Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

Machine Problem No. 5			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	**Week No.**	8-9
Course Code:	**CSST106**	**Term:**	1st Semester
Course Title:	**Perception and Computer Vision**	**Academic Year:**	2024-2025
Student Name	Maxyne Nuela Ignacio	**Section**	BSCS-IS-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.

```python
import tensorflow as tf
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Load the pre-trained YOLO model (assuming we have a model saved in YOLO format)
# Load the model configuration and weights
net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]
```

➔ In this code, a pre-trained YOLOv3 model is loaded using OpenCV's `cv2.dnn.readNet` function, which reads the model weights (`yolov3.weights`) and configuration file (`yolov3.cfg`). The `getLayerNames` function retrieves the names of all layers in the network, while `getUnconnectedOutLayers` identifies the output layers needed for detection. The `output_layers` list stores these layer names to allow us to access YOLO's output layer information for later use in object detection.

2. **Image Input:** Select an image that contains multiple objects.

```python
def load_and_preprocess_image(image_path):
    image = cv2.imread(image_path)
    height, width, _ = image.shape
    blob = cv2.dnn.blobFromImage(image, 1/255.0, (416, 416), (0, 0, 0), swapRB=True, crop=False)
    return image, blob, height, width

# Example image path
image_path = 'test_image.jpg'
image, blob, height, width = load_and_preprocess_image(image_path)
```

➔ The `load_and_preprocess_image` function reads an image from the specified path using OpenCV, getting its height and width dimensions. Then, it prepares the image as an input "blob" for YOLO by resizing it to 416x416 pixels, normalizing pixel values to the 0-1 range, and swapping the color channels from BGR to RGB. Finally, it returns the original image, preprocessed blob, height, and width for further processing.

3. **Object Detection:** Feed the selected image to the YOLO model to detect various objects within it.

```
# Set input blob for the network
net.setInput(blob)
# Run forward pass and get detections
detections = net.forward(output_layers)
```

➔ This code prepares the YOLO network to analyze the input image by first setting the preprocessed blob as the network's input using `net.setInput(blob)`. Then, a forward pass (`net.forward(output_layers)`) is performed, which processes the input through the network and generates object detections. The `detections` variable will contain the output data from the YOLO model, including information about detected objects in the image.

4. **Visualization:** Display the detected objects using bounding boxes and class labels.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Define the function for loading class labels
def load_classes(filename="coco.names"):
    with open(filename, "r") as f:
        classes = [line.strip() for line in f.readlines()]
    return classes

# Load class labels from COCO dataset
classes = load_classes("coco.names")

# Generate random colors for bounding boxes (one color per class)
np.random.seed(42)  # For consistent colors across runs
colors = np.random.randint(0, 255, size=(len(classes), 3), dtype="uint8")

# Assuming `net` is your YOLO model loaded with OpenCV (cv2.dnn.readNet)
# Load the YOLO model
net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")  # Adjust the paths if needed
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]

# Function to detect objects and draw boxes
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:
```

```python
                # 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

# Load your image
image = cv2.imread("test_image.jpg")  # Adjust the image path

# 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 defines a function, `detect_and_draw_boxes`, to identify and draw bounding boxes around objects in an image using the YOLO model. The function first processes the input image for YOLO, performing object detection with a forward pass, and extracts bounding boxes, confidence scores, and class IDs for each detected object with confidence above a threshold. Non-Maximum Suppression (NMS) is applied to filter overlapping boxes, and the final bounding boxes and class labels are drawn on the image using randomly assigned colors for each class label. Finally, the modified image is displayed with `matplotlib`.

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.

