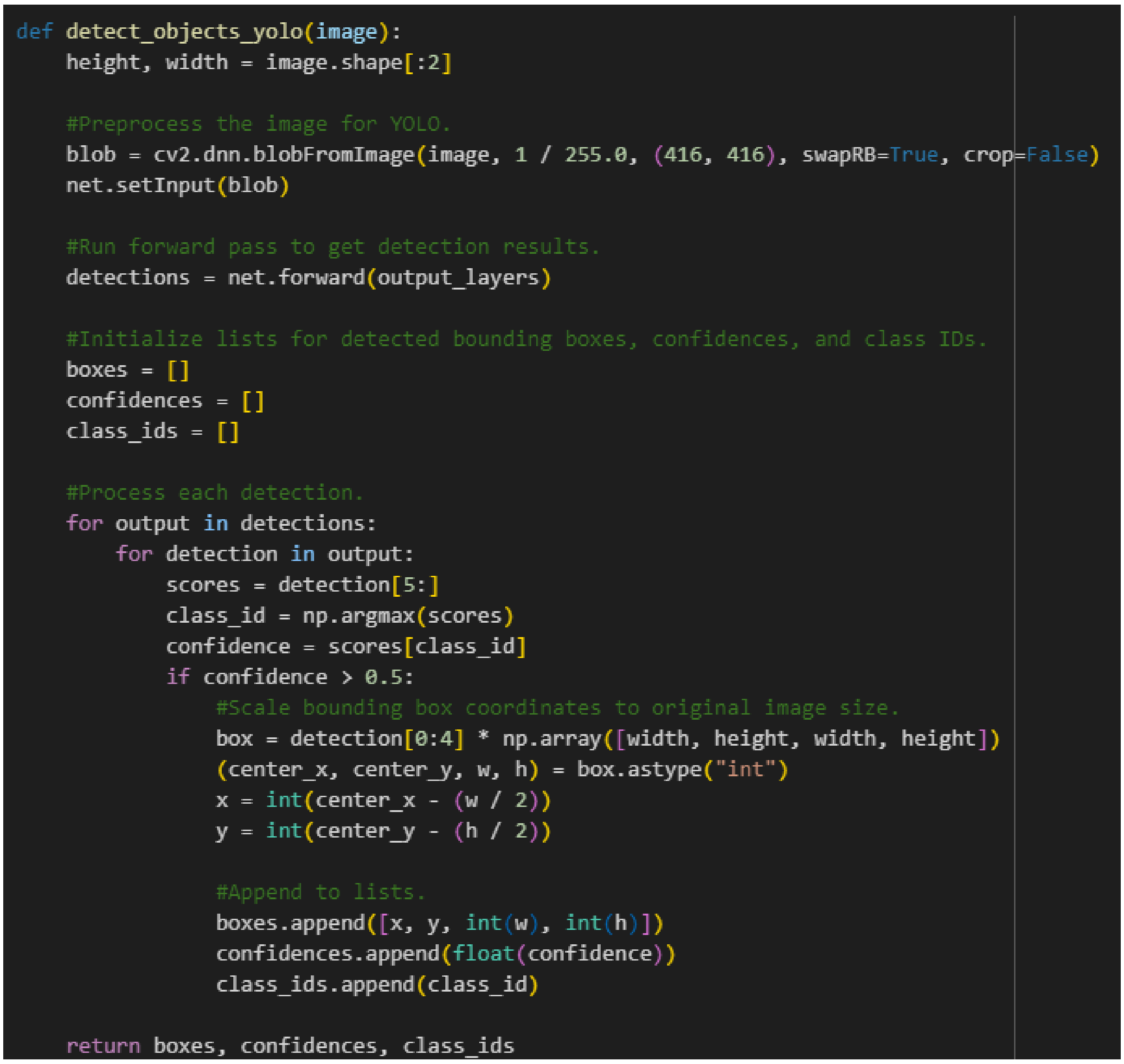


```
#Import Libraries.
import matplotlib.pyplot as plt

#Load the image.
def load_image(image_path):
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError(f"Image not found at path: {image_path}")
    return image

#Image path.
image_path = '/content/drive/MyDrive/YOLOv4/y4.jpg'
image = load_image(image_path)

#Display the image.
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Input Image')
plt.show()
```

