Computer Vision

Project Report:

OBJECT DETECTION AND DEPTH ESTIMATION

Project Author: Nirbhay Borikar
Date:15/12/2024

Contents

Chapter 1

Introduction

In this project report, we have explored algorithms for object detection and depth estimation. We utilized a subset of the KITTI dataset comprising 20 stereo images, along with the following associated files: the Labels file, which includes the ground truth bounding box coordinates, object types (e.g., Car, Pedestrian), and the ground truth distances from the camera to the detected objects; and the Calibration file, which provides the intrinsic matrix required for depth estimation.

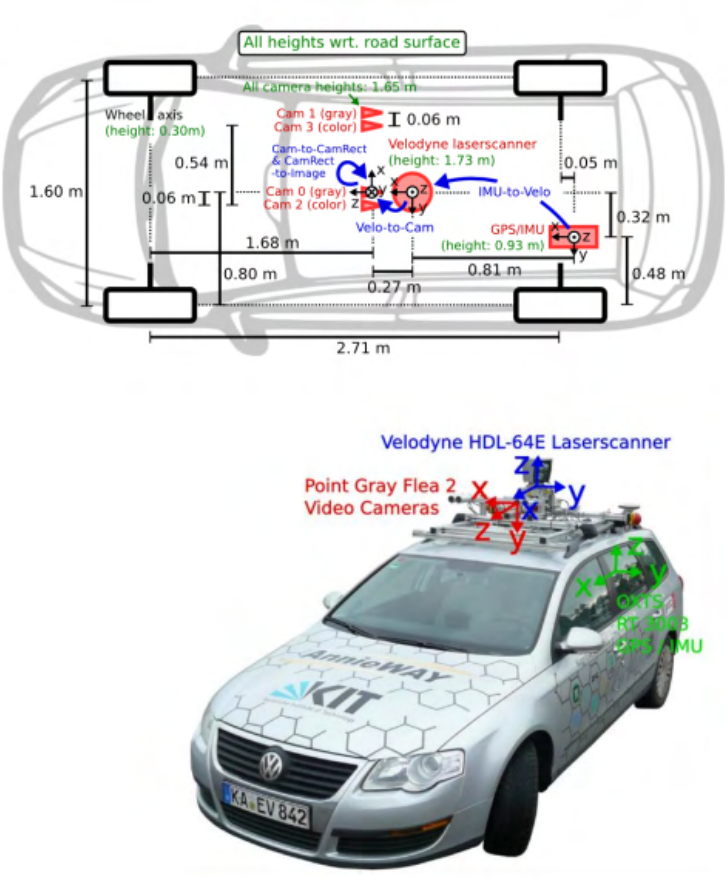


Figure 1: Camera setup in the vehicle (top) and the vehicle used in the KITTI dataset[1]

A pretrained deep neural network, YOLO 11m (You Only Look Once), was employed to detect cars and estimate the depth from the camera to the detected cars. This report outlines the methodologies adopted, key findings and observations, challenges encountered, outcomes achieved, and the future scope of the project.

Chapter 2

Algorithm Implemented

The following algorithm was implemented in this task to achieve the desired results.

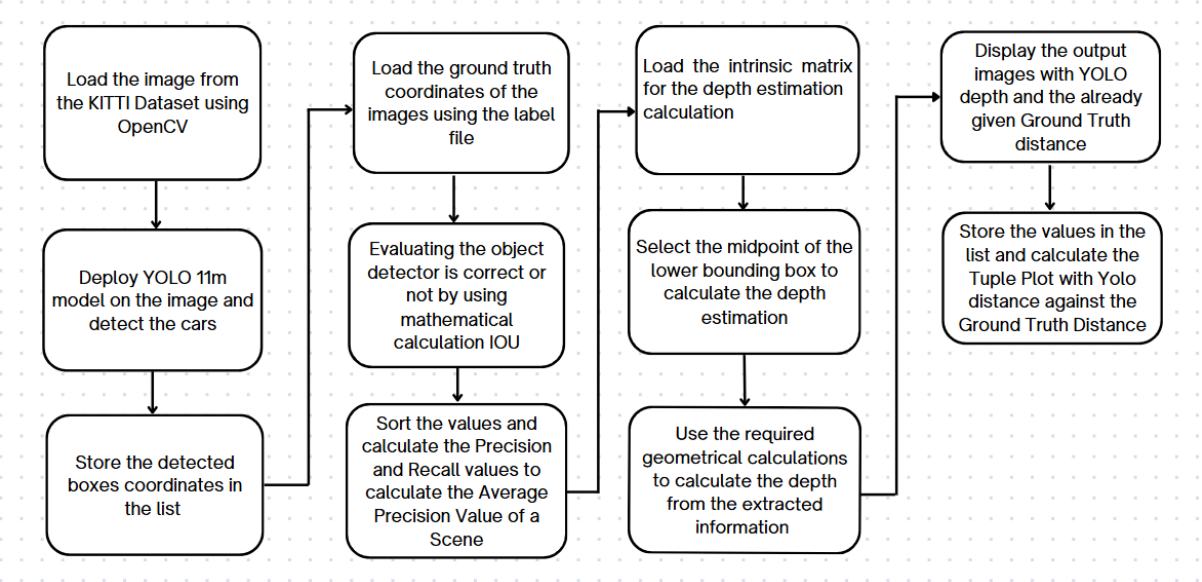


Figure 2: Steps followed in the algorithm

2.1 The KITTI Dataset

As mentioned in the Introduction, the KITTI dataset used in this project is a subset, comprising 20 stereo images, corresponding ground truth values, including coordinates, and the intrinsic matrices for depth estimation.

Figure 3: An instance of the KITTI Dataset image[2]

The lables file contains the type of the object, its ground truth coordinates(x1, y1, x2, y2) and the distance from the camera to the car in order.

```
Car 922.68 147.43 1241.0 297.73 9.392174559724607
Car 0.0 189.66 206.25 374.0 4.105116110713791
Car 111.3 184.93 369.81 334.65 8.901347913664482
```

Figure 4: A text file containing the values of the image used in page 2 [2]

Since the original KITTI dataset does not provide fixed distances for the objects, these distances must be calculated as ground truth values. We utilized the four corners $c1, c2, c3, c4$ and the four midpoints $m1, m2, m3, m4$, selecting the minimum of these eight values as the ground truth distance. It is depicted below.

