Exercise 4 - Object Detection and Recognition

In this exercise, there are 4 tasks covered:

- HOG (Histogram of Oriented Gradients) Object Detection
- YOLO (You Only Look Once) Object Detection
- SSD (Single Shot MultiBox Detector) with TensorFlow
- Traditional vs. Deep Learning Object Detection Comparison

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

Code:

```
import cv2
from skimage.feature import hog
import matplotlib.pyplot as plt
# Load an image
image = cv2.imread('car.jpg')

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

   # Apply HOG descriptor
features, hog_image = hog(
       gray_image,
       orientations=9,
       pixels_per_cell=(8, 8),
       cells_per_block=(2, 2),
       visualize=True
   )

   # Normalize the HOG image to improve brightness
hog_image = (hog_image - hog_image.min()) / (hog_image.max() -
hog_image.min())

   # Display the HOG image
plt.figure(figsize=(8, 8))
```

```
plt.axis('off')
plt.imshow(hog_image, cmap='gray')
plt.show()
```

Output:

Explanation:

The code demonstrates applying the **Histogram of Oriented Gradients (HOG)** feature descriptor to an image using OpenCV and scikit-image libraries. First, the code imports the necessary libraries: OpenCV for image processing, skimage for HOG features, and matplotlib for visualization.

An image is loaded using cv2.imread() and converted to grayscale with cv2.cvtColor(). This conversion is essential because HOG operates on single-channel images, simplifying the analysis by reducing color complexity. The HOG descriptor is then applied using the hog() function, which calculates the gradient orientations and magnitudes within the image. The parameters set orientations to 9, pixels per cell to (8, 8), and cells per block to (2, 2), influencing the granularity of the extracted features.

The resulting HOG image is normalized to enhance its brightness, making it more visually informative. Finally, the code uses matplotlib to display the HOG image in a clear, gray-scale format, providing an intuitive representation of the features detected.

The advantages of using the HOG feature descriptor include its effectiveness in object detection and recognition, particularly in detecting shapes and edges. It provides robust features that are less sensitive to variations in lighting and color, making it suitable for a variety of computer vision tasks. By employing HOG, the analysis can

focus on the structure of objects rather than their color, facilitating improved detection in complex scenes.

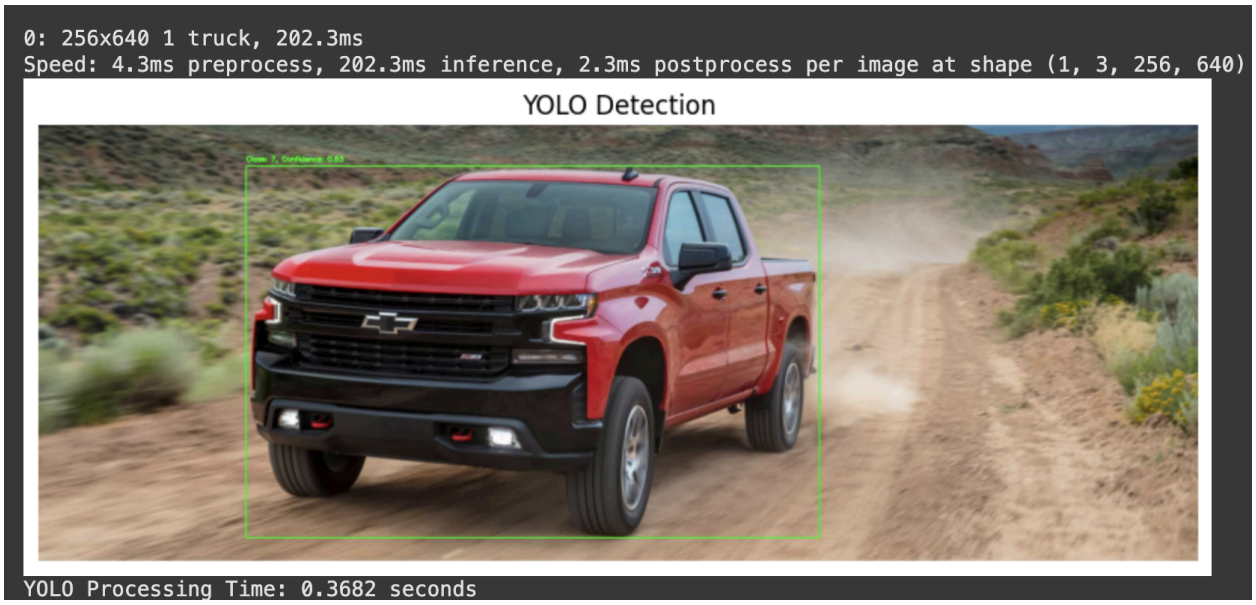
Task 2: YOLO (You Only Look Once) Object Detection

