EXERCISE NO. 4			
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:	

Object Detection and Recognition

Exercise 1: HOG (Histogram of Oriented Gradients) Object Detection

Task: HOG is a feature descriptor widely used for object detection, particularly for human detection. In this exercise, you will:

- Load an image containing a person or an object.
- Convert the image to grayscale.
- Apply the HOG descriptor to extract features.
- Visualize the gradient orientations on the image.
- Implement a simple object detector using HOG features.

Key Points:

- HOG focuses on the structure of objects through gradients.
- Useful for detecting humans and general object recognition.

HOG (Histogram of Oriented Gradients) Object Detection

The histogram of oriented gradients method is a feature descriptor technique used in computer vision and image processing for object detection. It focuses on the shape of an object, counting the occurrences of gradient orientation in each local region. It then generates a histogram using the magnitude and orientation of the gradient.

CODE:

```
#Import Libraries.
import cv2
from skimage import feature, exposure
import matplotlib.pyplot as plt
import numpy as np

#Load the image.
image_path = '/content/drive/MyDrive/iu.jpg'
image = cv2.imread(image_path)
if image is None:
    raise ValueError("Image not found at the specified path.")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

#Convert the image to grayscale.
gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

#Apply HOG descriptor to extract features.
hog_features, hog_image = feature.hog(
    gray_image,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    block_norm='L2-Hys',
    visualize=True
)

#Adjust the intensity of the HOG image for better visualization.
hog_image = exposure.rescale_intensity(hog_image, in_range=(0, 15))

#Visualize the original and HOG image.
plt.figure(figsize=(13, 10))

#Display the original image.
plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

#Display the HOG visualization image.
plt.subplot(1, 2, 2)
plt.imshow(hog_image, cmap='gray')
plt.title('HOG Visualization')
plt.axis('off')

plt.show()
```

This code demonstrates the process of applying the Histogram of Oriented Gradients (HOG) descriptor to an image for feature extraction and visualization. Initially, necessary libraries like OpenCV, scikit-image, and Matplotlib are imported. The code begins by loading an image from a specified path; if the image cannot be found, it raises an error. After loading, the image is converted to the RGB color format and subsequently to grayscale, as HOG calculations are typically applied to grayscale images for efficiency.

The `feature.hog` function from the `skimage` library is then used to compute the HOG features of the grayscale image. This function breaks the image down into cells and blocks, calculates gradients in different orientations, and organizes the gradients into histograms. Key parameters used here include 9 orientations, an 8x8 pixel grid per cell, and a 2x2 block size for normalization, with the `L2-Hys` norm to normalize the block. The

`visualize=True` option enables the function to generate a visual representation of the HOG descriptor, which provides an abstracted view of the edges and orientations present in the image.

To enhance the HOG image for better visibility, the `exposure.rescale_intensity` function scales the intensity values to a specific range, making the gradient information more discernible. Finally, Matplotlib is used to display both the original grayscale image and the processed HOG image side by side. The HOG visualization highlights edges and gradients in the image, effectively capturing structural features that would be useful for tasks like object recognition.

RESULT:

Exercise 2: YOLO (You Only Look Once) Object Detection

Task: YOLO is a deep learning-based object detection method. In this exercise, you will:

- Load a pre-trained YOLO model using TensorFlow.
- Feed an image to the YOLO model for object detection.
- Visualize the bounding boxes and class labels on the detected objects in the image.
- Test the model on multiple images to observe its performance.

Key Points:

- YOLO is fast and suitable for real-time object detection.
- It performs detection in a single pass, making it efficient for complex scenes.

YOLO (You Only Look Once) Object Detection

You Only Look Once or YOLO is an algorithm capable of detecting objects at first glance, performing detection and classification simultaneously. Find out all you need to know about this revolutionary approach to AI and Computer Vision

