ECE 484: Principles of Safe Autonomy	Fall 2025
Written by: ECE 484 team	*MP1 and HW1*
Website 	*Due Date: September 26, 2025*

1 Introduction

This assignment is consisted of two major parts. In the first part, you will answer 5 written questions related to concepts in computer vision. In the second part, you will implement a camera-based lane detection module using a neural network. The module will take a stream of raw images as input, and it will produce a stream of annotated images with the lane area marked as the output. You will implement the functions detailed in later in the documentation. You will also write a brief report `mp1_<groupname>.pdf`. In the assignment, you will gain some experience with Robot Operating System (ROS2) [1], and you are encouraged to use Pytorch functions to help you get familiar with neural networks, unless specified otherwise. This document will guide you to the first steps on implementing the lane detection module. You are free to take advantage of tutorials and documentations available online. **Make sure to cite all resources in your report. All the regulations for academic integrity and plagiarism spelled out in the student code apply.**

Learning objectives

- Binary Segmentation with a Neural Network
- Coordinate Transformation
- Working with ROS – Gazebo
- Working with OpenCV Library

System requirements

- Ubuntu 22.04
- ROS 2 Humble

2 Homework Problems

Note that for proofs, we expect you to write the proof yourself, and not simply copy the answer off of the internet/peers. Proving these on your own is good practice for quizzes/midterm. An aside, typically its quite easy to determine when students copy proofs verbatim. We are happy to help in office hours, homework party, or Campuswire.

Problem 1 Convolution [Individual] (15 points) Let an input tensor $\mathcal{X}$ have space $C_{\text{in}} \times W \times H$. Consider a 2-D convolution with C_{out} filters, kernel size k, stride s, padding p, and dilation d (Dilation in CNNs refers to inserting spaces between the elements of a convolutional kernel). Compute the expressions as a function of $k, s, p, d, C_{\text{in}}, C_{\text{out}}, W, H$ for the dimension of the output tensor.

Problem 2 Linear transformation of the convolution layer [Individual] (20 points) We have input tensor $\mathcal{X} \in \mathbb{R}^{4\times4}$, kernel $\mathcal{K} \in \mathbb{R}^{3\times3}$, stride=1, no padding. Please write out the output of $\mathcal{X} * \mathcal{K}$, where $\mathcal{X} * \mathcal{K}$ is convolution on $\mathcal{X}$ with kernel $\mathcal{K}$. Then, we flatten X into $\text{vec}(\mathcal{X}) = [1, 2, 3, 4, ..., 14, 15, 16]^\top \in \mathbb{R}^{16\times1}$, and we want a matrix A, where $A\text{vec}(\mathcal{X}) = \text{vec}(\mathcal{X} * \mathcal{K})$. What is the shape of A? What are the values in A?

$$\mathcal{X} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \mathcal{K} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

Problem 3 Computer Vision [Individual] (15 points) You are given the pixel coordinates $(x_i, y_i) = (400, 300)$ of a point observed by a camera, along with the camera's intrinsic matrix

$$\mathbf{K} = \begin{bmatrix} 800 & 0 & 320 \\ 0 & 800 & 240 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad [\mathbf{R} \,|\, \mathbf{t}] = \begin{bmatrix} 0.6 & -0.8 & 0 & 1 \\ 0.8 & 0.6 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{bmatrix}.$$

Recall that $[\mathbf{R} \,|\, \mathbf{t}]$ is the extrinsic matrix. Assuming that the depth of the point in the camera coordinate system, $\tilde{z}$, is $5\,\text{m}$, calculate the 3D world coordinates X_w, Y_w, Z_w of the point.

Problem 4 Backpropagation [Individual] (20 points) Consider a single neuron with

$$\hat{y} = f(wx + b),$$

where $f(z) = z^2$, the input $x = 2$, and the target $y = 8$. The initial parameters are $w = 1$ and $b = \frac{1}{2}$.

(a) Compute the forward pass and loss using the mean squared error:

$$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2.$$

(b) Compute the gradients:

$$\frac{\partial \mathcal{L}}{\partial w}, \qquad \frac{\partial \mathcal{L}}{\partial b}.$$

(c) Update w and b using gradient descent with a learning rate $\alpha = 0.001$.