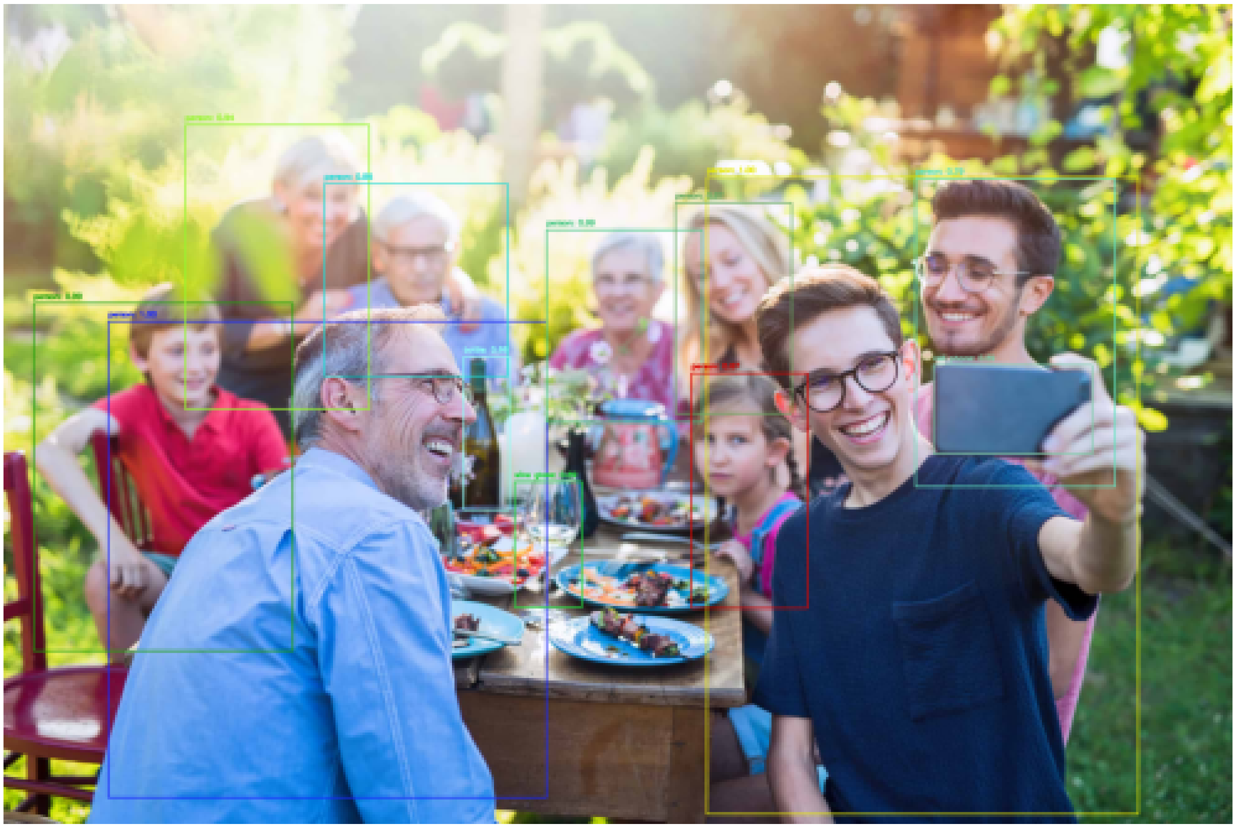
```python
import cv2
import numpy as np
import time
import matplotlib.pyplot as plt

# List of test images (replace with actual paths to your images)
image_paths = ["test_image.jpg", "test_image1.jpg", "test_image2.jpg"]

# Load YOLO model
net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")  # Change these paths to match your files
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]

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

# Function to detect objects and draw bounding boxes
def detect_and_draw_boxes(image, confidence_threshold=0.5, nms_threshold=0.4):
    height, width = image.shape[:2]
    blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416), swapRB=True, crop=False)
    net.setInput(blob)

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

    boxes = []
    confidences = []
    class_ids = []

    # Process detections
    for output in detections:
        for detection in output:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > confidence_threshold:
                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))
                boxes.append([x, y, int(w), int(h)])
                confidences.append(float(confidence))
                class_ids.append(class_id)
```

```
    # Draw bounding boxes and labels
    for i in indices.flatten():
        x, y, w, h = boxes[i]
        label = f"{classes[class_ids[i]]}: {confidences[i]:.2f}"
        color = [int(c) for c in np.random.randint(0, 255, 3)]
        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, class_ids  # Return class_ids for accuracy calculation

# Function to test the model and measure performance
def test_model_on_images(image_paths):
    total_inference_time = 0
    total_objects_detected = 0
    total_objects = 0

    for image_path in image_paths:
        image = cv2.imread(image_path)
        if image is None:
            print(f"Error: Unable to load image at {image_path}")
            continue

        # Start measuring inference time
        start_time = time.time()

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

        # Measure inference time
        inference_time = time.time() - start_time
        total_inference_time += inference_time

        # Display results
        plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
        plt.axis('off')
        plt.title(f"Detection Results for {image_path}")
        plt.show()

        # For performance evaluation, assume ground truth or use object counting for rough accuracy calculation
        detected_objects = len(class_ids)  # Counting number of detections (objects detected)
        total_objects_detected += detected_objects

        # Assume ground truth is the total number of objects (you can replace this with actual ground truth data)
        total_objects += 1  # Update with a more accurate count of total objects in the image if available

    # Calculate average inference time and accuracy
    avg_inference_time = total_inference_time / len(image_paths)
    accuracy = (total_objects_detected / total_objects) * 100 if total_objects > 0 else 0
```