CODE:

```python
#Import Libraries.
import cv2
import numpy as np
import matplotlib.pyplot as plt

#Set the paths to YOLOv4 configuration, weights, and coco.names files.
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 the COCO class labels.
with open(names_path, 'r') as f:
    classes = [line.strip() for line in f.readlines()]

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

#Load Pre-trained YOLOv4 model.
net = cv2.dnn.readNet(cfg_path, weights_path)

#Set up YOLOv4 layer names.
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers().flatten()]

#Define a function for object detection.
def detect_objects(image_path):
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError(f"Image not found at the specified path: {image_path}")

    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    height, width = image.shape[:2]

    #Preprocess the image for YOLOv4.
    blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416), swapRB=True, crop=False)
    net.setInput(blob)

    #Perform forward pass and get detection results.
    detections = net.forward(output_layers)

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

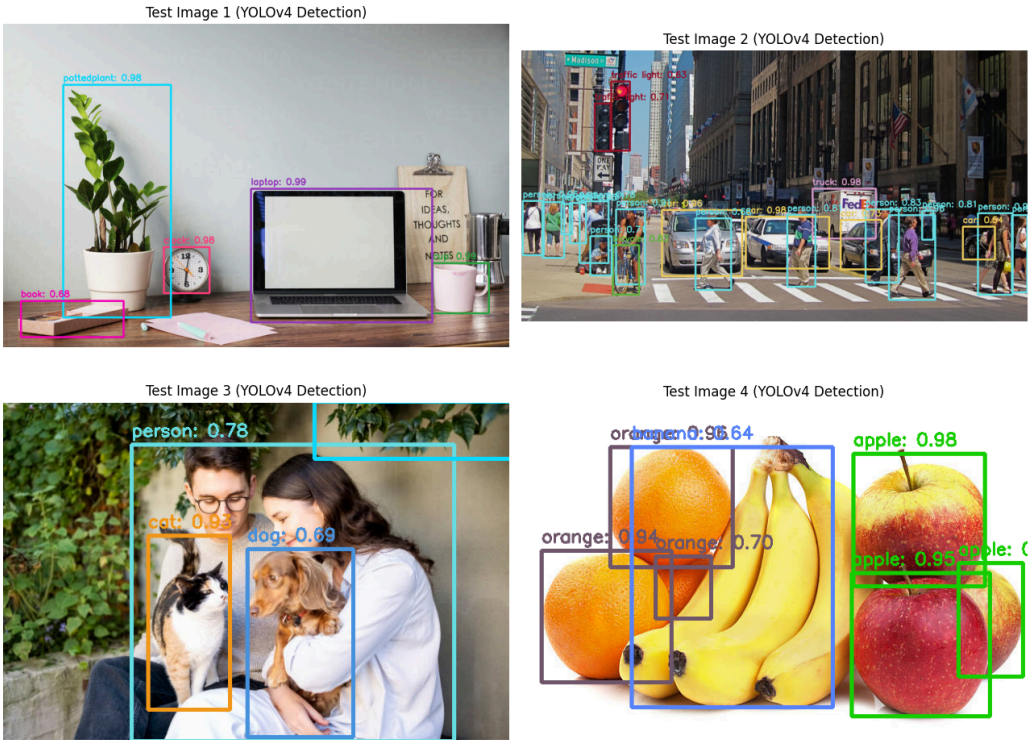
```
    #Process detections.
    for detection in detections:
        for obj in detection:
            scores = obj[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]

            if confidence > 0.5:
                center_x = int(obj[0] * width)
                center_y = int(obj[1] * height)
                w = int(obj[2] * width)
                h = int(obj[3] * height)
                x = int(center_x - w / 2)
                y = int(center_y - h / 2)

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

    #Apply Non-Maximum Suppression to remove overlapping boxes.
    indices = cv2.dnn.NMSBoxes(boxes, confidences, score_threshold=0.5, nms_threshold=0.4)

    #Draw bounding boxes and labels on the image with thicker lines and larger text.
    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_rgb, (x, y), (x + w, y + h), color, 3)
        cv2.putText(image_rgb, label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)

    return image_rgb

#Multiple images to test the model.
image_paths = [
    '/content/drive/MyDrive/YOLOv4/office.jpg',
    '/content/drive/MyDrive/YOLOv4/crosswalk.JPG',
    '/content/drive/MyDrive/YOLOv4/animals.jpg',
    '/content/drive/MyDrive/YOLOv4/fruits.jpg'
]

#Visualize the boidung boxes and class labels.
plt.figure(figsize=(13, 10))

for i, path in enumerate(image_paths, 1):
    detected_image = detect_objects(path)
    plt.subplot(2, 2, i)
    plt.imshow(detected_image)
    plt.axis('off')
    plt.title(f"Test Image {i} (YOLOv4 Detection)")