Problem 5 Backpropagation [Individual] (30 points) Consider a neural network with two hidden layers:

$$\mathbf{h}^{(1)} = f\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right), \quad \mathbf{h}^{(2)} = f\left(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right), \quad \hat{y} = W^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)},$$

where f(z) is the custom element-wise activation function defined as:

$$f(z) = \begin{cases} z^2 & \text{if } z \geq 0, \\ -0.5z & \text{if } z < 0. \end{cases}$$

and $W^{(1)}, W^{(2)} \in \mathbb{R}^{2\times2}, W^{(3)} \in \mathbb{R}^{1\times2}, \mathsf{b}^{(1)}, \mathsf{b}^{(2)}, \mathsf{x} \in \mathbb{R}^{2\times1}, \mathsf{b}^{(3)} \in \mathbb{R}^{1\times1}$.

(a) Derive symbolic expressions for the following gradients:

$$\frac{\partial \hat{y}}{\partial W^{(3)}}, \quad \frac{\partial \hat{y}}{\partial \mathbf{b}^{(3)}}, \quad \frac{\partial \hat{y}}{\partial W^{(2)}}, \quad \frac{\partial \hat{y}}{\partial \mathbf{b}^{(2)}}, \quad \frac{\partial \hat{y}}{\partial W^{(1)}}, \quad \frac{\partial \hat{y}}{\partial \mathbf{b}^{(1)}}.$$

(b) Using the given network parameters:

$$W^{(1)} = \begin{bmatrix} 2 & -1 \\ -3 & 1 \end{bmatrix}, \quad W^{(2)} = \begin{bmatrix} 0.5 & 0 \\ 1 & -2 \end{bmatrix}, \quad W^{(3)} = \begin{bmatrix} 1 & -1 \end{bmatrix},$$

$$\mathbf{b}^{(1)} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{b}^{(3)} = [0.2], \quad \mathbf{x} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Compute the output $\hat{y}$ and all gradients derived above.

3 MP1 Overview

3.1 Problem Statement

The commercial success and the proliferation of shared-autonomous vehicles in the recent decade has sparked a keen public interest in its underlying technology. **Lane detection** is a well-researched topic with a rich body of literature and is still an active area of research. This MP seeks to provide students with hands-on experience of designing a simple lane detection module using a neural network in a simulated environment; this will serve as a practical introduction to neural perception in autonomous driving.

3.2 System Diagram

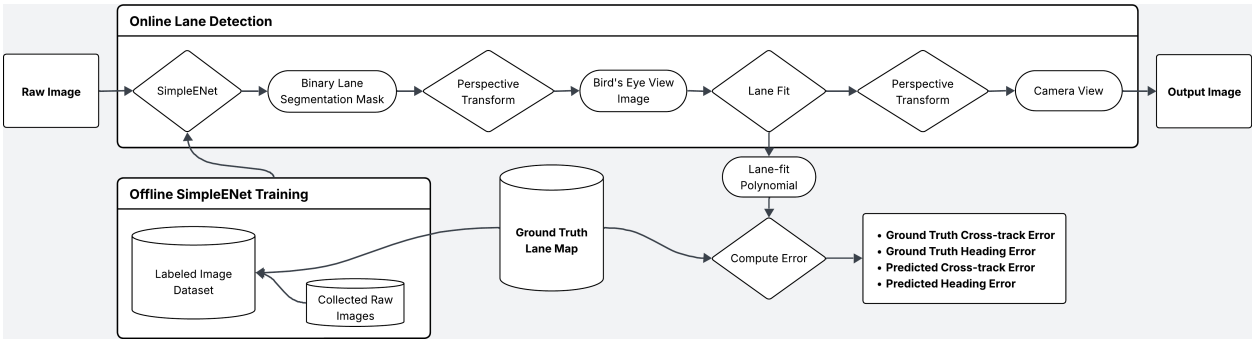


Figure 1: Lane Detection System Diagram

The system diagram of our lane detection pipeline is shown in Figure 1. First, you will train a simple neural network of your design using the ground-truth lane map to generate labels for collected raw images. During lane detection, the trained model outputs a binary mask image of lane lines given a raw image. This segmentation mask then gets converted to a bird's eye view (BEV) image. Finally, a lane fitting module fits polynomials to the lanes, returning an image with the detected lanes overlaid to the raw image as well as the polynomials. The system's performance is evaluated by computing errors, such as cross-track and heading errors, by comparing the fitted lane polynomial with the ground truth lane map. This MP requires you to implement only some of the said components, as listed under Section 4. Of course, you are encouraged to experiment with the entire codebase, as long as those changes are reverted upon submission.