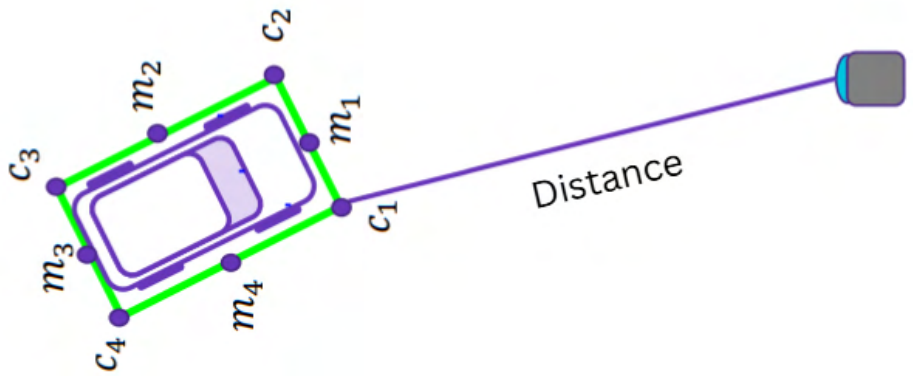


Figure 5: Method used to calculate the ground distance[1]

The KITTI data set also contains the calibration file which has the intrinsic camera matrices which is used for depth estimation.

```
7.215377197265625000e+02 0.000000000000000000e+00 6.095593261718750000e+02
0.000000000000000000e+00 7.215377197265625000e+02 1.728540039062500000e+02
0.000000000000000000e+00 0.000000000000000000e+00 1.000000000000000000e+00
```

Figure 6: A text file containing the intrinsic camera matrices of the image in page 2 [2]

The KITTI Dataset also have access of how the camera is setup in tpo of the vehicle.

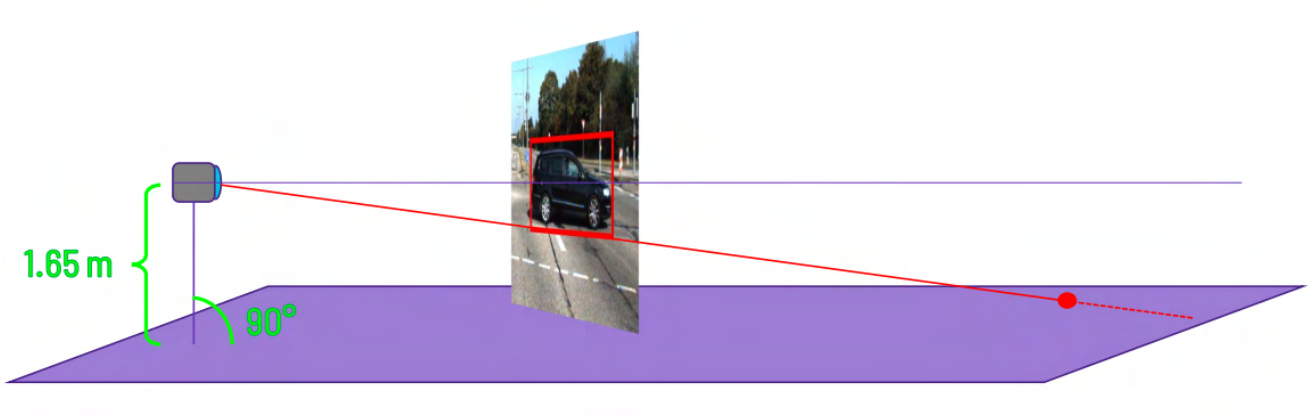


Figure 7: The camera is positioned in a height of 1.65 m from the driving plane[1]

2.2 Choosing the YOLO 11m model

The YOLO11 is the latest iteration of Ultralytics' YOLO series of real-time object detectors, redefining what is possible with industry-leading accuracy, speed, and efficiency. Building on the significant advances of previous versions of YOLO, YOLO11 introduces significant improvements in architecture and training methods, making it a versatile choice for a wide range of computer vision tasks.

Performance

Detection (COCO) | Segmentation (COCO) | Classification (ImageNet) | Pose (COCO) | OBB (DOTAv1)

See Detection Docs for usage examples with these models trained on COCO, which include 80 pre-trained classes.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

Figure 8: Performance metrics of the YOLO 11m model[3]

For our project, we selected the YOLO11m model due to its shorter processing time compared to the two models mentioned below and its higher mean average precision (mAP) compared to the two models mentioned above. So we selected the ideal one.

2.3 Detecting the cars and drawing Bounding boxes

Using the YOLO 11m model, the function detects the object "car" and retrieves its coordinates (x_1, y_1, x_2, y_2), storing them in a list. These coordinates are then used to draw bounding boxes around the detected objects, along with their confidence values. The confidence values are subsequently utilized to calculate the Precision and Recall metrics.

```
vehicle=cv2.imread('/home/nirbhayborikar/Documents/RWU/CV/project/KITTI_Selection/images/006042.png')
model = YOLO("yolo11m.pt")  # Load the YOLO model with the specified weight file
className = ["person", "bicycle", "car", "motorbike", "aeroplane"] # Define the list of class names corresponding to the model's output
results = model(vehicle, stream=True)# Perform object detection on the input image (vehicle) and stream the results
for r in results:# Iterate through the results from the model
    boxes = r.boxes  # Extract bounding box information for detected objects
    for box in boxes:  # Loop through each bounding box
        cls = int(box.cls[0])  # Convert the detected class ID (float) to an integer
        currentClass = className[cls]  # Get the class name using the class ID
        if currentClass == "car":# Check if the detected object is a car
            x1, y1, x2, y2 = box.xyxy[0]  # Extract bounding box coordinates (top-left and bottom-right corners)
            x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)  # Convert coordinates to integers
            w, h = x2 - x1, y2 - y1  # Calculate width and height of the bounding box
            print('top_left_coordinate', x1, y1, 'bottom_right_coordinate', x2, y2)# Print the coordinates of the bounding box

            cv2.rectangle(vehicle, (x1, y1), (x2, y2), (0, 0, 255), 3)# Draw a red rectangle around the detected car in the image
```