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

This code snippet performs object detection on multiple images using the YOLOv4 (You Only Look Once version 4) model, which is designed for fast and accurate object detection. Initially, the necessary libraries (OpenCV, NumPy, and Matplotlib) are imported. The paths to the YOLO configuration, pre-trained weights, and class labels are set, allowing the model to recognize objects in the images based on the COCO dataset (a popular large-scale object detection dataset). The class labels are loaded, and random colors are assigned to each class for visual distinction in the output. The YOLOv4 model is then loaded with OpenCV's DNN (Deep Neural Network) module, and the output layers are specified. A

function, `detect_objects`, is defined to handle the detection process. It takes an image path as input, loads and preprocesses the image by converting it to RGB and resizing it to fit the model's input dimensions. A blob is created from the image for efficient processing by the model. The network performs a forward pass, generating a list of detections, where each detection contains confidence scores and bounding box information for various classes. If an object's confidence score surpasses a threshold of 0.5, its bounding box coordinates and class ID are stored. Non-Maximum Suppression (NMS) is then applied to remove overlapping boxes, enhancing detection accuracy.

Each detected object is highlighted by a colored bounding box, and a label indicating the object's class and confidence score is drawn. This is done with thicker lines and larger text for improved readability. To test the model, a list of image paths is defined. For each image, the detection function is called, and the results are displayed using Matplotlib. Each image is shown in a grid format, making it easy to visualize the model's performance on different scenes, from an office to a crosswalk and nature. This setup provides a straightforward, reproducible approach to object detection on multiple images using YOLOv4.

RESULT:



Exercise 3: SSD (Single Shot MultiBox Detector) with TensorFlow

Task: SSD is a real-time object detection method. For this exercise:

• Load an image of your choice.

• Utilize the TensorFlow Object Detection API to apply the SSD model.

• Detect objects within the image and draw bounding boxes around them.
• Compare the results with those obtained from the YOLO model.

Key Points:
• SSD is efficient in terms of speed and accuracy.
• Ideal for applications requiring both speed and moderate precision.

SSD (Single Shot MultiBox Detector) with TensorFlow

SSD is an unified framework for object detection with a single network. It has been originally introduced in this research article. This repository contains a TensorFlow re-implementation of the original Caffe code.

CODE:

```
#Install Libraries.
!pip install tensorflow tensorflow-hub opencv-python matplotlib
```

This command installs essential libraries required for deep learning, computer vision, and data visualization tasks.

```python
#Import Libraries.
import tensorflow as tf
import tensorflow_hub as hub
import cv2
import numpy as np
import matplotlib.pyplot as plt

#Load the SSD model from TensorFlow Hub.
ssd_model = hub.load("https://tfhub.dev/tensorflow/ssd_mobilenet_v2/2")

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

    #Convert to RGB.
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    height, width = image.shape[:2]

    #Preprocess image for SSD model.
    input_tensor = tf.image.resize(image_rgb, (300, 300))
    input_tensor = tf.cast(input_tensor, dtype=tf.uint8)  #Convert to uint8.
    input_tensor = tf.expand_dims(input_tensor, axis=0)  #Add batch dimension.

    #Perform SSD detection.
    detections = ssd_model(input_tensor)

    #Extract results.
    detection_boxes = detections['detection_boxes'][0].numpy()
    detection_scores = detections['detection_scores'][0].numpy()
    detection_classes = detections['detection_classes'][0].numpy().astype(int)

    #Define confidence threshold.
    confidence_threshold = 0.5

    #Draw bounding boxes on the image.
    for i in range(len(detection_scores)):
        if detection_scores[i] >= confidence_threshold:
            box = detection_boxes[i]
            y_min, x_min, y_max, x_max = int(box[0] * height), int(box[1] * width), int(box[2] * height), int(box[3] * width)

            #Draw bounding box and label.
            cv2.rectangle(image_rgb, (x_min, y_min), (x_max, y_max), (0, 255, 0), 3)
            label = f"Class {detection_classes[i]}: {detection_scores[i]:.2f}"
            cv2.putText(image_rgb, label, (x_min, y_min - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

    return image_rgb
```

