Vehicle Cut-In Detection Using Yolov8 and TTC Calculation In Single Forward Pass

A report on PS-8: Vehicle Cut-In Detection for the Intel Unnati Industrial Training Program

By Satvik Beura

1. Technical Approach

1.1. Loading Data and Calculating Speed

The initial step involves loading the necessary data from a CSV file containing positional data and timestamps. (The path to be used can be modified by changing the “MAIN_PATH” variable.) This includes latitude, longitude, and altitude for each image frame. Using these absolute data points, the relative speeds between consecutive frames are calculated.

The function “calc” calculates the distance between two geographical points on earth using the Haversine formula, which considers the curvature of the Earth. The altitude difference is also calculated from the given coordinates.

The resulting speed is then smoothed using an Exponential Moving Average (with alpha=0.2) to maintain consistency and fix the issue of obtaining absolute speeds for every consecutive frame, which is then stored in the array “smoothed_speeds”. Storing the values is because of the pre-recorded nature of the data, which can easily be modified to fit real-time analysis.

1.2. Object Detection Using YOLOv8

To detect vehicles and objects, the YOLOv8 model from the Ultralytics module is used. The small model (yolov8s) was preferred after testing all other models as it gave the results with the highest accuracy considering the minimal inference time on GPU. (~15ms)

For each image, the model predicts bounding boxes, class IDs, and confidence scores.

1.3. Distance Calculation from Bounding Boxes

The height of each detected object class is used to estimate the distance from the camera using the point at (x/2, y/3) of each bounding box (to reduce the amount of false positives) and the estimated focal length of the camera, using the formula given in section 2.3 of this paper (https://arxiv.org/abs/2106.10319). The distance is taken into consideration only if the point falls within two lines denoting the forward direction of the camera car (as considering objects outside this area leads to reduced accuracy and a significant increase in the number of false positives), marked in two white lines, whose equations are considered as inequalities to judge which lane a point falls into.

1.4. Cut-In Detection

The Time-To-Collision (TTC) is calculated using the predicted distance and the speed at current frame and compared to a threshold (0.65s), below which a warning is issued, along with a

displayed image with red tone. The model is evaluated in two modes, one with a random image from the dataset, and the other on all the images in the dataset. The first mode annotates all detected images with confidence > 0.3, to check the model's functions on a random image from the dataset, the second mode is suited for real world applications, as it does not show every image with its annotations, but only the ones where a cut-in is detected. This allows decreasing evaluation time (~30fps with P100 GPU). The print of evaluation of every image is not recommended in constant evaluation mode as it significantly increases time of evaluation (since I used matplotlib.pyplot, evaluation is paused till graph is plotted). A workaround of this could be saving the cut-in detection images as logs in an output folder, since the dataset doesn't wait for the plotting.

Code Link:
https://github.com/ravesandstorm/Vehicle-TTC-Calculation/blob/main/Files/ttc-calc.ipynb

2. Issues faced:

2.1. Finding speed data

The dataset used (IDD Multimodal Primary, Secondary from https://idd.insaan.iiit.ac.in/) contained multiple images (Primary) along with OBD sensor data and Lidar sensor data (Secondary). The OBD Sensor contained data about the car, containing speed data. But from the timestamps given, the speed data seemed to be updated every 1.5s, which would give very inconsistent results if used to estimate TTC, as the TTC threshold itself was less than half (0.5s-0.7s) of the updation value, which would lead to inconsistent results and increase in the number of false positives. To handle this issue, current frame speed was estimated from the given positional data in latitude, longitude and altitude.

2.2 Distance estimation

The Lidar sensor data obtained from the secondary dataset is a valid method to calculate exact distances, but considering the file size for each frame, the increase in overall inference time was worth considering another approach to estimate distance.

This approach required the specifications of the camera, which I was not able to find. I estimated the focal length of the camera using perspectives of images taken from multiple other images online and made calculations based on that. The accuracy of the model depends heavily on this focal length estimated, and should be calibrated according to real life values.