4 Implementation Details

Below is a brief description of what you will implement in this MP.

Network Architecture Design an *architecture* of a simple neural network for lane segmentation.

End-to-End Neural Network Training Implement the *training loop* for a simple neural network.

Lane Segmentation Run *inference* on the neural network to obtain a binary lane segmentation image.

World-to-pixel Coordinate Transformation Perform a *world-to-pixel coordinate transformation* to obtain an accurate bird's eye view (BEV) image.

All the scripts you will need to modify are:

- `mp1/scripts/simple_train.py`
- `mp1/src/models/simple_enet.py`
- `mp1/scripts/run_bev_config.py`
- `mp1/scripts/run_lane_detection.py`

4.1 Network Architecture

In this section, we will implement a neural network capable of image segmentation to determine lanes within an image. The model takes a gray-scale image as input and produces pixel-wise predictions, classifying each pixel as either part of a lane or not. We will build a simplified version of ENet, a lightweight and efficient network used for real-time segmentation tasks. We provide four modules to build Simple ENet, and your task is to complete the missing blanks in the InitialBlock. The detailed flow of the InitialBlock is provided below:

- InitialBlock takes the input image with the shape of (1, 384, 640)
- Input image –> Convolution Layer –> Feature f_1 with the shape of (out_channels-1, 192, 320)
- Input Image –> Maxpooling Layer –> Feature f_2 with the shape of (1, 192, 320)
- Concatenate f_1 and f_2 along the second dimension (axis=1) –> Batch Norm –> Activation –> Output with the shape of (out_channels, 192, 320)

After building the modules for Simple ENet, we need to obtain images to train the model on. To do this, we have provided a script, `run_collect_images.py`, where you are able to drive the vehicle with WASD as well as save images. From here, `preprocess_data.py` will generate the labels for each image.

4.2 End-to-End Neural Network Training

In order to train the neural network, we need a training loop, which defines how the model processes data, computes loss, and updates its parameters through optimization. Your task is to choose the optimizer and update the model parameter with back propagation.

Optimizer is a key component of training, which adjusts the model's weights based on the computed gradients. You should select an appropriate optimizer (e.g., SGD, Adam, or AdamW) and initialize it with the model's parameters, along with essential hyperparameters such as the learning rate. During training, we iterate over the dataset using a DataLoader, moving input images and labels to the chosen computation device. After feeding the images into the model and performing a forward pass, we obtain the outputs and compute the loss. Then, we clear any previously accumulated gradients, perform a backward pass to compute gradients with respect to the loss, and use the optimizer to update the model parameters.

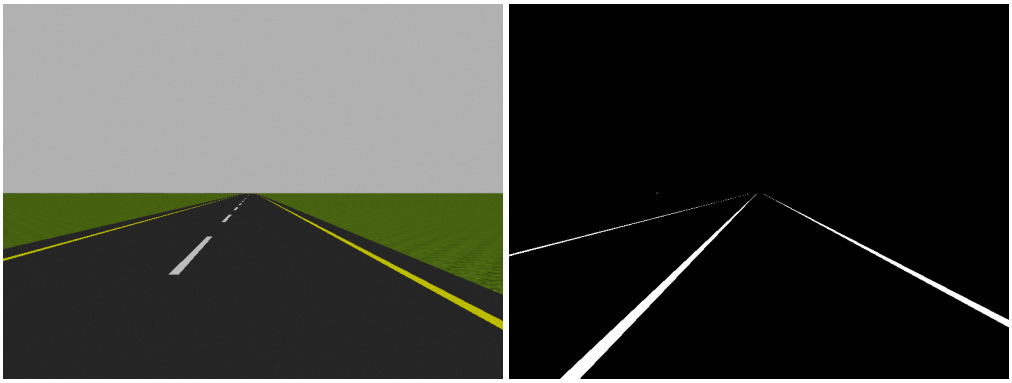
This completes one iteration of training. Over time, as the training progresses, the loss should gradually decrease and eventually converge, reflecting the model's improved ability to perform lane segmentation.