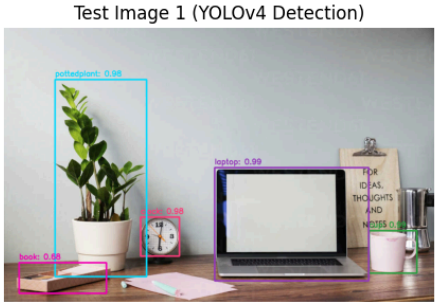
```
#Multiple images to test SSD and compare with YOLO.
image_paths = [
    '/content/drive/MyDrive/YOLOv4/office.jpg',
    '/content/drive/MyDrive/YOLOv4/crosswalk.JPG',
    '/content/drive/MyDrive/YOLOv4/animals.jpg',
    '/content/drive/MyDrive/YOLOv4/fruits.jpg'
]

#Visualization.
plt.figure(figsize=(14, 12))

for i, path in enumerate(image_paths, 1):
    #YOLO detection.
    yolo_detected_image = detect_objects(path)

    #SSD detection.
    ssd_detected_image = detect_objects_ssd(path)

    #Display YOLO result.
    plt.subplot(len(image_paths), 2, 2 * i - 1)
    plt.imshow(yolo_detected_image)
    plt.axis('off')
    plt.title(f"Test Image {i} (YOLOv4 Detection)")

    #Display SSD result.
    plt.subplot(len(image_paths), 2, 2 * i)
    plt.imshow(ssd_detected_image)
    plt.axis('off')
    plt.title(f"Test Image {i} (SSD Detection)")

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

This code performs object detection using two popular models: SSD (Single Shot Detector) and YOLO (You Only Look Once) on multiple images, then visualizes the results for comparison. The process begins by importing necessary libraries like TensorFlow, TensorFlow Hub, OpenCV, and Matplotlib. The SSD model is loaded from TensorFlow Hub, specifically the `ssd_mobilenet_v2` model, which is optimized for efficient object detection. The main function, `detect_objects_ssd`, takes an image path as input and processes the image for the SSD model. It reads the image using OpenCV, then converts it from BGR to RGB format and resizes it to 300x300 pixels (the required input size for the SSD model). The image is also cast to `uint8` format and reshaped to add a batch dimension, as the model expects batches of images. The SSD model performs detection and returns bounding boxes, scores, and class labels. A confidence threshold of 0.5 is set, meaning only detections with a score above 0.5 are considered. For these high-confidence detections, bounding boxes are drawn around the objects, and labels with detection confidence are added.

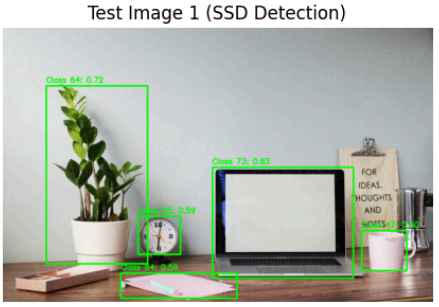
A list of image paths is defined, and the code iterates over each image, applying both YOLO and SSD detection. For visualization, Matplotlib is used to create a grid layout. For each image, the YOLO detection result is displayed on the left, and the SSD result is on the right, allowing for a side-by-side comparison. The images are shown in a subplot grid, with each pair of YOLO and SSD results placed together, helping to visually assess differences between the two models on various images, such as office scenes, crosswalks, animals, and fruits.

RESULT:

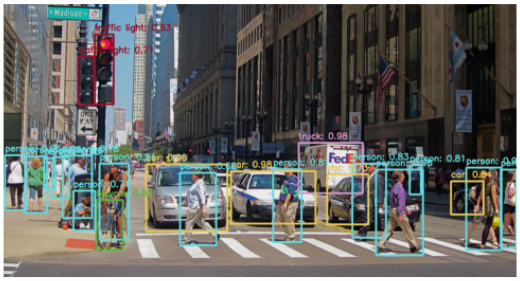
Test Image 1 (YOLOv4 Detection)



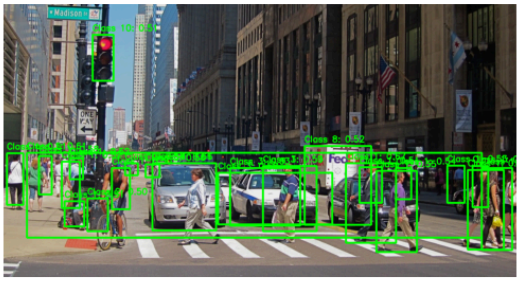
Test Image 1 (SSD Detection)



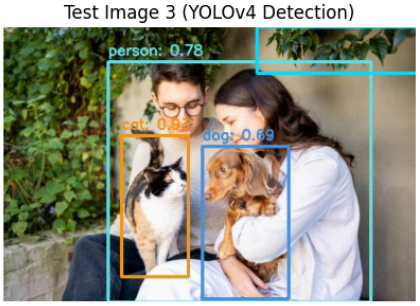
Test Image 2 (YOLOv4 Detection)



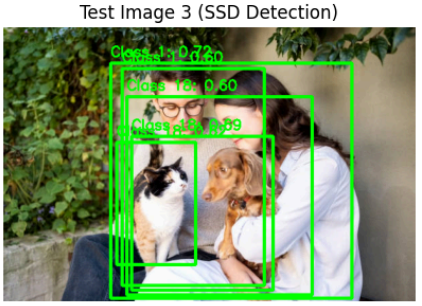
Test Image 2 (SSD Detection)



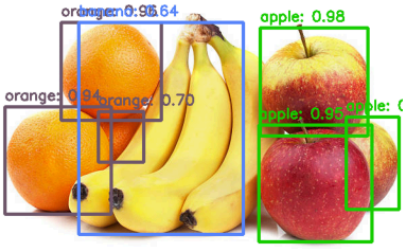
Test Image 3 (YOLOv4 Detection)



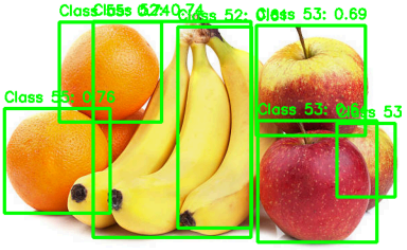
Test Image 3 (SSD Detection)



Test Image 4 (YOLOv4 Detection)



Test Image 4 (SSD Detection)



Exercise 4: Traditional vs. Deep Learning Object Detection Comparison

Task: Compare traditional object detection (e.g., HOG-SVM) with deep learning-based methods (YOLO, SSD):

• Implement HOG-SVM and either YOLO or SSD for the same dataset.
• Compare their performances in terms of accuracy and speed.
• Document the advantages and disadvantages of each method.

Key Points:

• Traditional methods may perform better in resource-constrained environments.

• Deep learning methods are generally more accurate but require more computational power.

Traditional vs. Deep Learning Object Detection Comparison

CODE:

```python
#HOG-SVM Implementation
#Import Libraries.
import cv2
import matplotlib.pyplot as plt
import time

#Initialize HOG descriptor with default people detector.
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

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

    #Convert image to grayscale.
    image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    #Start timing.
    start_time = time.time()

    #Detect people in the image using HOG-SVM.
    (rects, weights) = hog.detectMultiScale(image_gray, winStride=(4, 4), padding=(8, 8), scale=1.05)

    #End timing.
    end_time = time.time()
    hog_time = end_time - start_time

    #Draw bounding boxes.
    for (x, y, w, h) in rects:
        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

    print(f"HOG-SVM detection completed in {hog_time:.4f} seconds for {image_path}.")
    return image, hog_time
```

This code implements object detection using the Histogram of Oriented Gradients (HOG) with a Support Vector Machine (SVM) classifier, specifically designed to detect people in an image. It begins by importing essential libraries such as OpenCV, Matplotlib, and time-tracking utilities. The HOG descriptor, which focuses on edge directions to capture object shapes, is initialized with a pre-trained people detector, optimized for detecting human figures. The main function, `detect_objects_hog`, accepts an image file path as input. It reads the image using OpenCV, and if the image is unavailable at the specified path, it raises an error. The image is then converted to grayscale, which reduces the computational load and focuses on edge detection without considering color information. After this, a timer starts to

track the processing time for detection, giving insight into the efficiency of the HOG-SVM method.

The function then applies the HOG-SVM detector to the grayscale image. The `detectMultiScale` method scans the image, using parameters like `winStride`, which defines the step size in the sliding window, `padding`, which adds surrounding space around the window, and `scale`, which adjusts the size of the image at each scale step. The method returns `rects`, which are bounding boxes around detected objects, and `weights`, which represent confidence levels for each detection. Once detection is complete, the timer stops, and the elapsed time is printed, showing the detection speed. Finally, bounding boxes are drawn around each detected object in the original image to visualize the results, and the processed image and the detection time are returned. This implementation is an effective way to detect people in an image while providing timing information that can be useful for performance evaluation.