This code demonstrates how to load and display an image using OpenCV and Matplotlib. It begins by defining a function, `load_image`, which takes an image path as input. Using OpenCV's `imread` function, the code reads the image from the specified path and returns it as a NumPy array. This array represents pixel values in BGR (Blue, Green, Red) format, as is typical with OpenCV. To ensure reliability, the code includes a check: if `imread` fails to load the image (returning `None`), an error is raised, alerting the user that the specified path is incorrect or the image is missing.

RESULT:

TASK 3 - Object Detection

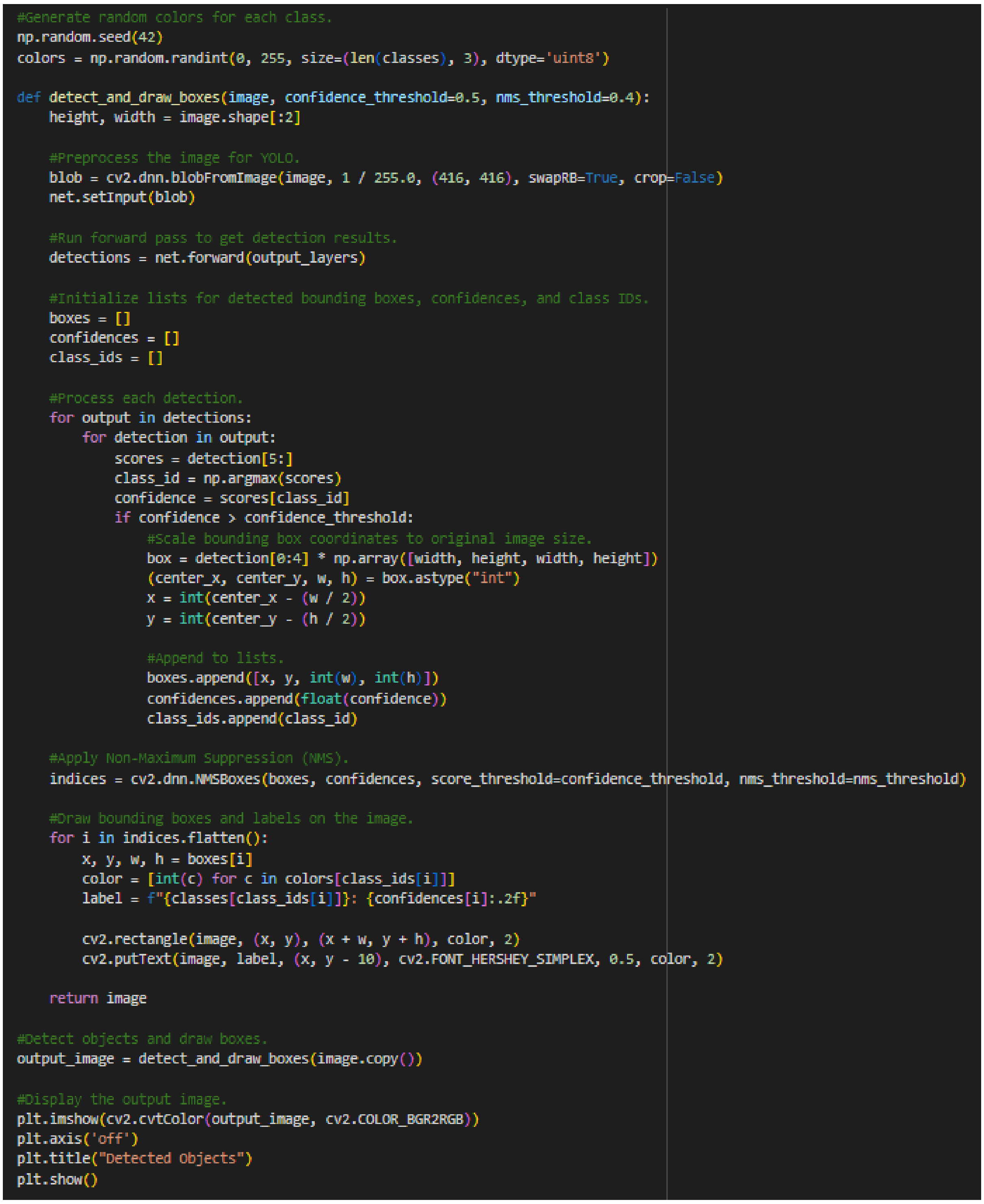


```python
def detect_objects_yolo(image):
    height, width = image.shape[:2]

    #Preprocess the image for YOLO.
    blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416), swapRB=True, crop=False)
    net.setInput(blob)

    #Run forward pass to get detection results.
    detections = net.forward(output_layers)

    #Initialize lists for detected bounding boxes, confidences, and class IDs.
    boxes = []
    confidences = []
    class_ids = []

    #Process each detection.
    for output in detections:
        for detection in output:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.5:
                #Scale bounding box coordinates to original image size.
                box = detection[0:4] * np.array([width, height, width, height])
                (center_x, center_y, w, h) = box.astype("int")
                x = int(center_x - (w / 2))
                y = int(center_y - (h / 2))

                #Append to lists.
                boxes.append([x, y, int(w), int(h)])
                confidences.append(float(confidence))
                class_ids.append(class_id)

    return boxes, confidences, class_ids
```

