Lab Sheet 1

Lane and Car Detection with OpenCV

Date: 08/08/2025

Aim

Applications of OpenCV:

- Lane and Car Detection

Introduction

OpenCV is a powerful open-source library used for a wide range of applications in computer vision, machine learning, and image processing. Its primary aim is to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in products.

The significance of OpenCV is in its ability to process images and videos to identify objects, faces, text, and even human actions in real time. These applications range from facial recognition systems used for security and photo tagging, and motion tracking for surveillance, to advanced driver-assistance systems (ADAS) in vehicles that detect lanes and obstacles. OpenCV enables people to create advanced applications capable of experiencing the visual world and interpreting it by offering a rich set of tools that are computationally efficient and useful.

Code Snippets and Explanation

1. Perspective Transformation (Bird's-Eye View)

To accurately calculate the curvature of the lane lines, first view from the top-down perspective, which simplifies the geometry of the problem. The 'perspectiveWarp' function achieves this by applying a perspective transform. It creates a "Bird's-Eye View" of the road ahead. This step allows us to treat the curved lane lines as polynomials in a 2D space.

```
def perspectiveWarp(inpImage):
    img_size = (inpImage.shape[1], inpImage.shape[0])

    src = np.float32([[550, 475], [256, 700], [1200, 700], [740, 475]])
    dst = np.float32([[256, 0], [256, 720], [1200, 720], [1200, 0]])

    matrix = cv2.getPerspectiveTransform(src, dst)
    minv = cv2.getPerspectiveTransform(dst, src)
    birdseye = cv2.warpPerspective(inpImage, matrix, img_size)

    return birdseye, None, None, minv, src, dst
```

Listing 1: Perspective Warping to Bird's-Eye View

2. Lane Fitting with Sliding Windows

After the image is transformed and pre-processed into a binary format, the 'slide_window_search' function identifies the pixels belonging to each lane. It uses a sliding window technique to move up the frame, capturing all the white pixels that belong to the lane lines. Finally, the collected points for each lane are used to mathematically represent the curvature of the lane.

```
def slide_window_search(binary_warped, histogram):
    # ... (logic to find pixel coordinates) ...

    # Collect points (leftx, lefty, rightx, righty)
    if lefty.size == 0 or righty.size == 0:
        return np.array([]), np.array([]), np.array([]), np.array([]), np.
   array([])

    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)

    ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0])
    left_fitx = left_fit[0] * ploty**2 + left_fit[1] * ploty + left_fit[2]
    right_fitx = right_fit[0] * ploty**2 + right_fit[1] * ploty + right_fit
   [2]

    return ploty, left_fit, right_fit, left_fitx, right_fitx
```

Listing 2: Sliding Window Search and Curve Fitting

3. Visualization and Final Overlay

The 'draw_lane_lines' function is for visualizing the final result. It takes the fitted curves and creates a polygon representing the drivable lane area. This polygon is filled with a semi-transparent yellow color and given a red border for clear visibility. This colored overlay, which exists in the bird's-eye perspective, is then transformed back to the original camera's perspective using an inverse perspective matrix.

```
def draw_lane_lines(original_image, warped_image, Minv, left_fitx,
   right_fitx, ploty):
    if left_fitx.size == 0 or right_fitx.size == 0:
        return original_image
    color_warp = np.zeros_like(original_image)

    # Create polygon from lane points
    pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
    pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx,
   ploty])))])
    pts = np.hstack((pts_left, pts_right))

    # Fill lane area with yellow and draw red borders
    cv2.fillPoly(color_warp, np.int_([pts]), (0, 255, 255))
    cv2.polylines(color_warp, np.int_([pts]), isClosed=True, color=(0, 0,
   255), thickness=5)

    # Undo warp and overlay on original image
```

```
    new_warp = cv2.warpPerspective(color_warp, Minv, (original_image.shape
[1], original_image.shape[0]))
    result = cv2.addWeighted(original_image, 1, new_warp, 0.3, 0)

    return result
```

Listing 3: Drawing Detected Lane with Border

Output

The system processes video frames and produces a real-time output where detected lanes are highlighted with a semi-transparent yellow overlay and red borders, while other vehicles are enclosed in green bounding boxes with their class labels.

Figure 1: Actual Video

Figure 2: Detection model output

Conclusion and Inference

This project demonstrates the integration of common computer vision techniques and modern deep learning models for a basic detection application. The use of OpenCV for image pre-processing, perspective transformation, and polynomial fitting proved effective for robust lane detection given good lighting conditions. The sliding window approach is a reliable technique for tracking lane curvature, though it can have failures in frames with low visibility or faded markings. The YOLOv8 model provided fast and accurate real-time detection of vehicles alongside the lane detection. Challenges remain in handling adverse weather conditions, strong shadows, and sharp curves, which can be addressed in future work.

Signature of Staff: