Lab Guide
LaneNet Lane Estimation

Content Description

The following document describes the implementation of the LaneNet lane estimation in python environment.

Lab Description

In this example, we will capture RGB images from the Intel RealSense camera, use LaneNet to estimate lane markings on the image, perform clustering on the output lane instance embeddings, overlay the RGB image with the estimation lanes, and display result. This pipeline is shown in Figure 1.

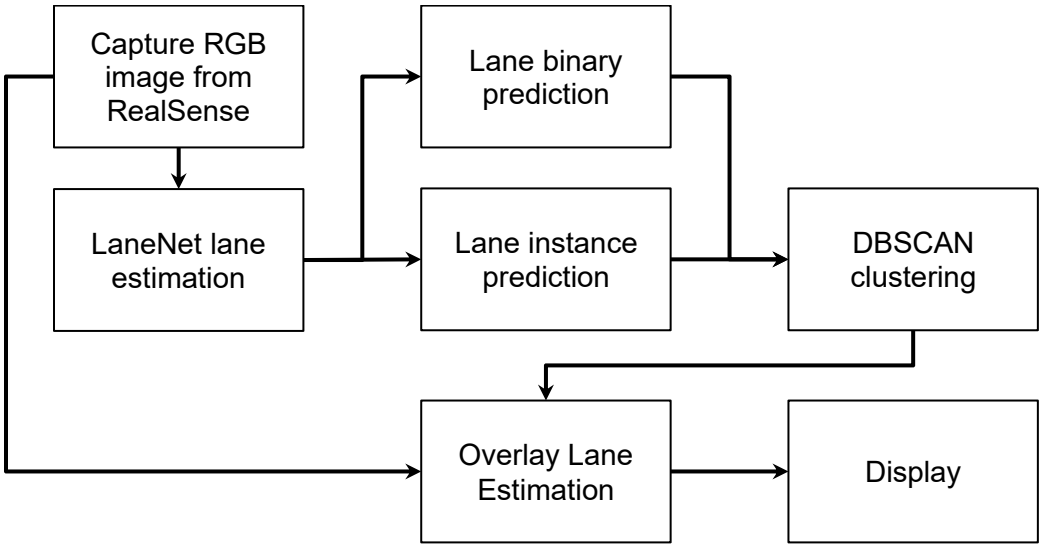

Figure 1. Component Diagram

Python

Running the example

1. Check User Manual – Software Python for details on deploying python scripts to the QCar2. The output of the **cv2.imshow()** function should look like Figure 2.

2. When running the script for the **first time**, make sure the QCar 2 is **connected to internet**, as the trained PyTorch model will be downloaded. In addition, the downloaded model will be converted to a TensorRT engine to improve inference time. The whole process can take up to 20 minutes.