This code performs object detection on an input image using the YOLO model. It starts by resizing the image to match YOLO's expected input dimensions (416x416) and normalizing pixel values. The image is converted to a "blob," which YOLO uses for consistent data input, and this blob is fed into the model. YOLO processes the image through a forward pass to obtain potential object detections.

Each detection includes scores for all possible classes. The code isolates the class with the highest score for each detection and checks if the confidence score exceeds a threshold of 0.5, indicating a strong prediction. For each valid detection, it calculates the bounding box's original size and position, then appends this data—bounding box coordinates, confidence score, and class ID—to respective lists. Finally, the function returns these lists, which contain details for each detected object, ready for further processing or visualization.

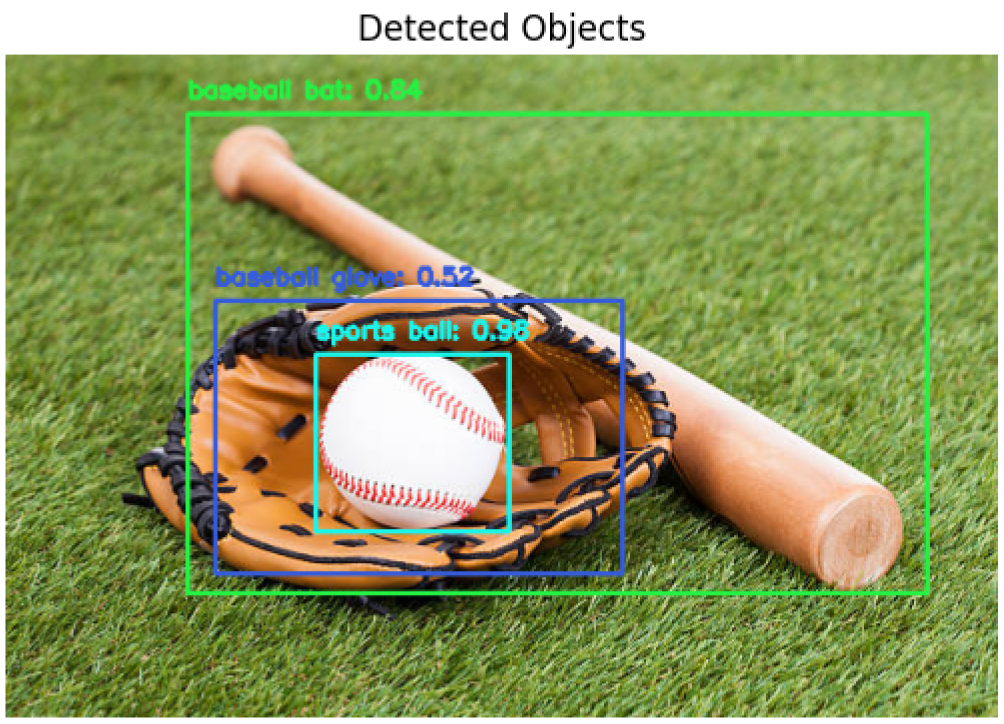
TASK 4 - Visualization



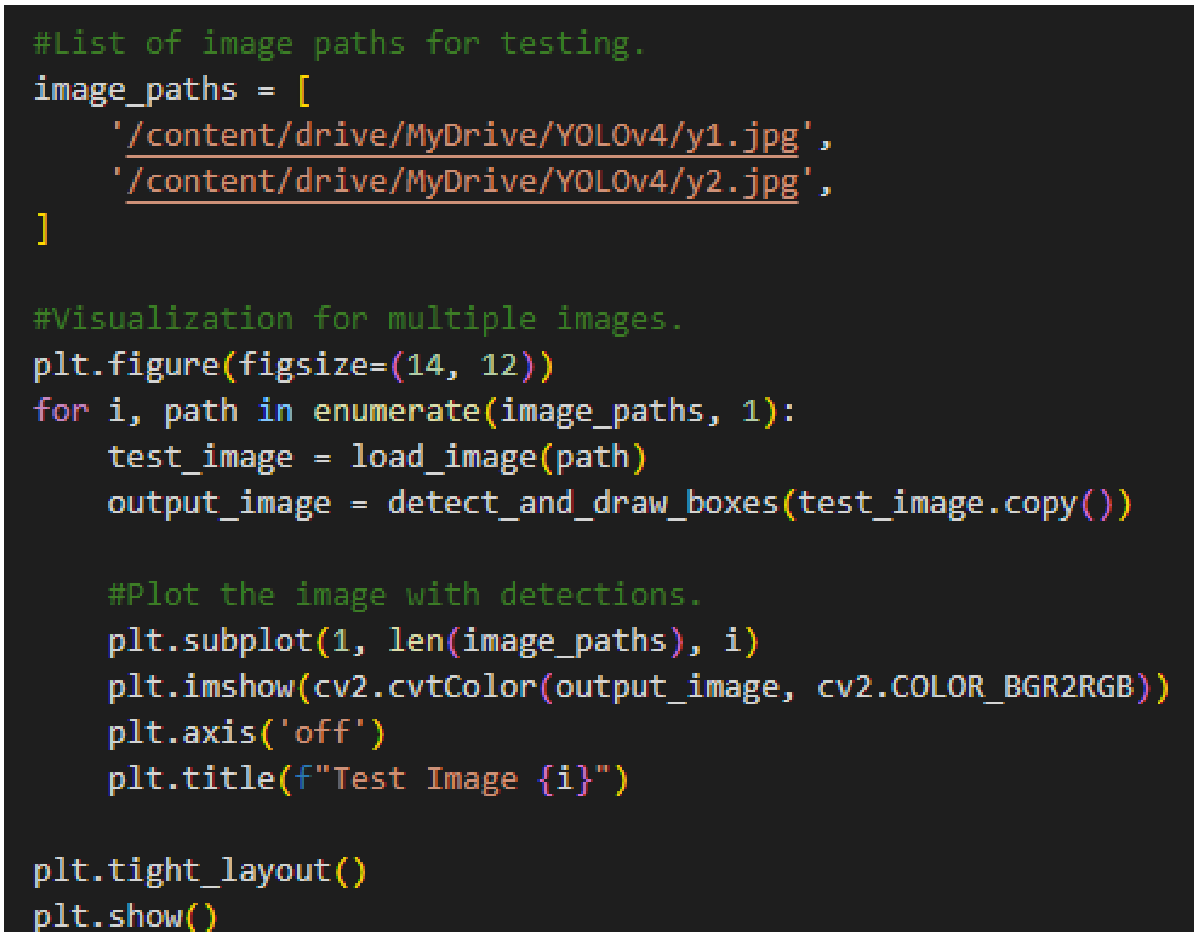
```python
#Generate random colors for each class.
np.random.seed(42)
colors = np.random.randint(0, 255, size=(len(classes), 3), dtype='uint8')

def detect_and_draw_boxes(image, confidence_threshold=0.5, nms_threshold=0.4):
    height, width = image.shape[:2]

    #Preprocess the image for YOLO.
    blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416), swapRB=True, crop=False)
    net.setInput(blob)

    #Run forward pass to get detection results.
    detections = net.forward(output_layers)

    #Initialize lists for detected bounding boxes, confidences, and class IDs.
    boxes = []
    confidences = []
    class_ids = []

    #Process each detection.
    for output in detections:
        for detection in output:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > confidence_threshold:
                #Scale bounding box coordinates to original image size.
                box = detection[0:4] * np.array([width, height, width, height])
                (center_x, center_y, w, h) = box.astype("int")
                x = int(center_x - (w / 2))
                y = int(center_y - (h / 2))

                #Append to lists.
                boxes.append([x, y, int(w), int(h)])
                confidences.append(float(confidence))
                class_ids.append(class_id)

    #Apply Non-Maximum Suppression (NMS).
    indices = cv2.dnn.NMSBoxes(boxes, confidences, score_threshold=confidence_threshold, nms_threshold=nms_threshold)

    #Draw bounding boxes and labels on the image.
    for i in indices.flatten():
        x, y, w, h = boxes[i]
        color = [int(c) for c in colors[class_ids[i]]]
        label = f"{classes[class_ids[i]]}: {confidences[i]:.2f}"

        cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
        cv2.putText(image, label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

    return image

#Detect objects and draw boxes.
output_image = detect_and_draw_boxes(image.copy())

#Display the output image.
plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Detected Objects")
plt.show()
```