4.3 Lane Segmentation

In this section, we will implement a binary lane segmentation pipeline using a neural network. The goal is to process a raw image and extract a binary mask highlighting lane pixels during the inference time. The main steps are as follows:

1. **Resize** the input image to (640×384), matching the model's expected input resolution.
2. **Grayscale and normalize** the image to single-channel float values in the $[0, 1]$ range.
3. **Convert** the preprocessed image into a PyTorch tensor and add the channel and batch dimensions to meet the model input requirements.
4. **Run inference** on the input using the pretrained model to get a pixel-wise classification output in real time.
5. **Post-process** the output by taking the argmax over the class dimension to get a predicted class map, convert it into a binary format, and finally resize it back to the original image dimensions.

The model is expected to output a two-channel tensor (background vs. lane), where the second channel corresponds to the lane class. By applying `argmax` over the channel axis, we extract the most probable class for each pixel. Multiplying this prediction by 255 gives us a binary mask with pixel values of either 0 (non-lane) or 255 (lane), suitable for visualization and downstream processing.



(a) (b)

Figure 2: (a) Input raw image (b) Binary lane segmentation mask

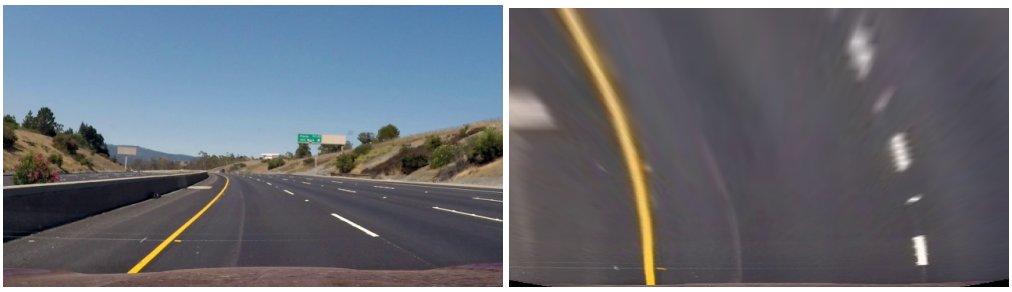
This function will serve as the backbone for visual lane perception in your module. **Make sure your preprocessing steps exactly match the training pipeline. Otherwise, model performance will degrade significantly.**

4.4 World-to-pixel Coordinate Transformation

In this section, you will implement world-to-pixel transformation to obtain an accurate bird's eye view (BEV) image. What is an "accurate" BEV image? A proper BEV should preserve *collinearity* (i.e., all points lying on a line initially still lie on a line after the transformation) as well as the resulting image displaying *parallel lane lines*. We achieve these by using a 3-by-3 perspective transformation matrix.

$$\begin{bmatrix} tu \\ tv \\ t \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1}$$

Here (x, y) and (u, v) are the coordinates of the same point in the coordinate systems of the original perspective and the new perspective. To find the matrix, we need to find the location of 4 points on the original image, namely the *source points*, and map the corresponding 4 points, on the BEV. Any 3 of those 4 points should not be on the same line.



(a) (b)

Figure 3: (a) Raw image (b) Bird's Eye View image

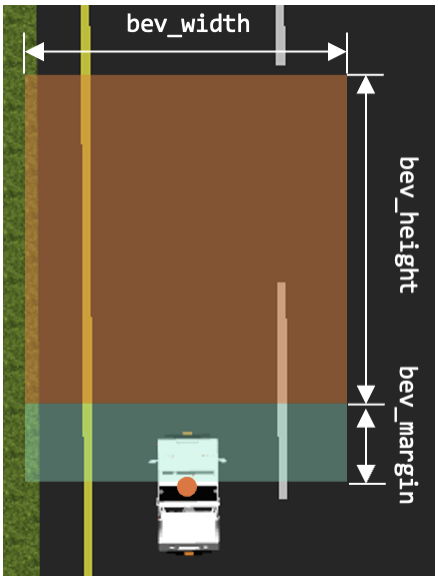


Figure 4: BEV rectangle with the hyperparameter names

Instead of choosing the source points manually, we will perform a world-to-pixel coordinate transformation of the corners of the rectangle in the world frame, determined by its height, width and the distance between its lower edge and the base link of the vehicle, shown in Figure 4. This offers several key advantages:

1. The resulting BEV image displays perfectly parallel lane lines

2. It enables a quick iteration of many different BEV configurations without much trial-and-error.

You are free to experiment with the hyperparameters, but for the submission, please return them to the original values for a more reliable evaluation. To save the BEV configuration as a JSON format, which other modules then can access, run:

```
python3 scripts/run_bev_config.py
```

5 Training the Model

Important: All the python execution commands below should be run in the `mp1/` directory. Relative paths in the code (e.g. output save paths) will fail if run in other locations.