Code:

```
import cv2
import numpy as np
from ultralytics import YOLO  # Import the YOLO class from the ultralytics package
from google.colab.patches import cv2_imshow
# Load the pre-trained YOLO model
model = YOLO('yolov8n.pt')  # Use a pre-trained YOLOv8 model

# Load an image
image = cv2.imread('car.jpg')

# Check if the image was loaded successfully
if image is None:
    raise FileNotFoundError("Image not found. Please check the image path.")

# Perform inference
results = model(image)

# Process results
for result in results:
   boxes = result.boxes  # Get the boxes from the detection results
   for box in boxes:
        x1, y1, x2, y2 = box.xyxy[0].numpy()  # Get the bounding box coordinates
       conf = box.conf[0].item()  # Get the confidence score
       class_id = int(box.cls[0].item())  # Get the class ID

       if conf > 0.5:  # Filter out low confidence detections
           label = f"Class: {class_id}, Confidence: {conf:.2f}"
           # Draw bounding box and label
```

```
            cv2.rectangle(image, (int(x1), int(y1)), (int(x2), int(y2)),
(0, 255, 0), 2)
                    cv2.putText(image, label, (int(x1), int(y1) - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

# Display the image with detections using cv2_imshow
#cv2_imshow(image)  # Use cv2_imshow instead of cv2.imshow()

# Additionally, using matplotlib to show the image
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))   # Convert BGR to RGB
for proper color display
plt.axis('off')
plt.title('YOLO Detection')
plt.show()
```

Output:



Explanation:

This showcases how to implement object detection using the **YOLO (You Only Look Once)** model with the Ultralytics package and OpenCV. First, the necessary libraries, including OpenCV, NumPy, and the YOLO model, are imported. The YOLO model is initialized with a pre-trained weights file (yolov8n.pt), which allows it to recognize various objects without the need for retraining.

An image is then loaded using cv2.imread(), and a check is performed to ensure the image has been loaded successfully. If the image cannot be found, a FileNotFoundError is raised, which helps to catch issues early in the process. After successfully loading the image, inference is performed using the YOLO model, generating detection results.

The results are processed to extract bounding boxes, confidence scores, and class IDs for detected objects. A loop iterates through the detected boxes, and for each detection, if the confidence score exceeds 0.5, a label is created indicating the class and confidence. The bounding box and label are then drawn on the image using OpenCV functions, enhancing the visual feedback of the detection process.

The code includes a commented-out line for displaying the image using cv2_imshow(), which is useful in Google Colab, where traditional OpenCV display functions may not work as expected. Instead, the image is ultimately displayed using Matplotlib. This conversion from BGR to RGB color format ensures that the colors appear correctly in the output image.

The advantages of using the YOLO model for object detection include its speed and accuracy. YOLO processes images in a single pass, allowing for real-time detection, which is beneficial for applications like surveillance, self-driving cars, and robotics. The model can identify multiple objects within a single frame, making it highly efficient for complex scenes. Overall, the code exemplifies a practical approach to implementing object detection with advanced neural networks, facilitating easy deployment in various applications.