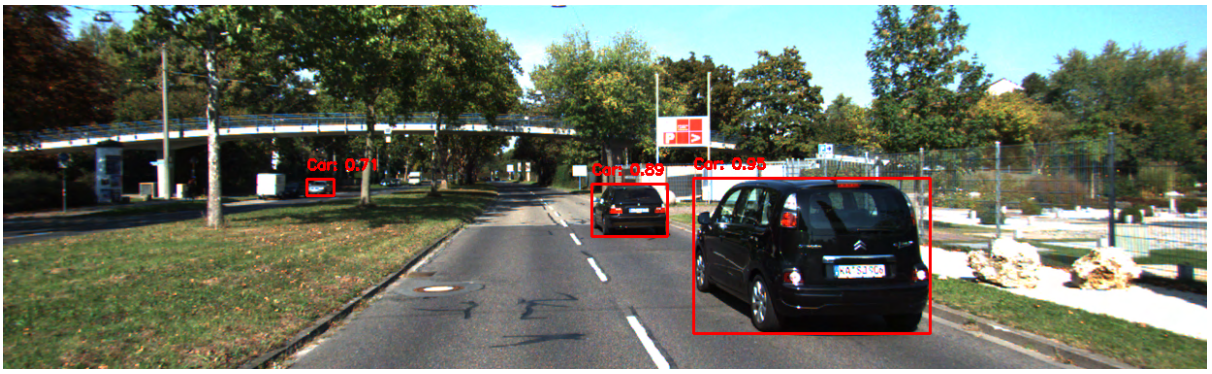


Figure 9: Bounding box detection for one instance

2.4 Loading the Ground truth bounding boxes

The labels file, which contains the ground truth values, is then loaded as it already includes the coordinates required to draw the ground truth bounding boxes.

```
# Define the path to the ground truth file
ground_truth_file = "/home/nirbhayborikar/Documents/RWU/CV/project/KITTI_Selection/labels/006374.txt"
ground_truth_boxes = [] # Initialize an empty list to store ground truth bounding boxes
try:                                    # Attempt to read the ground truth file
    with open(ground_truth_file, 'r') as file:  # Open the ground truth file for reading
        for line in file:  # Read each line in the file
            parts = line.strip().split()  # Split the line into parts (class and coordinates)
            x3, y3, x4, y4 = map(float, parts[1:5])  # Extract the bounding box coordinates and convert to float
            ground_truth_boxes.append((int(x3), int(y3), int(x4), int(y4)))  # Convert coordinates to integers and store as tuples
except FileNotFoundError:
    print("Error: Ground truth file not found.")  # Handle the case where the file is missing
    exit()  # Exit the program if the file is not found
for gt_box in ground_truth_boxes:   # Loop through the list of ground truth boxes
    x1, y1, x2, y2 = gt_box  # Unpack the bounding box coordinates
    cv2.rectangle(vehicle, (x1, y1), (x2, y2), (0, 255, 0), 2)  # Draw a green rectangle around the ground truth bounding box
```

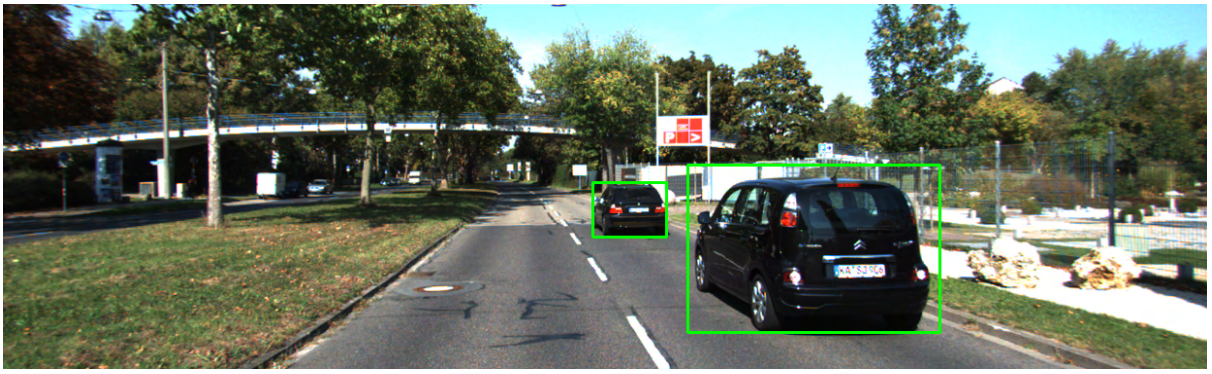


Figure 10: Ground truth Detection for one instance

2.5 Evaluation of the Object Detection Algorithm

To enhance the accuracy of object detection, any detected bounding box without a corresponding ground truth bounding box is removed from the final output. This ensures that only detected bounding boxes with matching ground truth values are displayed. To achieve this, a metric known as Intersection over Union (IoU) is implemented to evaluate the performance of the object detector. IoU measures how accurately the position of an object is predicted by comparing the overlap between the detected bounding box and the ground truth bounding box. The IoU calculation identifies the closest coordinates between detected and ground truth bounding boxes, ensuring precise alignment. A pictorial representation is provided below for better understanding.

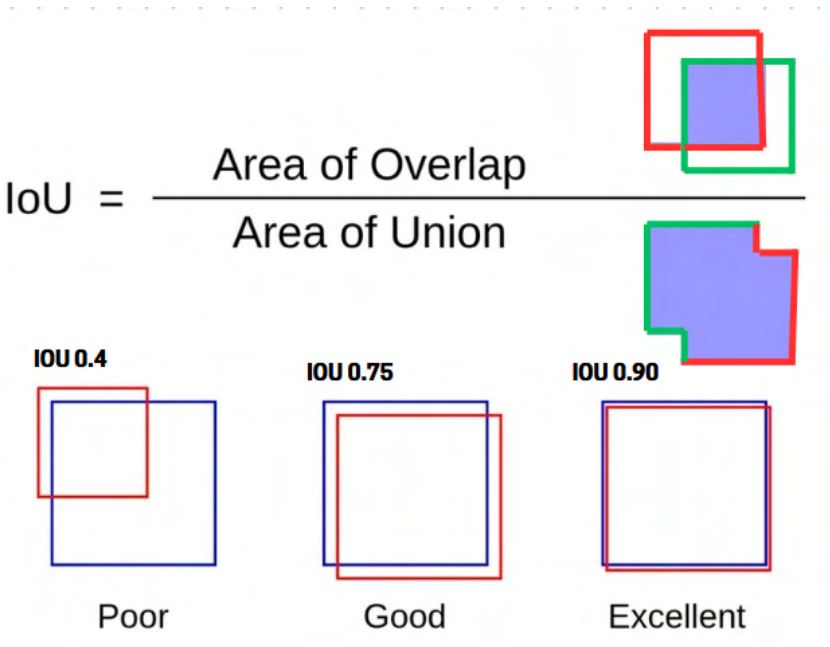


Figure 11: Evaluation of the Object Detection Algorithm[4]

We experimented with various values of λ, including 0.1, 0.3, and 0.5, to filter out YOLO-detected bounding boxes that closely matched the ground truth coordinates. For each YOLO bounding box, we calculated the Intersection over Union (IoU) with all the ground truth boxes and identified the maximum IoU value to determine the best match. By setting a threshold of IoU > 0.15, we successfully filtered out irrelevant bounding boxes. Bounding boxes with IoU $<= 0.15$ were excluded as they did not sufficiently align with the corresponding ground truth boxes, ensuring more accurate detection.

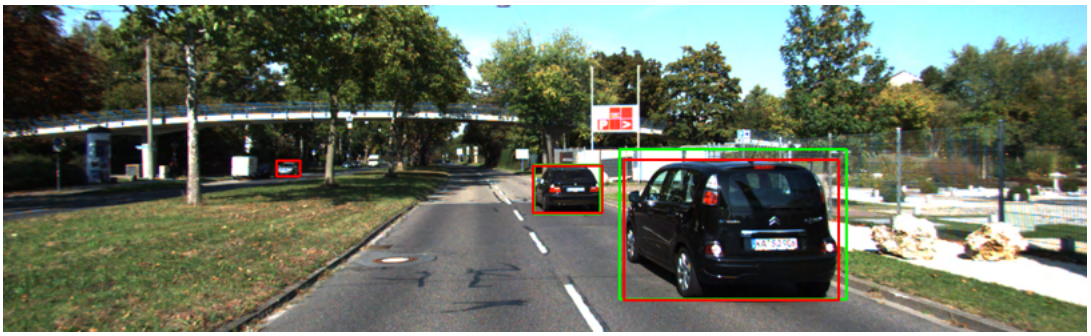


Figure 12: Image before applying Algorithm

```
if iou > 0.15:
    # Draw the detected box and display IoU as text
    x3, y3, x4, y4, conf = detected_box  # Detected box coordinates and confidence
    # Red rectangle for the detected box
    cv2.rectangle(vehicle, (x3, y3), (x4, y4), (0, 0, 255), 2)  # Red box for detected
    # Calculate IoU text position slightly above the top-left corner of the box
    text_position = (max(0, x3), max(0, y3 - 20))
    cvzone.putTextRect(vehicle,
            f'Car IoU: {math.ceil(iou * 100) / 100}',  # Round and display IoU
            text_position,
            scale=1.2,
            thickness=2,
            offset=1,
            colorB=(255, 255, 255),  # Background color (white)
            colorT=(0, 0, 0),        # Text color (black)
            border=0)               # No border to keep it minimal
```

Figure 13: Code Snippet of Filtering Algorithm

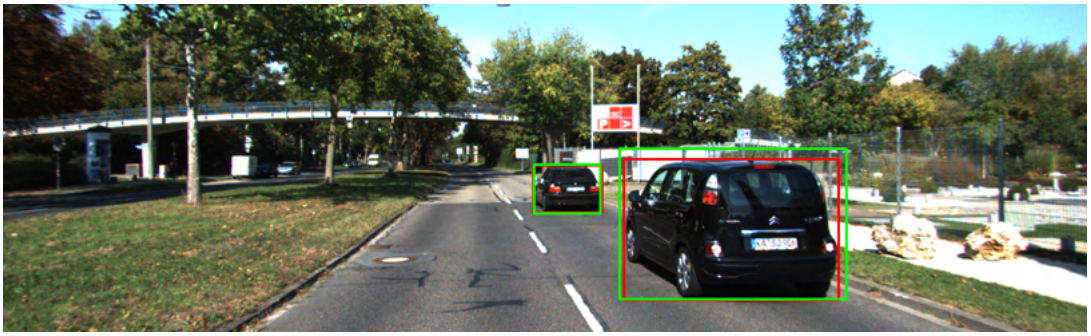


Figure 14: Image after applying Algorithm

2.5.1 Calculation of Precision and Recall values

After calculating the IoU values, they are evaluated against the threshold $\lambda = 0.75$ for further analysis. $\lambda = 0.75$ is chosen to ensure the model's detections are of high quality, but it's not so strict that it significantly lowers recall. This value offers a good compromise for performance evaluation. If the IoU value for a bounding box is greater than or equal to λ, it is classified as a **True Positive** and assigned a boolean value of 1, while a **False Positive** is assigned a value of 0. Conversely, if the IoU value is less than λ, the detection is classified as a **False Positive** (boolean 1) and not a **True Positive** (boolean 0). Next, the confidence values of the detected bounding boxes are sorted in descending order along with their corresponding IoU values. The *Tp* and *Fp* values are then assigned as described earlier. Precision and Recall values are calculated using the following formulas:

$$\text{Precision} = \frac{\text{True Positives (Tp)}}{\text{True Positives (Tp)} + \text{False Positives (Fp)}}$$

$$\text{Recall} = \frac{\text{True Positives (Tp)}}{N_{\text{gt}}}$$

where N_{gt} is the total number of ground truth values.

A calculation for one instance is given below

Detection Results Table:

Detection	Confidence	IoU	TP	FP	Precision	Recall	GT Box	Detected Box
1	0.90	0.67	0	1	0.00	0.00	(111, 184, 369, 334)	(161, 187, 358, 319)
2	0.89	0.89	1	0	0.50	0.33	(0, 189, 206, 374)	(0, 197, 195, 371)
3	0.87	0.95	1	0	0.67	0.67	(922, 147, 1241, 297)	(929, 151, 1241, 296)
4	0.83	0.00	0	1	0.50	0.67	N/A	(855, 148, 1111, 260)
5	0.48	0.00	0	1	0.40	0.67	N/A	(0, 147, 95, 201)

Final Average Precision (AP): 0.3889
no_of_ground_truth_box 3
no_of_detected_box 5

Figure 15: Mathematcial Calculations of image 006329

Then the Average Precision Value is calculated for all the images. It is detected by the formula

$$AP = \sum_{k=1}^{n} P_k \cdot (R_k - R_{k-1})$$

where P_k represents the Precision values and R_k represents the Recall values. Define $R_0 = 0$.

The Average Precision (AP) alng with Intersection Over Union(IOU) calculated for some of the images is given below.