5.1 Collecting Training Images

To train our model, we need a training dataset. You will collect the dataset yourself! We provide the script `scripts/run_collect_images.py` that controls the GEM car around the map and captures front camera images at your desired location. Use 'W','A','S','D' to control the movement and press 'C' to save the front camera view as PNG at `data/raw_images/`.

```
python3 scripts/run_collect_images.py
```

5.2 Labeling the Images

Upon collecting the raw images, we need to label them to train the model. One significant advantage of using simulation is that we have direct access to the ground-truth information, which may be difficult to obtain in the real world. Taking advantage of this fact, we have written a script that computes the ground-truth lane segmentation mask for each image you have taken in the previous step. We first need to generate *lane maps* that we will use to label your images. You can do so by executing the script:

```
python3 scripts/generate_map.py
```

If successful, the new directory `data/lane_map/` will contain raster images of road components that make up the entire map. Now, to generate the labeled dataset, simply run:

```
python3 scripts/preprocess_data.py
```

Useful tip: Before you proceed, make sure that the label is correctly generated by looking at the dataset, located in `data/dataset/`. It should contain each directory for the training set and the testing set, respectively. Moreover, both should contain an ordered series of **raw image and its corresponding segmentation mask label as a pair.**

5.3 Running the Training Loop

In order to run the training loops, you may need to initialize the optimizer with an appropriate learning rate, and complete the training loop for a single batch. After this, we are finally ready to train the model by running:

```
python3 scripts/simple_train.py
```

Fill in the training configuration (e.g. batch size, learning rate, epochs etc.). Tuning the hyper-parameters is essential in getting the most out of the model architecture you have designed. Make sure to experiment with different combinations; we have made sure this is feasible by shortening the training time. You will be asked to **provide details about the set of hyperparameters you have tried and why you chose the one over the others** in the report. You may find training visualization via wandb or screenshots of RViz useful for this question.

Useful tip: Before you run the training loop, check if your torch version supports GPU acceleration:

```
python3
>>> import torch
>>> torch.cuda.is_available()
```

6 Running Lane Detection

6.1 Python Packages

We need to install some Python packages to run the training pipeline:

```
cd src/mp1
pip install -r requirement.txt
```

6.2 ROS2 Environment Setup

To run the lane detection pipeline, you first need to build your ROS2 project. Start by setting up your terminal for ROS2 and Gazebo. We have already put these two commands in the lab computer .bashrc, so you do not need to execute these two on your own.

```
source /opt/ros/humble/setup.bash
source /usr/share/gazebo/setup.bash
```

Now, build the project by running the following command in the **base directory** of the repository. If you encounter an error during the build process, you could execute the two commands above.

```
colcon build --symlink-install
```

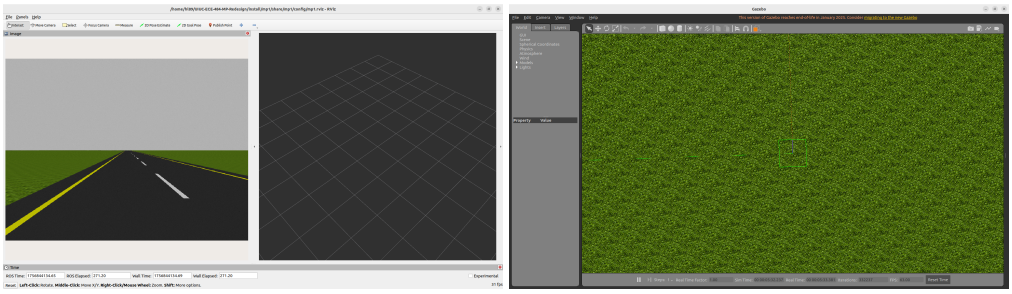
You should see three new directories generated in your base directory: `build/`, `install/`, and `log/`. Set up your terminal again with the build-specific `setup.bash` file, generated in the `install/` directory. **Remember to source these files for every new terminal session you open.**

```
source install/setup.bash
source
```

Now, you are ready to launch your ROS2 environment. Simply run the command:

```
ros2 launch mp1 mp1_launch.launch
```

If successful, you will see the RViz and Gazebo windows open. The image display panel on RViz confirms that ROS has access to the camera stream from GEM.