```python
#SSD Implementation
def detect_objects_ssd(image_path):
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError(f"Image not found at the specified path: {image_path}")

    #Convert image to grayscale.
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    height, width = image.shape[:2]

    #Resize image to 300x300, expected by SSD model, and keep it as uint8.
    input_tensor = tf.image.resize_with_pad(image_rgb, 300, 300)
    input_tensor = tf.cast(input_tensor, dtype=tf.uint8)
    input_tensor = tf.expand_dims(input_tensor, axis=0)

    #Start timing.
    start_time = time.time()

    #Run SSD model.
    detections = ssd_model(input_tensor)

    #End timing.
    end_time = time.time()
    ssd_time = end_time - start_time

    #Extract detection results.
    detection_boxes = detections['detection_boxes'][0].numpy()
    detection_scores = detections['detection_scores'][0].numpy()
    detection_classes = detections['detection_classes'][0].numpy().astype(int)

    #Set a confidence threshold.
    confidence_threshold = 0.5

    #Draw bounding boxes on the image.
    for i in range(len(detection_scores)):
        if detection_scores[i] >= confidence_threshold:
            box = detection_boxes[i]
            y_min, x_min, y_max, x_max = int(box[0] * height), int(box[1] * width), int(box[2] * height), int(box[3] * width)
            cv2.rectangle(image_rgb, (x_min, y_min), (x_max, y_max), (0, 255, 0), 3)
            label = f"Class {detection_classes[i]}: {detection_scores[i]:.2f}"
            cv2.putText(image_rgb, label, (x_min, y_min - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

    print(f"SSD detection completed in {ssd_time:.4f} seconds for {image_path}.")
    return image_rgb, ssd_time
```

This code implements object detection using the SSD (Single Shot Multibox Detector) model, optimized for quick and efficient identification of multiple objects within an

image. The function `detect_objects_ssd` begins by reading an input image from the specified file path using OpenCV. If the image is not found, an error message is raised. The image is then converted from BGR to RGB color format, as required by the SSD model, and the image's original dimensions (height and width) are saved for later use. Next, the image is resized to 300x300 pixels, the input size expected by the SSD model, while preserving the uint8 data format. This resized image is also reshaped to add a batch dimension, which prepares it for processing by the SSD model. To track the processing time, a timer starts just before the model performs detection on the input image.

Once the model completes the detection, the timer stops, and the time taken is recorded. The SSD model outputs several key pieces of information for each detected object, including bounding box coordinates, confidence scores, and class labels. These results are extracted for post-processing. A confidence threshold of 0.5 is applied, meaning only objects with a confidence score of at least 50% are considered valid detections. Bounding boxes are then drawn around each detected object on the original image using the extracted coordinates, scaling them back to match the original image dimensions. Labels are also added to indicate the class of the detected object and its confidence score. Finally, the function prints the detection time and returns the processed image with bounding boxes and the elapsed detection time, providing insight into SSD's performance on the given image.

```
#Multiple images to test HOG-SVM and SSD.
image_paths = [
    '/content/drive/MyDrive/YOLOv4/office.jpg',
    '/content/drive/MyDrive/YOLOv4/crosswalk.JPG',
    '/content/drive/MyDrive/YOLOv4/animals.jpg',
    '/content/drive/MyDrive/YOLOv4/fruits.jpg'
]

#Visualization.
plt.figure(figsize=(14, 12))

for i, path in enumerate(image_paths, 1):
    #HOG-SVM detection.
    hog_detected_image, hog_time = detect_objects_hog(path)

    #SSD detection.
    ssd_detected_image, ssd_time = detect_objects_ssd(path)

    #Display HOG-SVM result.
    plt.subplot(len(image_paths), 2, 2 * i - 1)
    plt.imshow(cv2.cvtColor(hog_detected_image, cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.title(f"HOG-SVM Detection - Time: {hog_time:.4f} sec")

    #Display SSD result.
    plt.subplot(len(image_paths), 2, 2 * i)
    plt.imshow(ssd_detected_image)
    plt.axis('off')
    plt.title(f"SSD Detection - Time: {ssd_time:.4f} sec")

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

This code segment performs object detection on multiple images using two methods: HOG-SVM and SSD, then visualizes and compares their results side-by-side. It first defines a list of image paths, each pointing to a different test image, such as scenes with people, animals, and objects. For each image, the code applies both HOG-SVM and SSD detection. The loop iterates over each image in the list. For each iteration, the `detect_objects_hog` function is called to detect objects using the HOG-SVM approach, which returns the processed image and the detection time. Similarly, `detect_objects_ssd` is called to detect objects with the SSD model, also returning an image with detections and the time taken.