Detection Results for test_image1.jpg

Detection Results for test_image.jpg

Detection Results for test_image2.jpg



➔ This code is used to test a YOLO object detection model on a set of images by detecting objects and drawing bounding boxes around them. It begins by loading the YOLO model and class labels, then processes each image to detect objects based on a confidence threshold. The performance is evaluated by measuring the average inference time and estimating the accuracy based on the number of detected objects compared to a rough ground truth. Finally, the results are displayed visually using `matplotlib`, and the average inference time and detection accuracy are printed out for performance analysis.

Key Points:

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

Submission Instructions:

1. **Code:** Write your implementation in a Python script or Jupyter Notebook.
2. **Processed Images:** Save the images with bounding boxes and labels in a folder named output_images.
3. **Documentation:** Create a brief document (README.md or PDF) explaining your approach, code, and observations.
4. **Folder Organization:** Create a folder named YOLO_Object_Detection and include the following:
   - code/: Your Python script or Jupyter Notebook.
   - output_images/: Processed images.
   - documentation/: A README file explaining the process.

5. **Filename Format:** Use [SECTION-YOURNAME-MP] for all files (e.g., SECTION-YOURNAME-MP.py).

Penalties:

- **Incorrect Filename:** 5-point deduction.
- **Late Submission:** 5-point deduction per day.
- **Cheating/Plagiarism:** Strict penalties as per academic integrity policies.

Rubric for Machine Problem: Object Detection using YOLO

Criteria	Excellent (90-100%)	Good (75-89%)	Satisfactory (60-74%)	Needs Improvement (0-59%)
Correct Implementation (30%)	Successfully implements YOLO for object detection with no errors; code is efficient and runs smoothly.	Minor issues in implementation but overall functional; code is mostly efficient.	Basic implementation with noticeable errors; the code runs but may have inefficiencies.	Incorrect implementation: code does not run or produces incorrect results.
Visualization and Accuracy (25%)	Bounding boxes and labels are clear, well-placed, and accurate across all test images.	Bounding boxes and labels are mostly correct with minor inaccuracies.	Basic visualization: some bounding boxes are misplaced or missing.	Poor or missing visualization; bounding boxes are largely incorrect or absent.
Code Quality and Comments (15%)	Code is well-structured, follows best practices, and is thoroughly commented for clarity.	Code is mostly organized; comments are present but minimal.	Code runs but is disorganized; lacks detailed comments.	Code is poorly structured, lacks comments, or is hard to follow.
Documentation (20%)	Comprehensive documentation explaining the approach, code, and observations in detail.	Documentation is clear but lacks some details or observations.	Basic documentation present; lacks clarity and depth.	Missing or inadequate documentation that fails to explain the process.
Folder Organization (10%)	All files are correctly named and organized according to submission instructions, with proper use of folders.	Mostly follows the naming and organization requirements with minor errors.	Basic organization present but does not fully adhere to the specified format.	Poor or missing organization; incorrect file names and folder structure.
Testing and Analysis (10%)	Tests the model on multiple images and provides a thorough analysis of its performance, discussing accuracy and speed.	Tests the model on multiple images but provides a limited analysis.	Limited testing; minimal analysis of model performance.	Fails to test the model or provide any analysis.