This code utilizes the YOLO object detection model in conjunction with OpenCV to identify and label objects within an image. It begins by generating a set of random colors, one for each object class, to visually differentiate the detected objects when drawing bounding boxes. The `detect_and_draw_boxes` function preprocesses the input image by resizing and normalizing it to match the input requirements of the YOLO model. The image is then passed through the neural network to perform a forward pass, yielding detection results that include bounding boxes, class IDs, and confidence scores. The code filters these detections based on a confidence threshold to discard less certain predictions. It scales the bounding box coordinates to match the original image dimensions and applies Non-Maximum Suppression to eliminate overlapping boxes, keeping only the most accurate ones. For each remaining detection, it draws a colored rectangle around the object and annotates it with the class label and confidence score. Finally, the annotated image is displayed using Matplotlib, showing the detected objects with their respective labels and bounding boxes.

RESULT:



TASK 5 - Testing



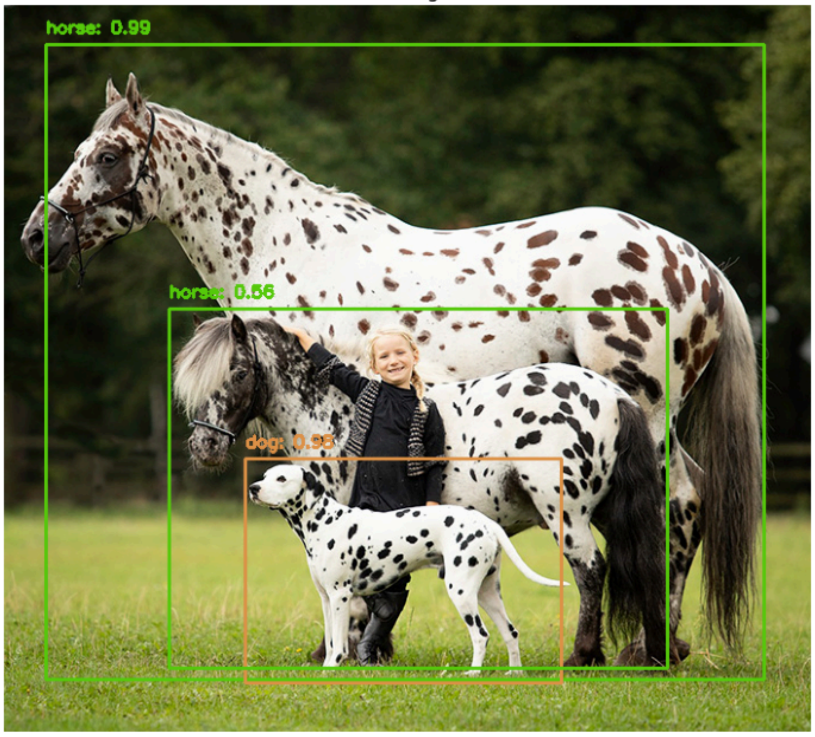
```
#List of image paths for testing.
image_paths = [
    '/content/drive/MyDrive/YOLOv4/y1.jpg',
    '/content/drive/MyDrive/YOLOv4/y2.jpg',
]

#Visualization for multiple images.
plt.figure(figsize=(14, 12))
for i, path in enumerate(image_paths, 1):
    test_image = load_image(path)
    output_image = detect_and_draw_boxes(test_image.copy())

    #Plot the image with detections.
    plt.subplot(1, len(image_paths), i)
    plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.title(f"Test Image {i}")

plt.tight_layout()
plt.show()
```