For visualization, the code uses Matplotlib to arrange the results in a grid layout. For each image, the HOG-SVM detection result is displayed on the left, showing bounding boxes and detection time in seconds. The SSD detection result for the same image is displayed on the right, also with bounding boxes and the time taken for detection. The images are presented in pairs, with each row showing the two methods' results for a single image, making it easy to visually assess differences in detection accuracy and speed between HOG-SVM and SSD.

Finally, `plt.tight_layout()` adjusts the spacing between plots to improve readability, and `plt.show()` renders the entire visualization, providing a clear comparison of HOG-SVM and SSD detection performance on a variety of images.

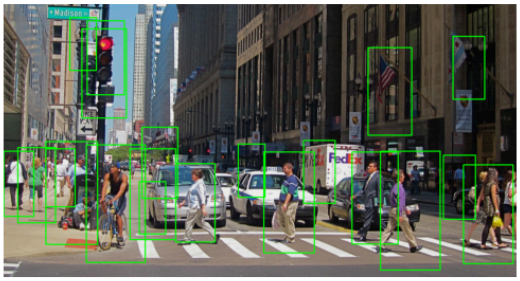
RESULT:

```
HOG-SVM detection completed in 2.2975 seconds for /content/drive/MyDrive/YOLOv4/office.jpg.
SSD detection completed in 0.0915 seconds for /content/drive/MyDrive/YOLOv4/office.jpg.
HOG-SVM detection completed in 1.9794 seconds for /content/drive/MyDrive/YOLOv4/crosswalk.JPG.
SSD detection completed in 0.0891 seconds for /content/drive/MyDrive/YOLOv4/crosswalk.JPG.
HOG-SVM detection completed in 0.4624 seconds for /content/drive/MyDrive/YOLOv4/animals.jpg.
SSD detection completed in 0.0924 seconds for /content/drive/MyDrive/YOLOv4/animals.jpg.
HOG-SVM detection completed in 0.4578 seconds for /content/drive/MyDrive/YOLOv4/fruits.jpg.
SSD detection completed in 0.1020 seconds for /content/drive/MyDrive/YOLOv4/fruits.jpg.
```

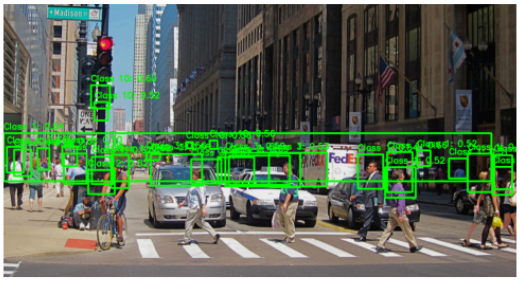
HOG-SVM Detection - Time: 2.2975 sec

SSD Detection - Time: 0.0915 sec

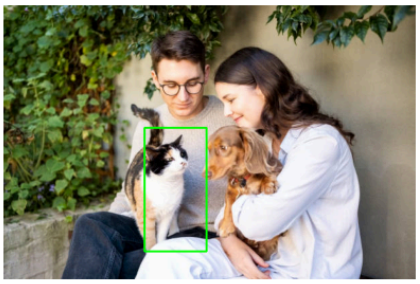
HOG-SVM Detection - Time: 1.9794 sec



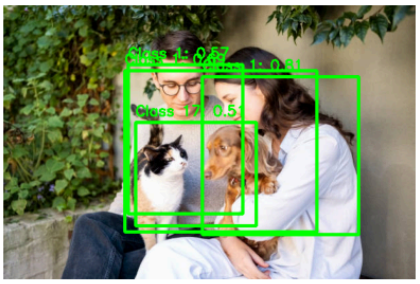
SSD Detection - Time: 0.0891 sec



HOG-SVM Detection - Time: 0.4624 sec



SSD Detection - Time: 0.0924 sec



HOG-SVM Detection - Time: 0.4578 sec

SSD Detection - Time: 0.1020 sec