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

Code:

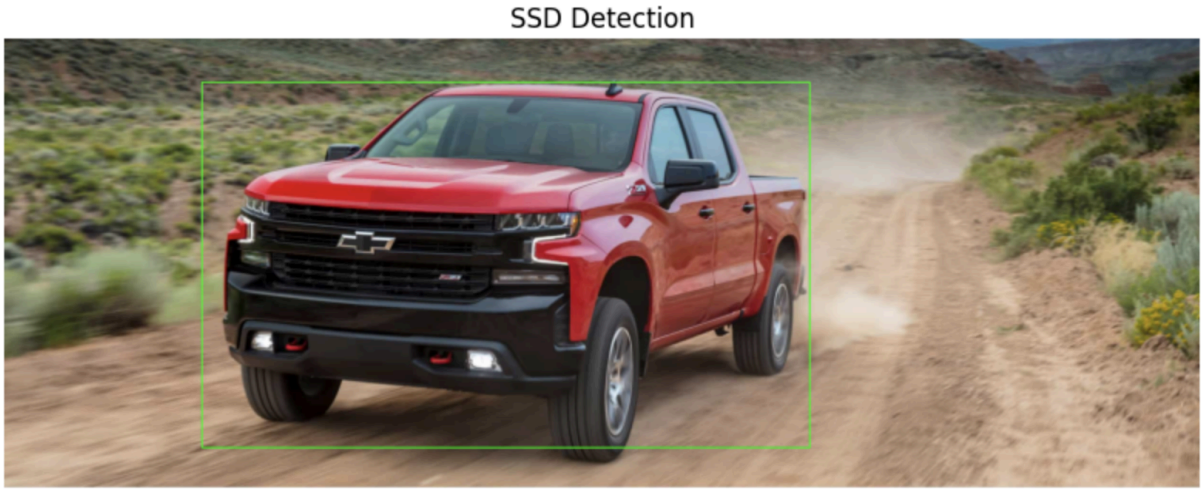
```
import tensorflow as tf
import cv2
import numpy as np
```

```
import matplotlib.pyplot as plt

# Load pre-trained SSD model

model_path = '/content/openimages_model/openimages_v4_ssd_mobilenet_v2_1'
# Path to the model directory

model = tf.saved_model.load(model_path)


# Print available signatures

print("Available signatures:")

for signature_key in model.signatures.keys():

   print(signature_key)


# Access the appropriate serving function

infer = model.signatures['default']  # Using 'default' as the key


# Load image

image_path = 'car.jpg'  # Make sure this image is uploaded to Colab

image_np = cv2.imread(image_path)

image_np = cv2.cvtColor(image_np, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB


# Convert the image to a tensor and add batch dimension

input_tensor = tf.convert_to_tensor(image_np, dtype=tf.float32)  # Convert
to float32

input_tensor = input_tensor[tf.newaxis, ...]


# Normalize the pixel values to [0, 1] range

input_tensor /= 255.0


# Run the model

detections = infer(input_tensor)
```

```
# Print the output to check the structure
print("Output from model:")
for key, value in detections.items():
   print(f"{key}: {value.shape}")

# Get the number of detections
num_detections = int(detections['detection_scores'].shape[0])     # Get number of detections

# Visualize the bounding boxes
for i in range(num_detections):
   if detections['detection_scores'][i] > 0.5:  # Confidence threshold
       # Get bounding box coordinates
       ymin, xmin, ymax, xmax = detections['detection_boxes'][i].numpy()
          (left, right, top, bottom) = (xmin * image_np.shape[1], xmax * image_np.shape[1],
                                        ymin * image_np.shape[0], ymax * image_np.shape[0])

       # Draw bounding box
            cv2.rectangle(image_np, (int(left), int(top)), (int(right), int(bottom)), (0, 255, 0), 2)

# Display the image with bounding boxes using matplotlib
plt.figure(figsize=(10, 10))
plt.imshow(image_np)
plt.axis('off')
plt.title('SSD Detection')
plt.show()
```

Output:

```
Available signatures:
default
Output from model:
detection_class_labels: (100,)
detection_boxes: (100, 4)
detection_scores: (100,)
detection_class_names: (100,)
detection_class_entities: (100,)
```



Explanation:

This demonstrates how to perform object detection using a pre-trained **SSD (Single Shot MultiBox Detector) model in TensorFlow**. It begins by importing necessary libraries and loading the SSD model from a specified path. The model's available signatures are printed to understand how to call it for inference.