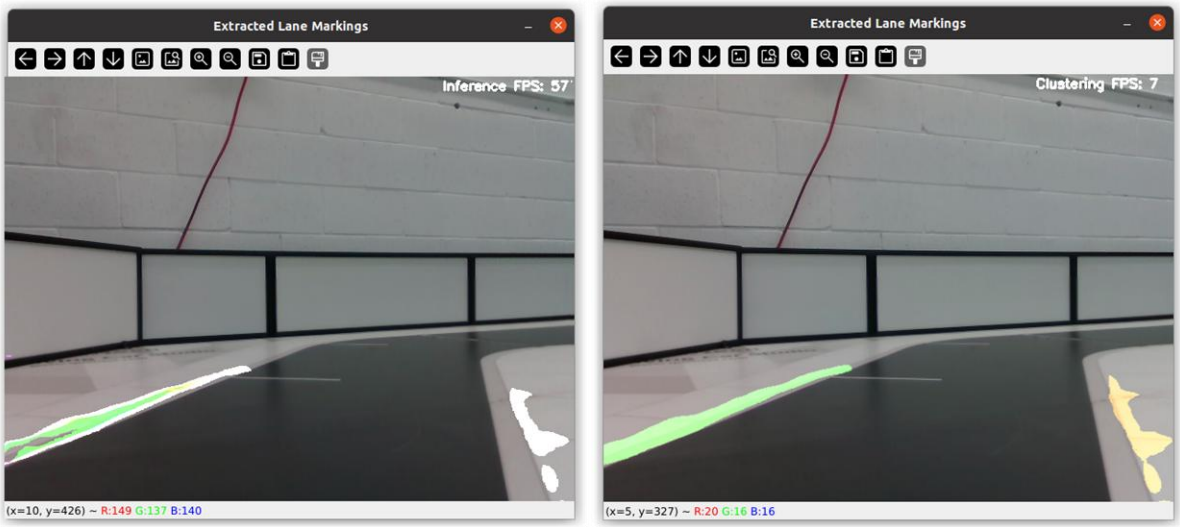

Figure 2. Annotated Camera Feed Before Post-processing (left) and After (right)

Details

1. Initialization

```
myLaneNet = LaneNet(
                    # modelPath = 'path/to/model',
                    imageHeight = imageHeight,
                    imageWidth = imageWidth,
                    rowUpperBound = 240
                    )
```

To initialize the LaneNet model, an optional path to a pretrained model can be provided. Otherwise, the default LaneNet model trained by Quanser will be loaded. The expected height and width of the camera feed also need to be provided at initialization, as they will be used to resize the predictions so they can be overlaid with the original image. The *rowUpperBound* defined the row index above which the input image will be cropped out. A *rowUpperBound* of 0 indicates that the entire

input image is used. After initialization, the LaneNet model is created and stored under the **myLaneNet.net** attribute. This implementation of LaneNet is based on [Iroh Cao's implementation](#).

2. Pre-processing and Prediction

```
rgbProcessed=myLaneNet.pre_process(myCamRGB.imageBufferRGB)
binaryPred , instancePred = myLaneNet.predict(rgbProcessed)
```

Before prediction takes place, the input image needs to be cropped at the row indicated by ***rowUpperBound*** and be pre-processed into the tensor with a dimension of 1x3x256x512 (BxCxHxW), as well as applying the same batch normalization as the training set.

The processed image should be used as the input for the **predict()** method. The outputs of the **predict()** method are the binary image (***imageHeight*** x ***imageWidth***) which indicates the location of all estimated lane markings (***binaryPred***), and ***imageHeight*** x ***imageWidth*** x3 embeddings which can be used to determine separate instances of lane markings (***instancePred***). The output of the **predict()** method is resized and stitched such that they are in the same shape as the camera feed, the raw output can be accessed by calling ***myLaneNet.binaryPredRaw*** and ***myLaneNet.instancePredRaw***

3. Post-processing and render

```
isolatedLane = myLaneNet.post_process(eps=0.5,
                                      min_samples=250,
                                      min_area=100)
# annotatedImg = myLaneNet.render(showFPS = True)
annotatedImg = myLaneNet.post_process_render(showFPS = True)
```

The **render()** method returns the annotated RGB which has overlaid estimated lane markings. Since the embeddings are configured to have a dimensionality of 3, it can be easily visualized. Lane instances that are closer in the embedding space will appear to have more similar colors. The **post_process()** method utilizes scikit-learn API to perform DBSCAN clustering on the masked lane instance embeddings. Each output cluster indicates a separate lane marking. The **post_process_render()** method overlays the isolated lane markings with the input camera feed. The **render()** method can be called without calling **post_process(),** while **post_process_render()** has to be called after calling **post_process().**

4. Performance considerations

DBSCAN is a computationally expensive process, making **post_process()** run at a much lower rate than **predict()**, as shown in Figure 1. For practical self-driving applications, consider isolating lane markings via connected component analysis on the binary prediction. In addition, **render()** and **post_process_render()** are also noticeably time-consuming processes, consider not rendering the predictions to further reduce computation time.