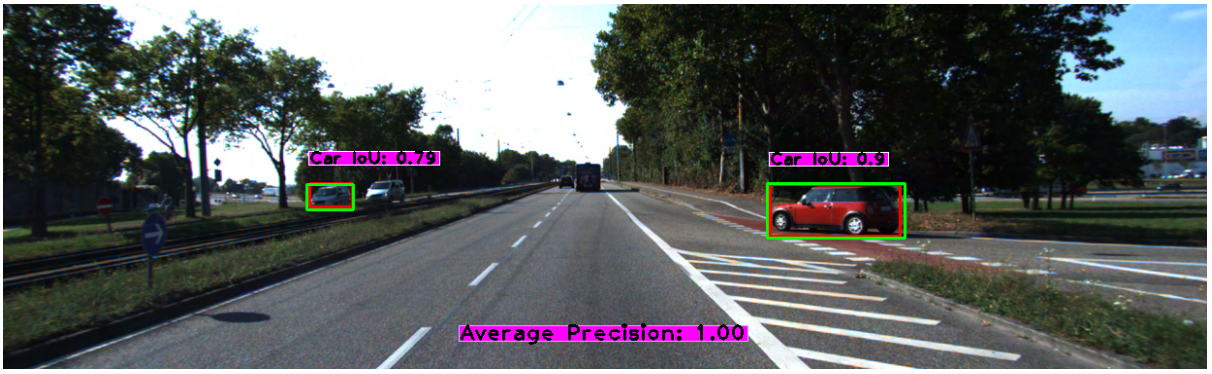


Figure 16: Average Precision Value for Image 006042

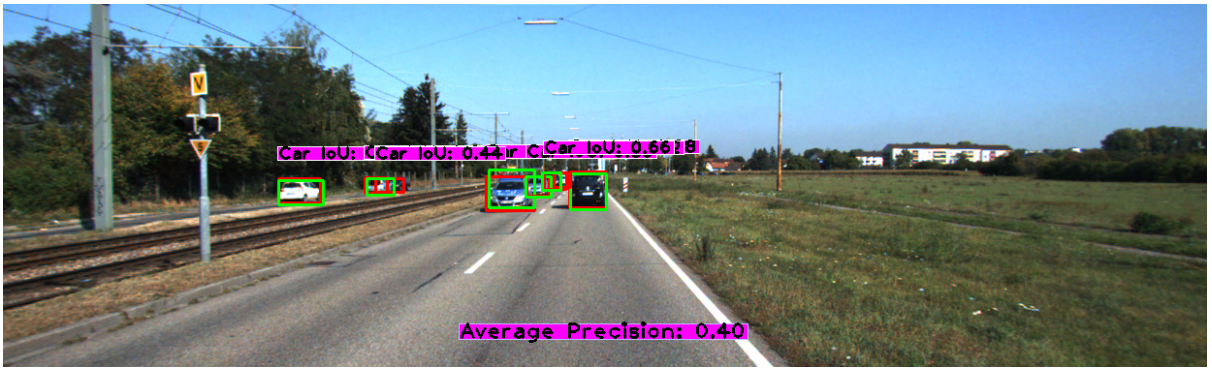


Figure 17: Average Precision Value for Image 006059

Figure 18: Average Precision Value for Image 006067

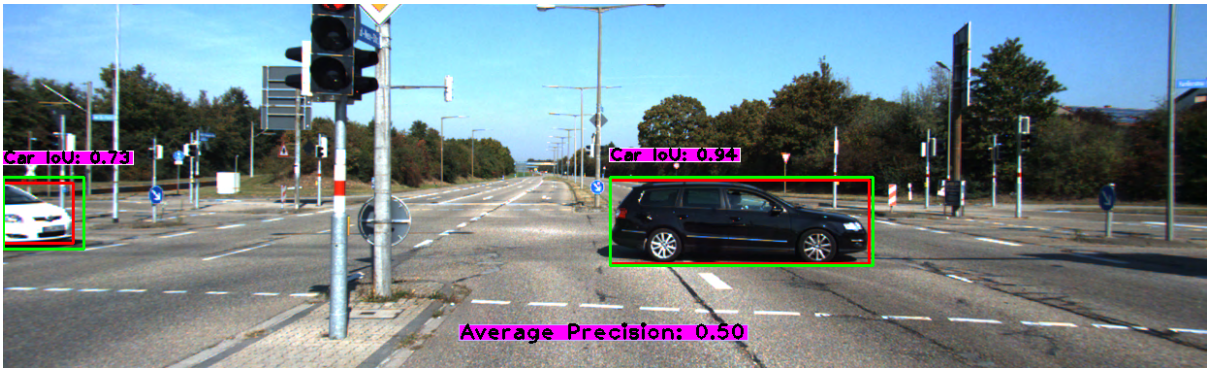


Figure 19: Average Precision Value for Image 006227

Figure 20: Average Precision Value for Image 006291

Figure 21: Average Precision Value for Image 006329

Chapter 3

Algorithm Implemented for Depth Estimation

As discussed above, the KITTI dataset contains the instrinsic matrix of the camera sensor itself in the form

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

where fx, fy are the focal lengths in pixels and s1,s2 are the are the coordinates of the principal point (also called the optical center) on the image plane in pixels, typically near the center of the image.
For calculating the depth, we use the midpoint of the lower bounding box[y_lower] as the camera height is given.

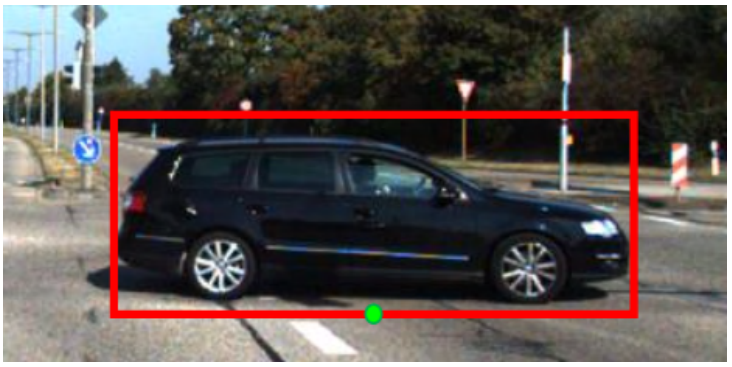


Figure 22: Point Green depicts the midpoint[1]

The depth can be estimated by the construction of two triangles, one is outside the camera and the other one inside the camera. The geometrical representation of the scenario is given below.