An image is loaded and converted from BGR to RGB format. The image is transformed into a tensor, with a batch dimension added and pixel values normalized to a range of [0, 1]. The model is then invoked to generate detections, with the output structure printed for verification.

The code iterates through the detections, drawing bounding boxes around objects with a confidence score above 0.5. Finally, the image with the drawn bounding boxes is displayed using Matplotlib.

Using the SSD model offers advantages such as fast, accurate object detection in a single pass, making it suitable for real-time applications in areas like surveillance and autonomous driving. This code exemplifies an efficient approach to implementing deep learning-based object detection.

Task 4: Traditional vs. Deep Learning Object Detection Comparison

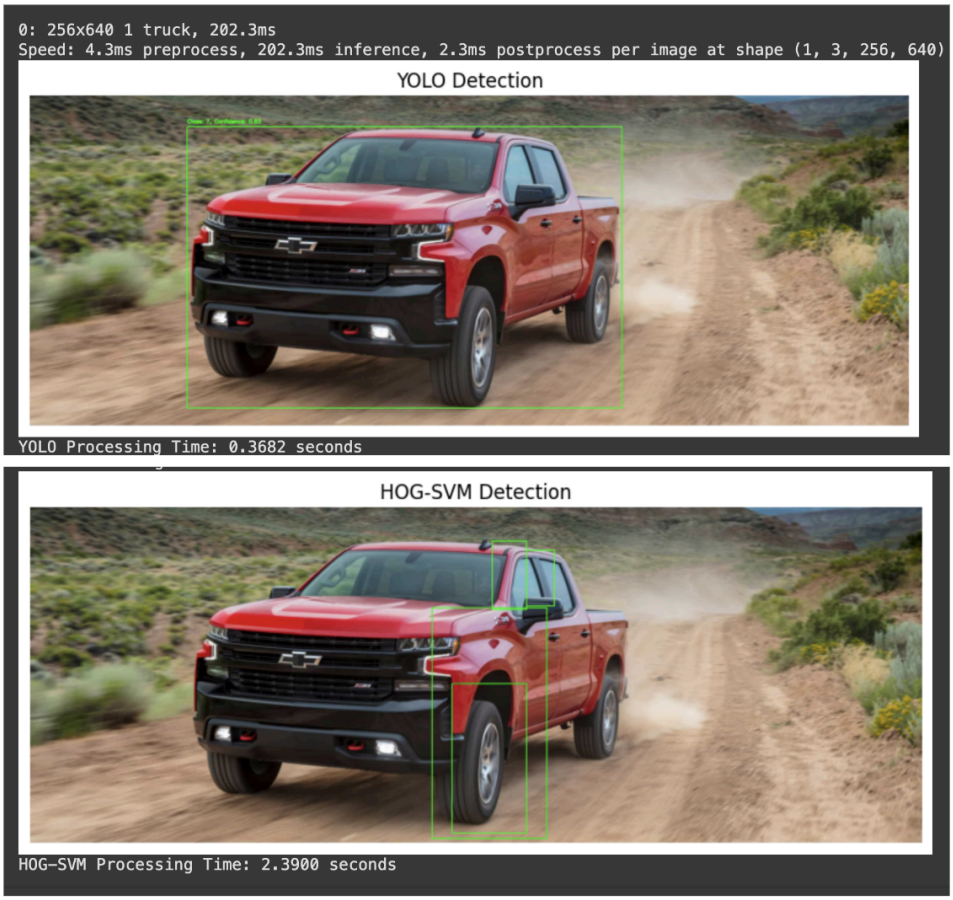
Code:

```
def display_image(image, title):
   plt.figure(figsize=(10, 10))
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))   # Convert BGR to
RGB for proper color display
   plt.axis('off')
   plt.title(title)
   plt.show()

# Load the pre-trained YOLO model
model = YOLO('yolov8n.pt')  # Use a pre-trained YOLOv8 model

# Load an image for detection
image_path = 'car.jpg'
image_yolo = cv2.imread(image_path)

# Check if the image was loaded successfully
if image_yolo is None:
     raise FileNotFoundError("Image not found. Please check the image
path.")

# --------- YOLO Detection ---------
start_time_yolo = time.time()  # Start time for YOLO speed measurement
results = model(image_yolo)  # Perform inference
end_time_yolo = time.time()  # End time for YOLO speed measurement

# Process YOLO results
for result in results:
   boxes = result.boxes  # Get the boxes from the detection results
   for box in boxes:
```

```
            x1, y1, x2, y2 = box.xyxy[0].numpy()  # Get the bounding box
coordinates
        conf = box.conf[0].item()  # Get the confidence score
        class_id = int(box.cls[0].item())  # Get the class ID

        if conf > 0.5:  # Filter out low confidence detections
            label = f"Class: {class_id}, Confidence: {conf:.2f}"
            # Draw bounding box and label
                  cv2.rectangle(image_yolo, (int(x1), int(y1)), (int(x2),
int(y2)), (0, 255, 0), 2)
                cv2.putText(image_yolo, label, (int(x1), int(y1) - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

# Display YOLO detection results
display_image(image_yolo, 'YOLO Detection')
print(f"YOLO Processing Time: {end_time_yolo - start_time_yolo:.4f}
seconds")


# --------- HOG-SVM Detection ---------
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

# Load the same image for HOG-SVM detection
image_hog = cv2.imread(image_path)

# Check if the image was loaded successfully
if image_hog is None:
    raise FileNotFoundError("Image not found. Please check the image
path.")

# Perform HOG detection
start_time_hog = time.time()  # Start time for HOG speed measurement
boxes, weights = hog.detectMultiScale(image_hog, winStride=(8, 8),
padding=(8, 8), scale=1.05)
end_time_hog = time.time()  # End time for HOG speed measurement

# Draw bounding boxes on the HOG-SVM image
for (x, y, w, h) in boxes:
   cv2.rectangle(image_hog, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