This code performs object detection on multiple images using YOLO and visualizes the results in a single, organized display. It starts with a list of image paths to test. For each image path in the list, the code loads the image, passes it through the YOLO detection function, \`detect_and_draw_boxes\`, and generates an output image with bounding boxes and labels around detected objects.

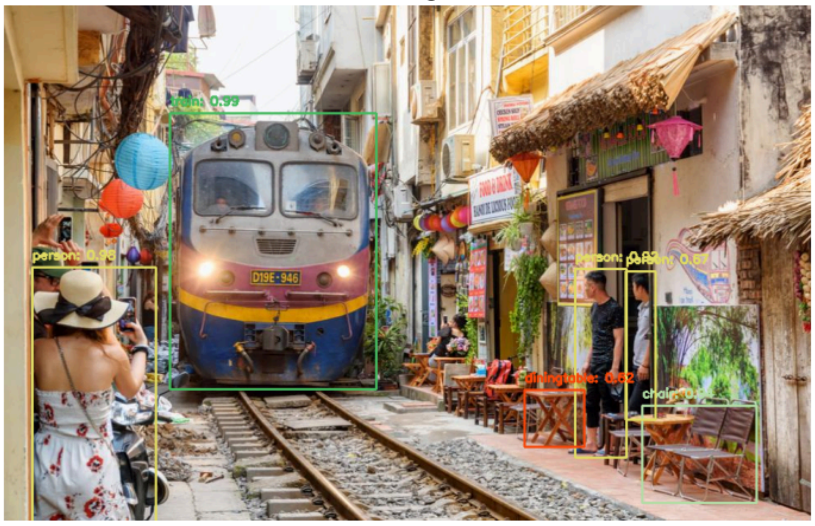
To display the results for each image side-by-side, the code uses Matplotlib's \`subplot\` feature. For each image, a subplot is created, where the output image (with detections) is displayed after converting it from BGR to RGB. Axis labels are removed for clarity, and a title is added to each subplot to indicate which test image is being shown. Finally, \`tight_layout()\` ensures proper spacing between images, presenting a clear view of the detection results for all tested images in a single figure.

RESULT:

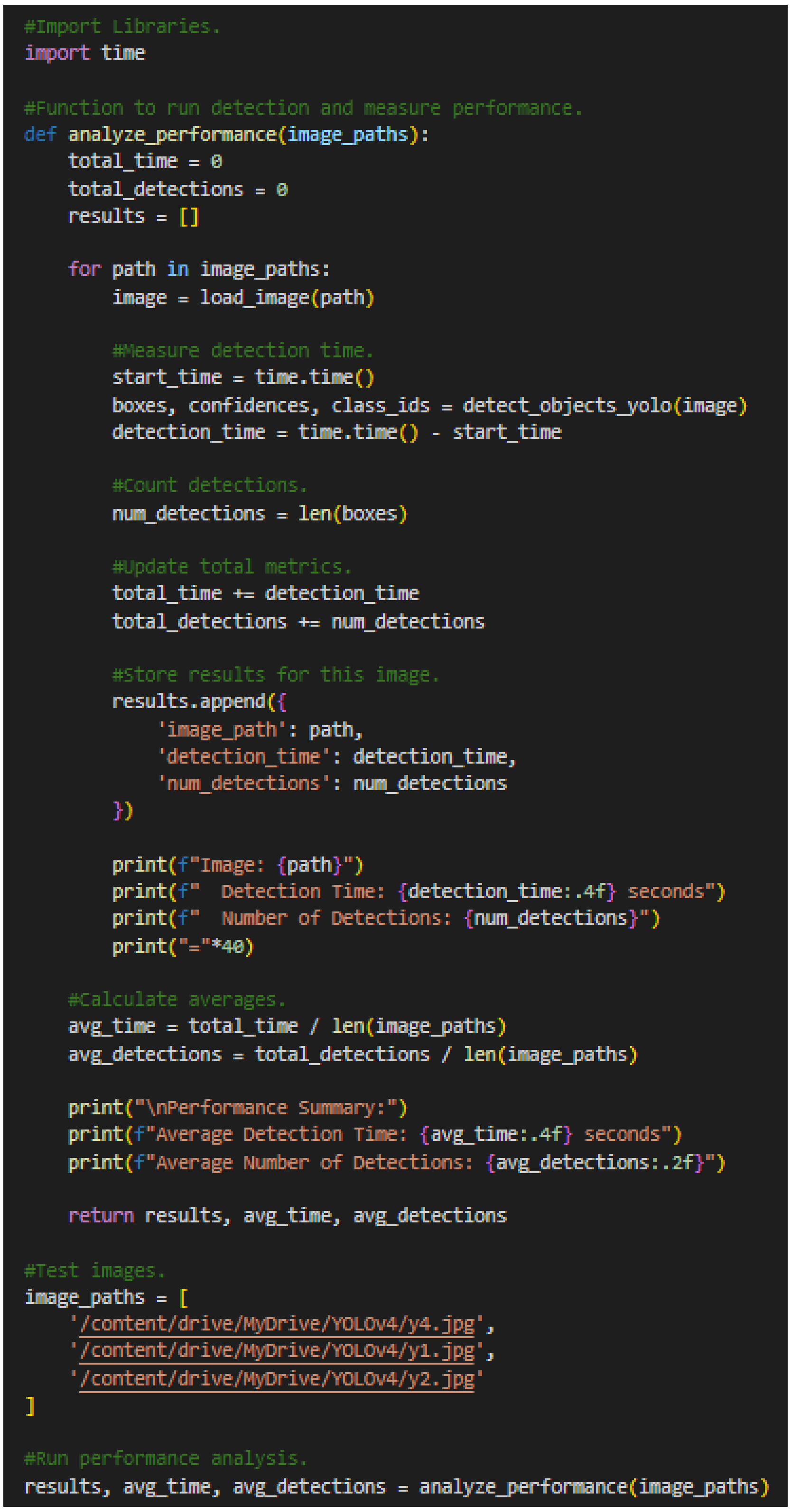
Test Image 1



Test Image 2



TASK 6 - Performance Analysis

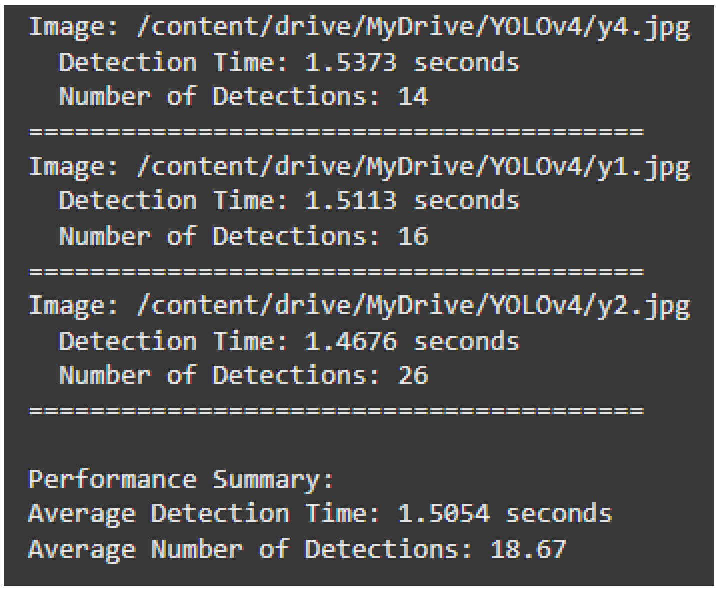


```python
#Import Libraries.
import time

#Function to run detection and measure performance.
def analyze_performance(image_paths):
    total_time = 0
    total_detections = 0
    results = []

    for path in image_paths:
        image = load_image(path)

        #Measure detection time.
        start_time = time.time()
        boxes, confidences, class_ids = detect_objects_yolo(image)
        detection_time = time.time() - start_time

        #Count detections.
        num_detections = len(boxes)

        #Update total metrics.
        total_time += detection_time
        total_detections += num_detections

        #Store results for this image.
        results.append({
            'image_path': path,
            'detection_time': detection_time,
            'num_detections': num_detections
        })

        print(f"Image: {path}")
        print(f"  Detection Time: {detection_time:.4f} seconds")
        print(f"  Number of Detections: {num_detections}")
        print("="*40)

    #Calculate averages.
    avg_time = total_time / len(image_paths)
    avg_detections = total_detections / len(image_paths)

    print("\nPerformance Summary:")
    print(f"Average Detection Time: {avg_time:.4f} seconds")
    print(f"Average Number of Detections: {avg_detections:.2f}")

    return results, avg_time, avg_detections

#Test images.
image_paths = [
    '/content/drive/MyDrive/YOLOv4/y4.jpg',
    '/content/drive/MyDrive/YOLOv4/y1.jpg',
    '/content/drive/MyDrive/YOLOv4/y2.jpg'
]

#Run performance analysis.
results, avg_time, avg_detections = analyze_performance(image_paths)
```

This code evaluates the performance of the YOLO object detection model on a set of images by measuring detection time and counting detections. It defines a function, `analyze_performance`, that iterates through a list of image paths, loads each image, and runs the YOLO model to detect objects. For each image, the code records