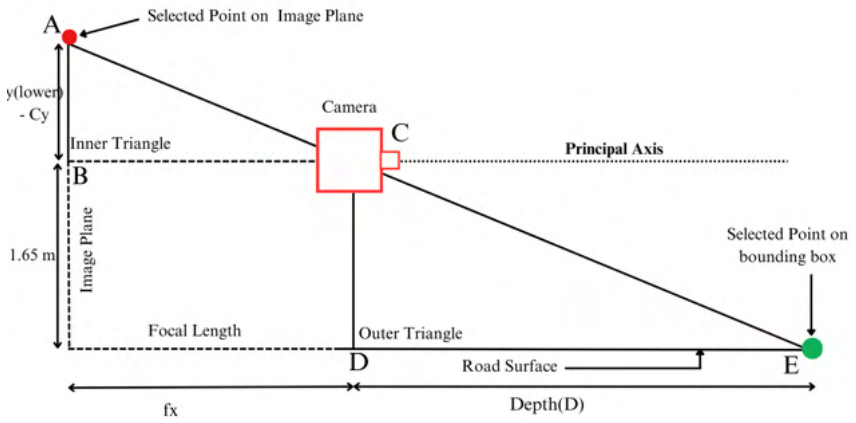


Figure 23: Geometrical Diagram of the setup

As the Triangles ABC and CDE are similar, the below formula can be used to calculate the depth estimation.

$$\frac{y_{\text{lower}-c_y}}{f_x} = \frac{1.65(camera\_height)}{\text{Depth}(D)}$$

The above can be rewritten as,

$$\text{Depth(D)} = \frac{f_x \cdot 1.65\text{m}}{\text{Image Height (in pixels)}}$$

where,

$$\text{Image Height (in pixels)} = y_{\text{lower}} - c_y$$

As the Image Height and fx is measured in pixels, the depth estimation is obtained in meters.

```python
# Load intrinsic matrix
def load_intrinsic_matrix(filepath):
    with open(filepath, 'r') as f:
        lines = f.readlines()
    matrix_values = []
    for line in lines:
        matrix_values.extend(map(float, line.split()))
    intrinsic_matrix = np.array(matrix_values).reshape(3, 3)
    return intrinsic_matrix
intrinsic_matrix_path = "C:\\Users\\Lenovo\\Desktop\\rw\\computer vision\\TASK_2_YOLO\\KITTI_Selection (1)\\KITTI_Selection\\calib\\006374.tx
K = load_intrinsic_matrix(intrinsic_matrix_path) # This loads the intrinsic matrix from a text file into the variable K.
camera_height = 1.65 # Camera parameters in meters
fx, fy = K[0, 0], K[1, 1] # Extract focal length (fx, fy) and principal point (cx, cy) from the intrinsic matrix (K).
cx, cy = K[0, 2], K[1, 2] # Principal point (the center of the image in pixel coordinates).
detected_boxes.sort(key=lambda x: x[4], reverse=True) # sorting based on confidence value
for detected_box in detected_boxes: # Iterate over each detection (now sorted by confidence)
    x1, y1, x2, y2, conf = detected_box # Unpack the bounding box coordinates and confidence of the detection.
    # Depth estimation using the midpoint of the lower bound (as per the provided formula)
    x_in_image = abs(((x1 + x2) / 2) - cx)# Calculate the horizontal distance from the principal point to the midpoint of the detected box
    image_height = abs(y2 - cy)
    # Calculate the vertical distance from the principal point (cy) to the bottom of the bounding box (y2).
    focal_length = fx # Use the focal length of the camera (fx) for depth estimation.
    image_distance = math.sqrt(x_in_image**2 + focal_length**2) # Use the Pythagorean theorem
    distance = (camera_height * image_distance) / image_height # Estimate depth formula.
    detected_depth = round(distance, 2) # Round the estimated depth to two decimal places.
```

Figure 24: Snippet of the depth calculation

Chapter 4

Results

The code snippet was executed within a loop for all 20 input images, along with their corresponding labels and calibration files. The algorithm did not work for the two frames as they did not contain any cars. The Ground Truth and YOLO depth values were stored in a list, which was subsequently used to create a tuple plot.