(a) (b)

Figure 5: (a) RViz start screen (b) Gazebo start screen

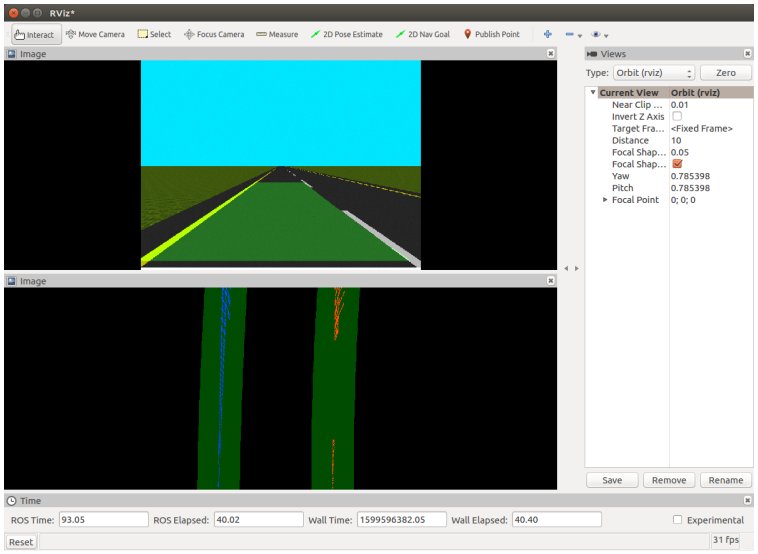
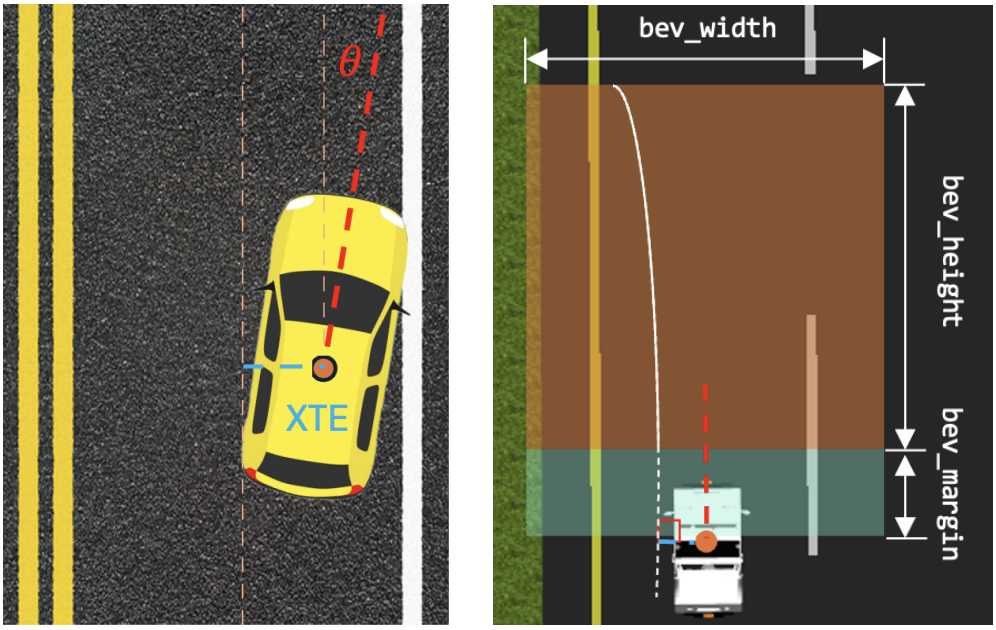


Figure 6: RViz screen of a functional lane detection algorithm

6.3 Evaluation

Now that we have a working lane detection module, let's see how we can quantitatively measure its performance. We will first need to load the checkpoint that performs the best in the validation set. Then, among the diverse metrics for evaluation, we will specifically focus on the *cross-track error* and the *heading error*. As the names suggest, cross-track error indicates the distance of the vehicle from the center of the lane, while the heading error indicates the angular displacement of the direction the vehicle is facing from that of the lane (look at Figure-7(a)).