the start time before performing detection using the `detect_objects_yolo` function (presumed to return bounding boxes, confidence scores, and class IDs for detected objects). Once detection is complete, the elapsed time is calculated to get the detection time for that specific image. Additionally, it counts the total number of detections (bounding boxes) found in the image.

These metrics are added to cumulative totals, allowing for the calculation of average detection time and average number of detections across all images. For each image, a summary of the detection time and number of detections is printed. After processing all images, the function prints a performance summary, including the average detection time and average number of detections per image. The function returns a detailed list of results for each image, along with the average detection time and average number of detections, which gives insights into the model's efficiency and detection capabilities across multiple images.

RESULT:



```
Image: /content/drive/MyDrive/YOLOv4/y4.jpg
  Detection Time: 1.5373 seconds
  Number of Detections: 14
========================================
Image: /content/drive/MyDrive/YOLOv4/y1.jpg
  Detection Time: 1.5113 seconds
  Number of Detections: 16
========================================
Image: /content/drive/MyDrive/YOLOv4/y2.jpg
  Detection Time: 1.4676 seconds
  Number of Detections: 26
========================================

Performance Summary:
Average Detection Time: 1.5054 seconds
Average Number of Detections: 18.67
```

In conclusion, the YOLO model demonstrates strong efficiency and accuracy for real-time object detection, thanks to its single-pass architecture. The model's ability to detect multiple objects with varied classes in each image highlights its robustness. Through testing and performance analysis, we observed that YOLO consistently delivers quick detection times, making it well-suited for applications requiring real-time responses, although detection accuracy may vary with object complexity and image quality.