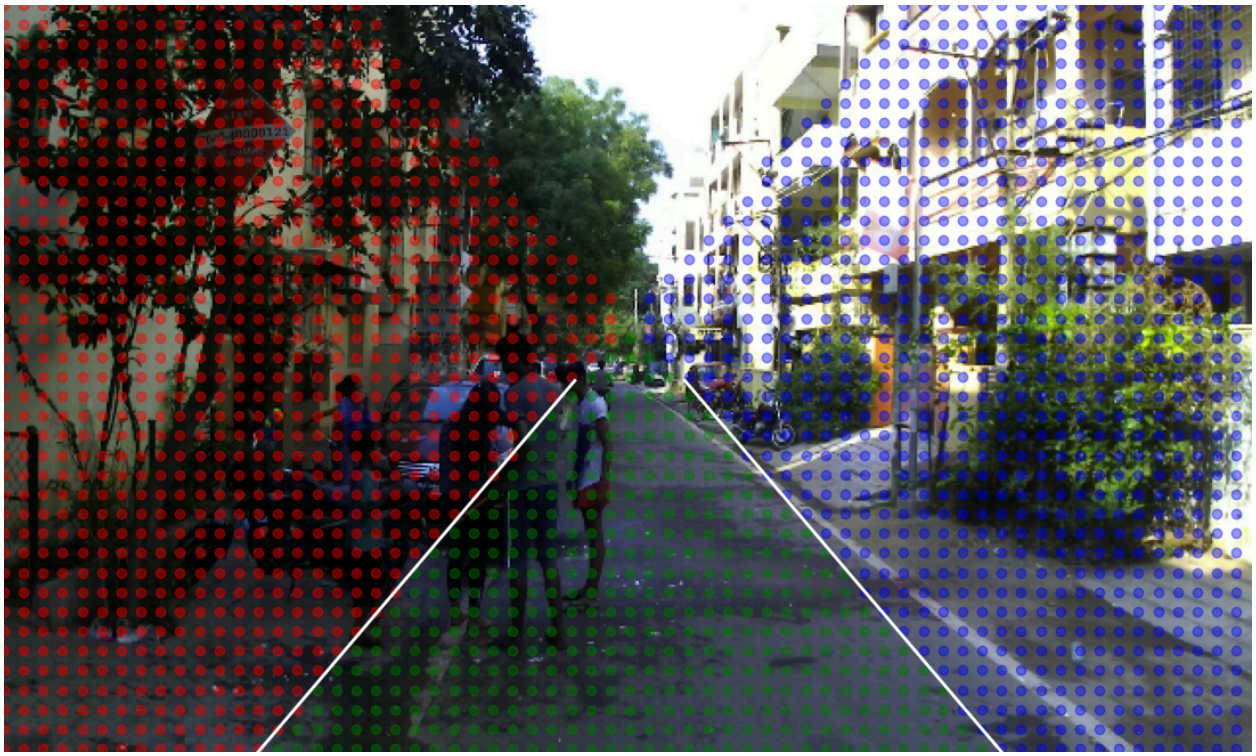
2.3 Lane estimation

The definition of a lane isn't explicitly mentioned and might change based on which side of the road the vehicle is on (right or left), which meant I had to estimate the points based on my observations, and use an image that had the camera roughly in the middle of the lane. A lane system was used to reduce the number of false positives.

The inequality "m2*(x1-(1-top_val)*image_width) - y1 + image_width/2 > 0" was used first and resulted in a classification like this, where 2 is the middle lane.

Because of inaccurate predictions, I plotted a point for every 10 pixels and gave different colors to every section, across the whole image, with the middle lane in green, which led to very bad results, meaning I had to change my equation.

Changing the equation to “m1 * (x1 - top_val * image_width) - y1 + image_height/3 < 0” resulted in a correct lane prediction graph.



I changed the equation to “-192 * x1 - 163.84 * y1 + 87490.56 < 0” after calculating the line equations myself considering the 4 points on the image, which would result in less calculation time for every point.

3. Results

3.1 Overall Evaluation Time

For each device, I calculated the time for evaluating 1000 frames, from which I was able to get the frame rate the model was capable of without falling behind, in order to give an accurate TTC calculation and warning.

For 100 frames, time = 42.92s on CPU. (NOT RECOMMENDED)

CPU is generally not recommended for inference on images at all, getting a very high evaluation time for just 100 frames.

For 1000 frames, time taken = 34.1369 on the NVIDIA T4 GPU, ~28 FPS.

This device gave very good inference times, capable of running the project reliably with real time image data of 24 FPS.

For 1000 frames, time taken = 30.0911 on the NVIDIA P100 GPU, ~33FPS.
This device gave the best inference times, capable of running the project reliably with real time image data of 30 FPS.

The links for the videos are:

Model On Kaggle.mp4
Output-video-1.mp4