```
# Display HOG-SVM detection results
display_image(image_hog, 'HOG-SVM Detection')
print(f"HOG-SVM Processing Time: {end_time_hog - start_time_hog:.4f}
seconds")
```

Output:



Explanation:

This compares two object detection methods: **YOLO (You Only Look Once) and HOG (Histogram of Oriented Gradients) with SVM (Support Vector Machine)**. It evaluates their performance and visualizes the detection results.

First, a helper function display_image is defined to facilitate the display of images using Matplotlib. This function converts images from BGR to RGB format for proper color representation, turns off the axis, and sets the title.

The code then loads a pre-trained YOLOv8 model and an image for detection. It checks if the image is loaded successfully, raising a FileNotFoundError if not. The YOLO detection process is initiated, with timing measurements for performance analysis. The results are processed to extract bounding box coordinates, confidence scores, and class IDs. Detections with confidence scores above 0.5 are drawn on the image, along with corresponding labels.

After visualizing the YOLO detection results, the code measures the processing time and prints it. The same image is loaded again for HOG-SVM detection, and the HOG descriptor is set up to use the default people detector. The HOG detection is performed, with timing also measured for comparison. Detected bounding boxes are drawn on the HOG-SVM image.

Finally, the HOG-SVM detection results are displayed, and the processing time is printed.

This comparison shows the differences between traditional and deep learning-based object detection techniques. YOLO is generally faster and can detect multiple objects in a single pass, while HOG-SVM is a more traditional method that may have limitations in terms of speed and detection accuracy. This experiment provides valuable insights into the strengths and weaknesses of each approach for object detection tasks.

- END OF DOCUMENTATION -