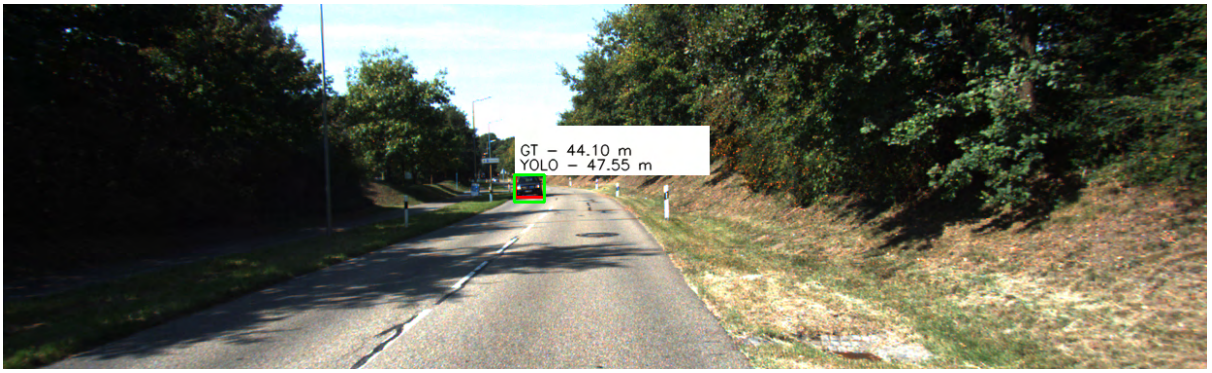


Figure 25: Output of Image 006206

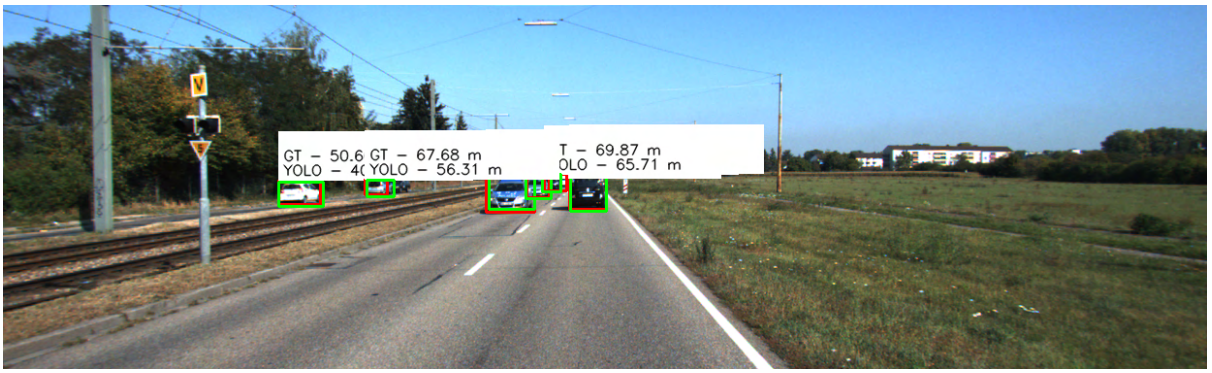


Figure 26: Output of Image 006059

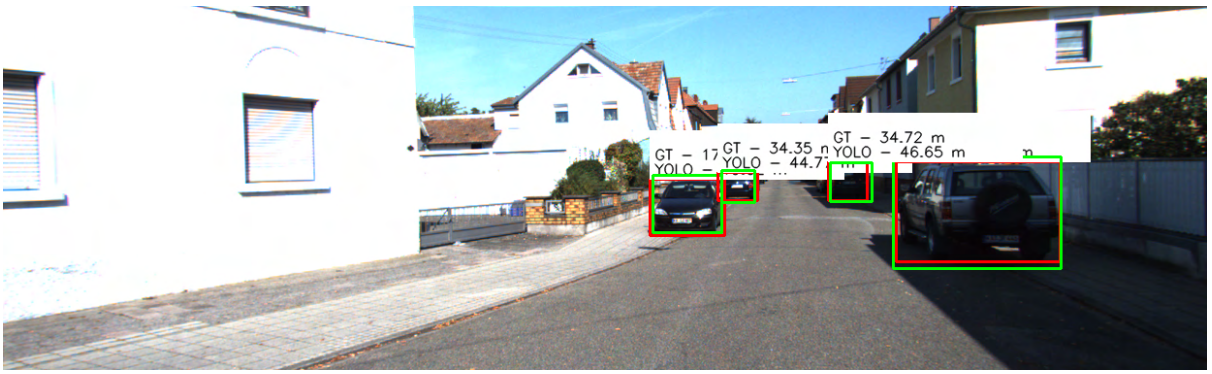


Figure 27: Output of Image 006098

Figure 28: Output of Image 006227

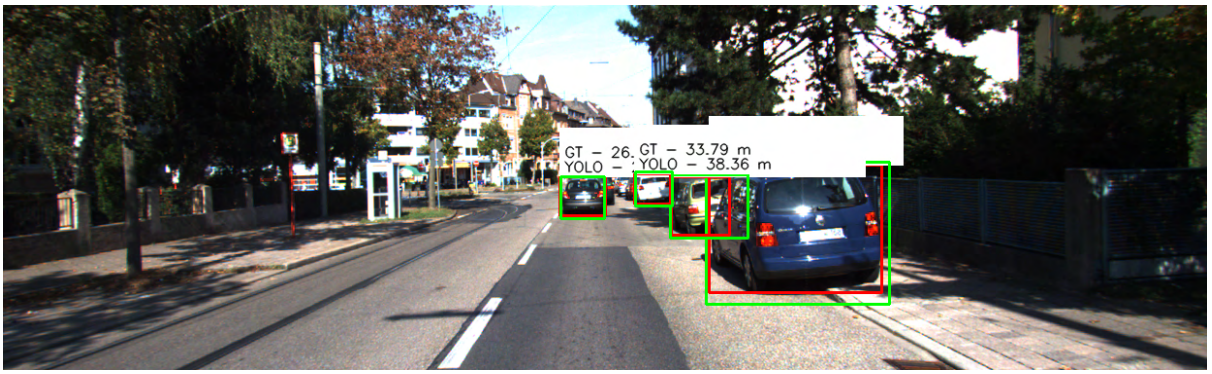


Figure 29: Output of Image 006253

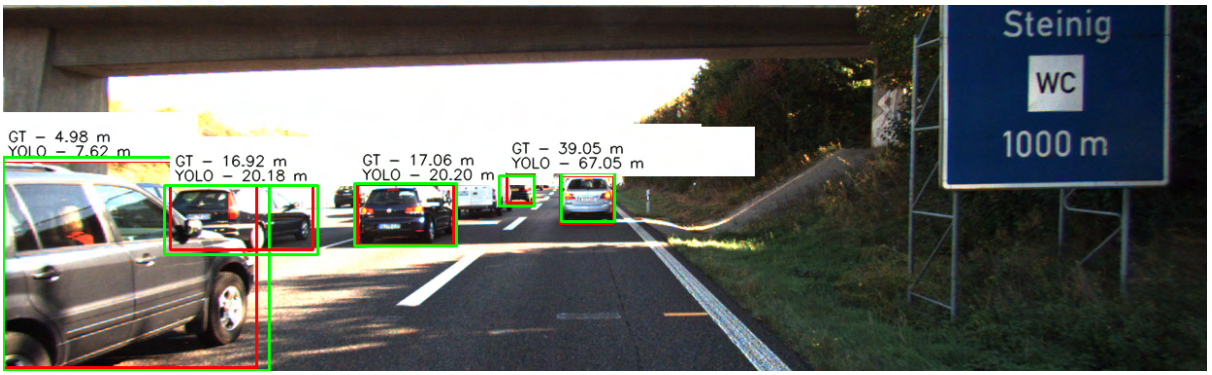
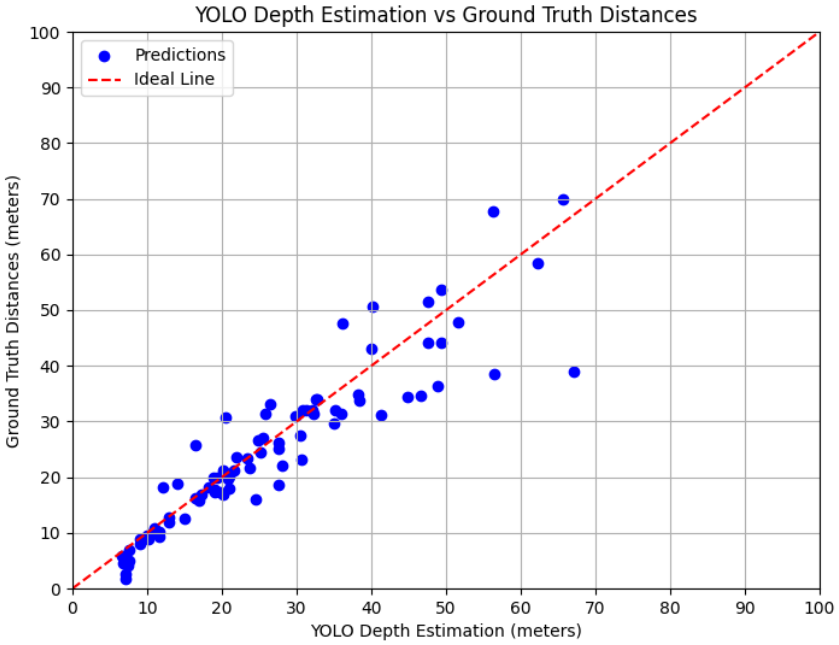


Figure 30: Output of Image 006310



A tuple plot was created with YOLO Depth (meters) on the x-axis and Ground Truth Detected Depth (meters) on the y-axis. An ideal straight line was included as a reference for comparison. From the analysis of the plot, it can be observed that YOLO's depth estimation is relatively accurate for objects within a range of less than 40 meters. However, between 40 and 70 meters, the depth estimation becomes less reliable, possibly due to the camera's inability to accurately detect objects at greater distances.

