ECE 484: Principles of Safe Autonomy	SPRING 2025
Written by: ECE484 team	*HW1, MP1: Lane detection*
	Due Date: 2/21/2025

1 Introduction

In this assignment, you will apply neural networks and computer vision techniques for lane detection. In the Section 2, you will **independently** answer questions related to neural networks and vision. Then in Section 3, you and your **group** will implement a deep learning based lane detection module. Your code should be in the files `train.py` and `test_lane_detection.py`. Your group will also write a brief report `mp1_<groupname>.pdf`, answering questions in Section 3. You will have to use functions from the Pytorch library. This document gives you the first steps to get started on implementing the lane detection module. You will have to take advantage of tutorials and documentations available online. Cite all resources in your report. **All the regulations for academic integrity and plagiarism spelled out in the student code apply.**

Learning objectives

- Gradients for neural network models
- Perspective transforms
- Semantic segmentation mask
- Embedding mask

System requirements

- Ubuntu 20.04
- Mambaforge

2 Homework Problems

Write the proof yourself, and do not copy off of the Internet/peers. Proving these on your own is good practice for quizzes/midterms. An aside, it is typically quite simple to determine when students copy proofs. We are happy to help over Campuswire, during office hours, etc.

Problem 1 [Individual] (10 points) This is a review problem related to safety and verification. The following figure shows four different automata with identical states $Q = \{q_0, \ldots, q_7\}$. $Q_0 = \{q_0\}$ for all the automata. We attach a red or green color to each state, denoted by $q_i.col$, for the ease of writing requirements.

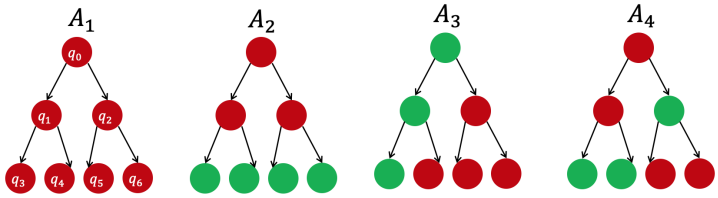
(a) Is $\mathcal{A}_1$ deterministic? Why or why not?

(b) How many executions does $\mathcal{A}_2$ have?

(c) Consider the following 4 requirements:

- Always red: $R_1 = \{\alpha \mid \forall i,\ \alpha_i.col = red\}$.
- Never green: $Unsafe = \{q \mid q.col = green\}$.
- Eventually green: $R_2 = \{\alpha \mid \exists i,\ \alpha_i.col = green\}$.
- Never red: $R_3 = \{\alpha \mid \forall i,\ \alpha_i.col \neq red\}$.

For each of the 4 automata $\mathcal{A}_1, \ldots \mathcal{A}_4$ and each of the 4 requirements, say whether the automaton satisfies the requirement or give a counter-example. You can present the answer in the form of a 4×4 table.



Problem 2 [Individual] (15 points) Consider a neural network with one hidden layer:

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

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation function, where $\mathbf{x} \in \mathbb{R}^{nx1}$.

(a) Derive symbolic expressions for the following gradients:

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

(b) Using the given network parameters:

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

Compute the output $\hat{\mathbf{y}}$ and all gradients derived above. Identify which weight in $W^{(1)}$ has the largest influence on $\hat{\mathbf{y}}$.

(c) Explain how replacing σ with ReLU would affect the gradient magnitudes. Discuss the practical implications for training this network.

Problem 3 [Individual] (25 points) Consider a neural network with two hidden layers:

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

where $\mathbf{f}(\mathbf{z})$ is the custom element-wise activation function defined as:

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

(a) Derive symbolic expressions for the following gradients:

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

(b) Using the given network parameters:

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

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

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

Problem 4 [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 $w = 1$ 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:

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

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

Problem 5 [Individual] (15 points) You are given the pixel coordinates $(\mathbf{x_i}, \mathbf{y_i}) = (400, 300)$ of a point observed by a camera, along with the camera's intrinsic matrix

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

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

Problem 6 [Individual] (15 points)

(a) An autonomous vehicle detects an object through its onboard camera system at a distance of 1000 mm. The camera captures the object's image with a width of 200 pixels and a height of 400 pixels. Using the camera's focal length of 50 mm and a pixel size of 1 mm/pixel, the vehicle computes the magnification $\mathbf{M}$ as the ratio of image size to object size: $M = \frac{f}{Z}$, as 0.05. From this, compute the actual object dimensions ($\mathbf{W_o}$, $\mathbf{H_o}$).

(b) Now while the car was trying to find a parking spot, the width of the object in the image increased to 300 pixels. Using this information, compute the distance of the object.

---

3 Programming Assignment: Lane detection using ENet

3.1 Introduction

The goal of this programming assignment is to detect lanes from images using a neural network (NN) with something called a *dual-head architecture* (we don't expect you to know what this is; we provide details in Sec. 3.2.1). Once the NN is properly trained, the NN will produce 2 outputs for each pixel. Each *head* of the NN produces one output.

The first output is the probability that the pixel belongs to a lane (or to another entity such as a vehicle, a tree, etc.). This is called *segmentation*. The second output will assign a vector value to each pixel which captures a notion of distance is useful for determining whether two pixels belong to the same lane. This second output is called an *embedding*. Intuitively, you can think of an embedding as a vector representation of some other data type, whether that be a string of alphanumeric characters, an audio waveform, or an entire image.

You are going to use a neural network architecture called **ENet**. In this assignment, you will implement various components essential for training and evaluating the model. Specifically, you will load and pre-process the dataset, initialize the optimizer, set up the training loop, implement validation code to assess model performance, and apply a perspective transform to convert the image from the camera view to a bird's-eye view. By the end of this assignment, you will have a fully trained lane detection model and a visualization of detected lanes in both the original and transformed perspectives.

3.2 System Overview

The lane detection system is built around a preconfigured ENet model with a dual-head architecture, where each output serves a distinct purpose in lane detection. Below is an overview of the system pipeline.

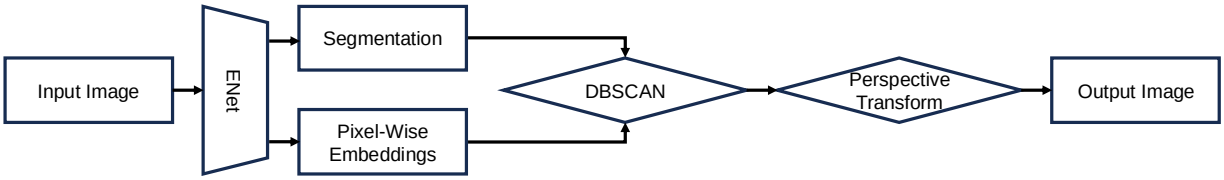


Figure 1: **System Overview.** Given an input image, the ENet model processes the image and generates two outputs: segmentation and pixel-wise embeddings. A clustering algorithm, DBSCAN, is then applied to group pixels and extract lane information from the ENet outputs. Finally, a perspective transformation is performed to convert the image into a bird's-eye view.

3.2.1 Dual-Head Architecture

The ENet model consists of two output heads:

Head 1: Semantic Segmentation Mask This output predicts a mask that classifies each pixel as either part of a lane or background. It provides a high-level lane detection output by labeling lane pixels.

Head 2: Pixel-Wise Embeddings This output assigns a feature vector to each pixel, capturing detailed lane instance information. These embeddings help differentiate between different lane markings by grouping pixels that belong to the same lane.

3.2.2 Loss Function

The model is trained using two types of loss functions: (1) **segmentation loss** for accurate pixel classification, and (2) **discriminative loss** for clustering lane pixels correctly.

Segmentation Loss To ensure accurate lane classification, we use *Cross-Entropy Loss*, a standard function for classification tasks. Cross-Entropy Loss measures how well the model's predicted probability for each pixel matches the actual label. It penalizes incorrect predictions, guiding the model to improve its classification over time. For each pixel in the image:

1. The model predicts a probability between 0 and 1, representing how likely the pixel belongs to a lane.
2. The correct label (ground truth) is either 1 (lane pixel) or 0 (background).
3. Cross-Entropy Loss, $\mathcal{L}_{seg}$, calculates a penalty based on the difference between the predicted probability and the actual label. Mathematically, we can compute this loss as

$$\mathcal{L}_{\text{seg}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

where N is the total number of pixels, y_i is the ground-truth label for the i-th pixel (1 for lane, 0 for background). $\hat{y}_i$ is the model's predicted probability for the i-th pixel.

Discriminative Loss While segmentation loss classifies lane pixels, it does not ensure that pixels belonging to different lanes are well-separated. This is where **lane clustering** comes in. Instead of treating lane pixels independently, we group them into distinct clusters, where each cluster represents a lane. To achieve this, we use **discriminative loss**, which encourages pixels within the same lane to be close together while pushing different lane clusters apart. To aid in your intuition, a visualization of this is included in Fig. 3.2.2.

Let C be the set of lane clusters, where each cluster $c \in C$ contains N_c pixels. The loss function consists of three components:

- **Variance Loss ($\mathcal{L}_{var}$)**: This term ensures that pixels within the same lane remain close to their cluster center, applying an intra-cluster pull force.
- **Distance Loss ($\mathcal{L}_{dist}$)**: This term pushes different lane clusters away from each other to avoid overlap, acting as an inter-cluster repulsion force.
- **Regularization Loss ($\mathcal{L}_{reg}$)**: This term prevents cluster centers from drifting too far from the origin, stabilizing activations.

Each of these terms is mathematically defined as follows:

$$\mathcal{L}_{var} = \frac{1}{|C|}\sum_{c\in C}\frac{1}{N_c}\sum_{i\in c}\max(\|\mathbf{x}_i - \mu_c\| - \delta_v, 0)^2 \tag{1}$$

$$\mathcal{L}_{dist} = \frac{1}{|C|(|C|-1)}\sum_{c_1\neq c_2}\max(2\delta_d - \|\mu_{c_1} - \mu_{c_2}\|, 0)^2 \tag{2}$$

$$\mathcal{L}_{reg} = \frac{1}{|C|}\sum_{c\in C}\|\mu_c\|^2 \tag{3}$$

where $\mathbf{x}_i$ represents a pixel embedding, μ_c is the cluster center for lane c, and δ_v, δ_d are margin parameters that control intra- and inter-cluster distances, respectively.

The total discriminative loss is a weighted sum of all three components:

$$\mathcal{L}_{\text{disc}} = \alpha\cdot\mathcal{L}_{var} + \beta\cdot\mathcal{L}_{dist} + \gamma\cdot\mathcal{L}_{reg} \tag{4}$$

The weighting factors α, β, and γ control the relative importance of these terms.

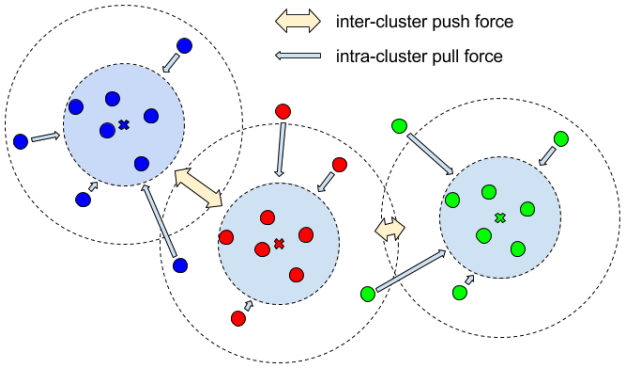


Figure 2: **Visualization of the discriminative loss from the original paper.** The intra-cluster pulling force moves embeddings toward their respective cluster centers, while the inter-cluster repelling force pushes cluster centers apart. These forces are only active within a certain distance, determined by the margins δ_v and δ_d, represented by the dotted circles.

Total Loss The final loss function is computed as the summation of the segmentation loss and the discriminative loss:

$$\mathcal{L}_{\text{total}} = (1 - \lambda)\mathcal{L}_{\text{seg}} + \lambda\mathcal{L}_{\text{disc}} \tag{5}$$

The parameter λ balances the two objectives, allowing the model to jointly optimize for segmentation and lane embeddings.

4 Development Instructions

To begin, clone the repository to your desired location with:

```
$ git clone https://gitlab.engr.illinois.edu/GolfCar/mp-release-sp25.git
```

All the `python` scripts you will need to modify are `train.py` and `test_lane_detection.py` which are located in the following path:

```
$ cd mp-release-sp25/src/mp1
```

The machines in the lab already have the necessary packages, including the TUSimple dataset, installed. **If you are setting up the environment on your own computer, please refer to Section 4.1 for instructions before continuing.** If this is your first time using Weights & Biases (wandb), register your account. Now activate the Conda environment:

```
$ source /opt/miniforge3/bin/activate
```

You will be in the (base) environment of Conda. Now, install the necessary packages from the Conda package manager by the following command. This will also generate another environment called (tusimple) inside your (base) Conda environment.

```
$ mamba env create -f conda_environment.yml
$ conda activate tusimple
```

After the installation has finished, log in to wandb. The prompt will ask you for a user-specific API key generated when you created your account above. You are now all set!

```
$ wandb login
```

4.1 Setting Up the Environment on Your Own Computer

We recommend using Mambaforge (a smaller version of Anaconda) for a faster and more efficient environment setup. Install Mambaforge by following the instructions on the linked github page based on the specifications of your machine. You also need to download the TUSimple dataset to your desired location. Be aware that depending on the locations or configurations of your installations, the exact commands for the lab machines might not work on your machine as well.

4.2 Directory Structure and Dataset

The project follows the directory structure outlined below. **For the lab machines, the TUSimple dataset is stored in the `/opt/data/` directory.** If you are working on your own machine, the path to the dataset can be different.

```
[/mp-release-sp25/src/mp1]

|- /checkpoints           # Generated after checkpointing
|- /datasets              # Code for loading and preprocessing datasets
|- /models                # Model architectures for lane detection
|- /utils                 # Utility functions (e.g., visualization)
|- train.py               # Training the model
|- eval.py                # Evaluating the model
|- test_lane_detection.py # Testing and visualizing lane detection
|- conda_environment.yml     # yml for package installation
```

```
/opt                            # Shared directory
  |- /data
       |- /TUSimple               # TUSimple dataset
```

4.3 Preparing the Dataset and Dataloader

To train and validate the model, you will need to load and preprocess the dataset. Modify the code below to use the `LaneDataset` class from `datasets/lane_dataset.py` along with PyTorch's `DataLoader` for efficient batch processing. Refer to the PyTorch tutorial on data loading for guidance: PyTorch Data Loading Tutorial.

```
# TODO: Load and preprocess the training and validation datasets.
# Hint: Use the LaneDataset class and PyTorch's DataLoader.

################################################################################
# train_dataset = ...
# train_loader = DataLoader(...)

# val_dataset = ...
# val_loader = DataLoader(...)
################################################################################
```

4.4 Initializing the Optimizer

Initialize the Adam optimizer with an appropriate learning rate and weight decay. Refer to the PyTorch documentation on optimizers for details.

```
# TODO: Initialize the Adam optimizer with appropriate learning rate and weight decay.

################################################################################
# optimizer = ...
################################################################################
```

4.5 Writing the Training Loop

Complete the training step for a single batch. Refer to the PyTorch training tutorial for guidance.

```
# TODO: Complete the training step for a single batch.

################################################################################
# Hint:
# 1. Move 'images', 'binary_labels', and 'instance_labels' to the correct device.
# 2. Perform a forward pass using 'enet_model' to get predictions.
# 3. Compute the binary and instance losses using 'compute_loss'.
# 4. Sum the losses ('loss = binary_loss + instance_loss') for backpropagation.
# 5. Zero the optimizer gradients, backpropagate the loss, and take an optimizer step.
################################################################################
```

4.6 Implementing the Validation Code

Perform validation after each epoch using the `validate` function defined in `train.py`.

```
# TODO: Perform validation after each epoch.

################################################################################
# Hint:
# Call the 'validate' function, passing the model and validation data loader.

# val_binary_loss, val_instance_loss, val_total_loss = ...
################################################################################
```

4.7 Perspective transform

In this section, we will implement a function which converts the image to *Bird's eye view* (looking down from the top). This will give a geometric representation of the lanes on the 2D plane and the relative position and heading of the vehicle between those lanes. This view is much more useful for controlling the vehicle, than the first-person view. In the *Bird's eye view*, the distance of pixels on the image will be proportional to the actual distance, thus simplifies calculations in control modules in the future labs. The output is the image converted to *Bird's eye view*.

```
def perspective_transform(image):
    """
    Get bird's eye view from input image
    """

    return transformed_image
```

During the perspective transform we wish to preserve collinearity (i.e., all points lying on a line initially still lie on a line after transformation). The perspective transformation requires a 3-by-3 transformation matrix. Here (x, y) and (u, v) are the coordinates of the same point in the coordinate systems of the original perspective and new perspective.

$$\begin{bmatrix} tu \\ tv \\ t \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6}$$

To find the matrix, we need to find the location of 4 points on the original image and map the same 4 points on the *Bird's Eye View*. Any 3 of those 4 points should not be on the same line. Put those 2 groups of points into cv2.getPerspectiveTransform(), the output will be the transformation matrix. Pass the matrix into cv2.warpPerspective() and we will have the warped *Bird's Eye View* image. Hint: The following website might be useful to find the warp points of the image.

4.8 Visualize lanes

Having detected the lanes from images, now implement `visualize_lanes_row(images, instances_maps, alpha)` that plots multiple images, with the respective lane predictions overlaid on top with the specified alpha value. The images should be converted to their bird's-eye-view using `perspective_transform(image)` function you wrote. Make sure to follow the figure format specified in the starter code. The final result should look like this:

```
def visualize_lanes_row(images, instances_maps, alpha):
    """
    Visualize lane predictions for multiple images in a single row
    """
    plt.tight_layout()
    plt.show()
```

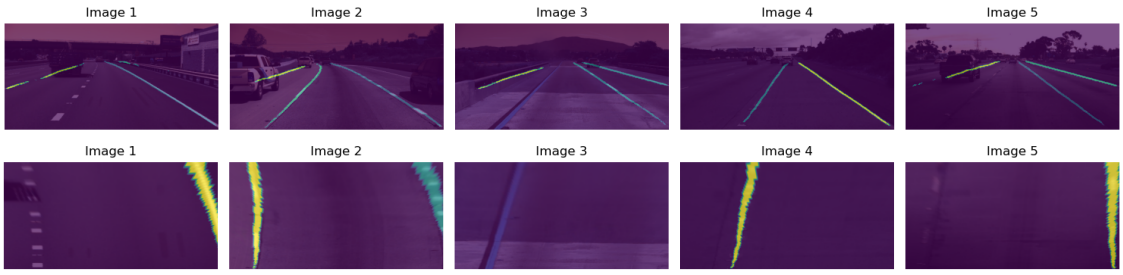


Figure 3: Visualized output without (up) and with (down) the transformation

5 Code Evaluation

5.1 Checkpointing

Save the best model checkpoint in the following location:

```
checkpoints/enet_checkpoint_epoch_best.pth
```

5.2 Evaluation

Evaluate and visualize your model's performance using the provided scripts:

- Run `eval.py` to evaluate the model on the validation dataset.
- Run `test_lane_detection.py` to visualize lane segmentation and clustering results.

6 Report

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

Problem 7 [Group] (30 points) Explain how you selected the following hyperparameters:

- **BATCH_SIZE**: Justify your choice and discuss its impact on training speed and model convergence.
- **Learning Rate (LR)**: How did you determine the appropriate learning rate?
- **Number of Epochs**: How did you decide on the total number of training epochs? What criteria did you use to determine whether training should continue or stop?

How do these parameters influence the training performance and final accuracy of your model?

Problem 8 [Group] (20 points) Provide qualitative and quantitative results of your trained model. Include the following visualizations:

- The original input image.
- The ground truth segmentation mask.
- The predicted segmentation mask.
- The lane embedding visualization, using the first three values of the embedding as RGB channels.

Analyze the performance of your model based on these results.

Problem 9 [Group] (30 points) Conduct an ablation study to analyze the impact of different loss components on model performance. Specifically:

- Train the model with a **high weighting on segmentation loss** ($\mathcal{L}_{\text{seg}}$), while keeping the contribution of discriminative loss minimal. Evaluate its performance on lane detection.
- Train the model with a **high weighting on discriminative loss** ($\mathcal{L}_{\text{disc}}$), reducing the influence of segmentation loss. Analyze its effect on clustering and lane differentiation.
- Compare these results with a model trained using a **balanced loss function** ($\mathcal{L}_{\text{total}}$), where both segmentation and discriminative losses contribute meaningfully.

Discuss how adjusting the relative importance of these loss terms affects the model's ability to:

- Accurately segment lane pixels.
- Differentiate between distinct lane instances.

Problem 10 [Group] (10 points) Visualize and analyze the transformed output from `test_lane_detection.py`.

- Include both the original and transformed outputs in your report.

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.

7 Submission Instructions

7.1 Individual Submission (Homework 1 - Problems 1-6)

- Write your solutions in a PDF file named `hw1_<netid>.pdf` and upload it to Canvas.
- Include your **name and netid** in the document.
- You may discuss the problems with others, but you must write your own answers.
- Cite any external resources you use.

7.2 Group Submission (MP1 - Problems 7-10)

- One member from each group should submit the report as `mp1_<groupname>.pdf` on Canvas.
- Include the **names and netids** of all group members in the report.
- Upload your **code** (`train.py`, `test_lane_detection.py`, and any other relevant files) to a cloud storage service (Google Drive, Dropbox, or Box).
- Provide a **sharable link** to the uploaded code in your report.

7.3 Files to Submit

- **For Homework 1 (Individual Submission):** `hw1_<netid>.pdf`
- **For MP1 (Group Submission):** `mp1_<groupname>.pdf`
- **Code files:** `train.py`, `test_lane_detection.py`, and any additional relevant scripts.
- **Best model checkpoint file.**