(a) (b)

Figure 7: (a) Cross-track error (XTE) and heading error (θ) (b) Scaled BEV polynomial (solid) and its extrapolation (dashed)

Simply put, these metrics measure *lateral displacement* and *direction alignment* of our vehicle from the perceived lane (any guess why this might be important in autonomous driving?). The closer the cross-track error and the heading error computed from our prediction are to their ground-truth, the better the lane detection algorithm is!

Implementing these metrics for our prediction involves some post-processing. As Figure-7(b) suggests, we will treat the predicted polynomial as the center of the lane. However, we cannot use it directly because the polynomial is represented in terms of the BEV image pixel coordinates. Luckily, if we know the dimension (px) of the BEV image along with the dimension (m) of its corresponding BEV area, we can easily compute the scaling factor between them! As you might recall, we indeed have already saved these information in our BEV configuration file in Section-4.4. Once we have scaled the polynomial, we compute:

1. **Predicted cross-track error:** The *perpendicular distance* between the vehicle and the polynomial.
2. **Predicted heading error:** The *derivative* of the polynomial evaluated at the same point.

Note that we are not trying to minimize the cross-track and heading errors themselves, but the *difference between the respective errors*, each computed from our prediction and the ground-truth. Run the script to print the errors in your console:

```
python3 scripts/run_error_calculation.py
```

We will provide the validation set that you could adjust the hyperparameters for your Simple ENet training and have a private test set for grading. **We will grade your model on our private test set, but for the demo, you will need to show the cross-track and heading errors.** To run the evaluation of the lane prediction in the console:

```
python3 scripts/eval.py
```

7 Report

Each group should upload a short report with following questions answered.

Problem 6 [Group] (15 points) How do you determine whether a model is overfitting to the training set, and what methods can be used to mitigate it? How do you diagnose underfitting, and what methods can be used to reduce it? Please list two methods, respectively.

Problem 7 [Group] (20 points) What problem might occur if a model is trained mainly on sunny-day images but tested on snowy-day images? List two methods that could improve the model's performance on snowy days. Briefly describe how each method works.

Problem 8 [Group] (30 points) Record a short videos of Rviz window and Gazebo to show that your code works. You can either use screen recording software or smart phone to record. Please ensure that your links to videos are functional before submission.

Problem 9 [Group] (25 points) We will grade your model on our private test set based on the accuracy of lane detection using the checkpoints you provide. You may tune your model with the validation set we provide and demonstrate the effort how you improve model's performance. For example, you can use the training loss curve and loss on the validation set to show your effort. Also, explain how you selected the BATCH_SIZE, Learning Rate (LR), and Number of Epochs when you tune the hyperparameters. Please ensure that you submit the final checkpoints and training loss curve for grading.

Demo (10 points) You will need to demo your solution on both scenarios to the TAs during the lab demo. There may be an autograding component, using an error metric calculation between your solution and a golden solution.

8 Submission instructions

Problems 1-5 must be done individually (Homework 1). Write solutions to each problem in a file named `hw1_<netid>.pdf` and upload the document in Canvas. Include your name and netid in the pdf. You may discuss solutions with others, but do not use written notes from those discussions to write your answers. If you use/read any material outside of those provided for this class to help grapple with the problem, you should cite them explicitly.

Problems 6-9 can be done in groups of 3-4. Figures of **training loss curve** should be submitted in the report. Students need to take a short videos clearly showing that their code works and answer other questions. Your video must show that your implementation can detect lanes when the car is **moving**. The videos can be uploaded to a website (YouTube, Vimeo, or Google Drive with public access) and you need to include the link in the report. One member of each team will upload the report `mp1_<groupname>.pdf` to Canvas. Please include the names and netids of all the group members and cite any external resources you may have used in your solutions. Also, please upload your code (any files that you changed) and **final checkpoint** in a .zip file.

References

[1] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.