Other frames and Estimations

We reviewed certain frames that exhibited discrepancies in their depth estimation.
1) Greater distances were estimated for the cars, as they were located far away. Additionally, the camera alignment may have been inaccurate, potentially due to the terrain being a hilly area.

Figure 31: Object at Far distances

2) The camera is unable to detect the car properly, possibly due to low lighting conditions.

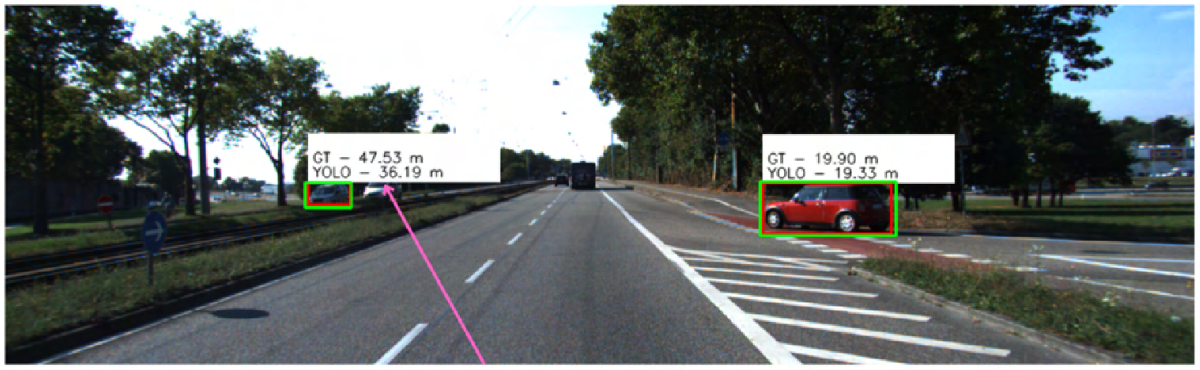


Figure 32: Object under diminished Light condition

3) Another issue we encountered is that the cameras are unable to detect objects that are closer to them. This may be due to the camera's field of view, which starts at approximately 7 meters, as detection below this range is not possible due to limited pixel resolution, sensor characteristics, or algorithmic constraints. As a result, objects within this range are detected at a distance of 7 meters, rather than their actual distance.

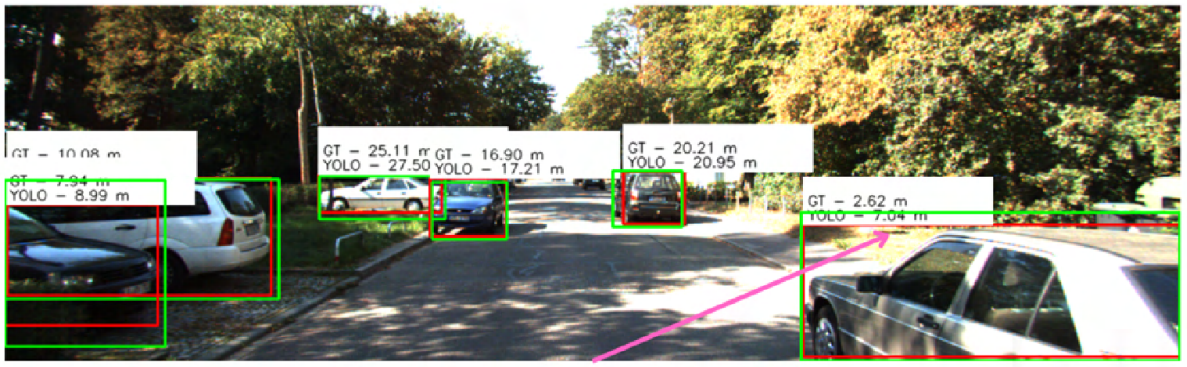


Figure 33: Objects which are closer to the camera

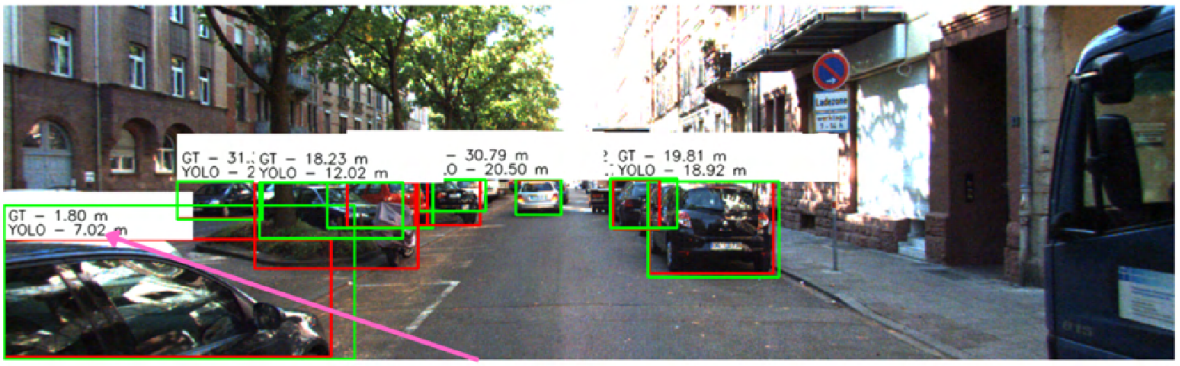


Figure 34: Objects which are closer to the camera

Conclusion

In reflection on the results, the observed error percentages and anomalies in depth estimation using a single monocular camera highlight significant limitations, making it unsuitable as the sole source of depth information. Critical factors such as uneven gradients in the driving surface, lens distortion, the accuracy of object detection models, and proper calibration significantly influence overall performance.

Additionally, discrepancies in the number of car detections between the ground truth annotations and YOLOv11m outputs were observed. Addressing this issue could involve enhancing the ground truth dataset with additional car detections to reduce mismatches. Furthermore, if the objective is to detect only a specific class of objects, training a custom YOLO model tailored for that class using a specialized dataset may yield more precise results.

Despite these limitations, monocular camera-based depth estimation remains viable in certain scenarios, such as highway environments. It can serve as a practical solution for vehicles lacking radar sensors by assisting in maintaining safe following distances and providing driver alerts in situations of close proximity.

Ultimately, by utilizing a subset of the KITTI dataset comprising the necessary images and calibration data, the task of vehicle depth estimation was successfully achieved. This was accomplished through the construction of a two-dimensional geometric model to estimate the unknown depth.

Github Repository:
https://github.com/nirbhayborikar/-Car_detection_and_depth_estimation-_opencv_and_yolo11m