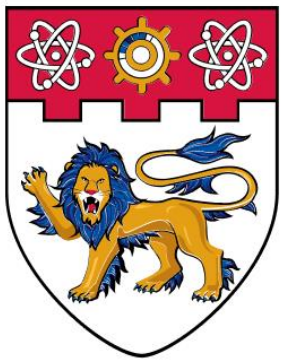
Lane detection algorithm for autonomous vehicles using machine learning

Goh, Terence Wei Liang

2024

Goh, T. W. L. (2024). Lane detection algorithm for autonomous vehicles using machine learning. Final Year Project (FYP), Nanyang Technological University, Singapore. https://hdl.handle.net/10356/177301



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Lane detection algorithm for autonomous vehicles using machine learning

(FYP)

Submitted by: Goh Wei Liang Terence

Matriculation Number: U2021720C

Supervisor: Assoc Professor Lyu Chen
Zhou Yanxin

Abstract

Lane detection is a crucial element of any advanced driver assistance system or autonomous driving technology. Developing a robust lane detection system capable of navigating various road conditions, sucis essential. Traditional techniques that rely on image processing and model fitting have demonstrated proficiency in detecting lanes through distinct features but often falter under suboptimal conditions.

The evolution of machine learning and enhanced computational capabilities have enabled the creation of self-learning algorithms designed to manage the intricate task of extracting and interpreting relevant features for lane identification. However, these models generally require significant computational resources, leading to extended training and prediction times. In order to be on par or better than the conventional methods, a significant amount of training data is needed for the machine learning model to be efficient and accurate.

Hence this report aims on the machine learning model Efficient Neural Network (Enet) to be able to properly perform lane detection accurately and efficiently

Acknowledgement

I am deeply grateful to everyone who supported me in completing this project. My heartfelt thanks go to my supervisor, Assistant Professor Lyu Chen from the School of Mechanical and Aerospace Engineering, for granting me the opportunity to engage with this cutting-edge machine learning project. His commitment to biweekly discussions despite his busy schedule provided me with invaluable guidance and feedback, enhancing my understanding of big data applications in real-world scenarios.

I also extend my gratitude to Zhou Yanxin for his availability and insightful suggestions, which expanded my approach to problem-solving and greatly assisted me in overcoming numerous technical challenges.

Finally, I am thankful to the School of Mechanical and Aerospace Engineering for enabling me to engage in research that goes beyond the curriculum, tackle real-life problems, and enhance my learning through this Final Year Project. This experience has been a significant and fulfilling part of my university journey, for which I am profoundly appreciative.

Contents

1. Introduction

Every year, approximately 1.19 million people lose their lives due to road traffic accidents, and millions more suffer injuries, often leading to disabilities [1]. These incidents not only cause immense personal loss but also lead to significant economic impacts on nations due to treatment costs and lost productivity. With road traffic injuries being the leading cause of death for children and young adults aged 5-29 years, there is an urgent need for comprehensive strategies to improve road safety globally. To reduce these preventable accidents and improve road safety, various

Driver Assistance Systems (DAS) have been designed

Driver assistance systems utilize inputs from various sensors around the vehicle to support the driver, including features like Adaptive Cruise Control, Lane Departure Warning, Lane Detection, and Blind Spot Monitoring[2] . Among these, lane detection is particularly significant as it offers substantial benefits to drivers and forms the basis for many advanced assistance technologies. This report will primarily focus on exploring lane detection systems.

Since lane markings are a visual component, visual sensor data from cameras or LiDARs are usually used for researching lane detection. Lane detection is usually divided into two main categories: image processing and machine learning methods[3]. Image processing usually involves feature extraction first and then model fitting. Feature Extraction classifies imagery where it identifies lane features such as width, colour edges in order to identify the lanes. The next component, Model Fitting, uses a mathematical model to fit on the lane positions in order to predict possible lane curvature.

2. Literature Review

The interest and development of autonomous vehicles have seen a significant increase, driven by advancements in technology and growing industry enthusiasm. In this literature review, we will explore all techniques involved in lane detection to integrate them into our final proposed lane detector.

2.1 Feature Extraction

Feature extraction involves transforming input data into a set of features that can be effectively used for machine learning tasks[5]. This process reduces the dimensionality of data by selecting only the most relevant information, making it easier for algorithms to process and learn from. By doing so, it enhances computational efficiency and accuracy of the learning process.

2.1.1 Lane Boundary

Edges in images can be identified as small regions with an abrupt change in pixel intensity. Since lane markings are distinctive from their background road surface, edges can be extracted using a whole range of gradient-based filters. The procedure is as follows:

Figure1: Gaussian Filter and Box Filter[6]

1. Image Smoothen using box filter or Gaussian filter to reduce noise as seen in Figure 1
2. Performing a specific operation to extract potential edge points
3. Selecting true members from edge point candidates that represent the lane best

Some of the edge detection operations are Sobel and Canny filter. The Sobel filter plays a pivotal role by enhancing edge detection capabilities, specifically identifying lane boundaries by emphasizing areas of high spatial frequency. By applying a pair of 3x3 kernels across the image, it calculates gradients along the horizontal and vertical axes as seen in Figure 2. These gradients

help in accurately delineating the edges as seen in Figure 3, which are crucial for determining the orientation and magnitude of lane lines within the captured images. The Canny edge detection involves a multi-stage process including Gaussian blur, intensity gradient finding, non-maximum suppression, double thresholding, and edge tracking by hysteresis to detect a broad range of edges, effectively mapping out real edges from noise as seen in Figure 4 [12]. Together, these methods contribute significantly to the identification and analysis of edges in images, crucial for interpreting road lanes accurately.

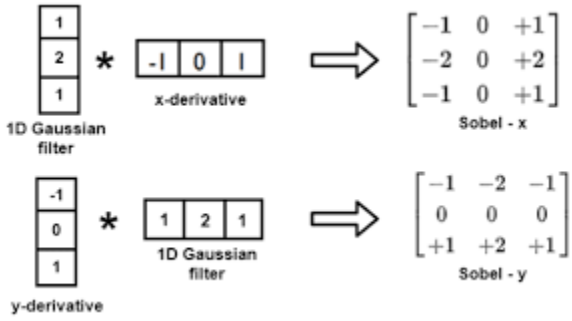


Figure 2: Sobel Filter [8]

Figure 3: Canny Edge Detection [9]

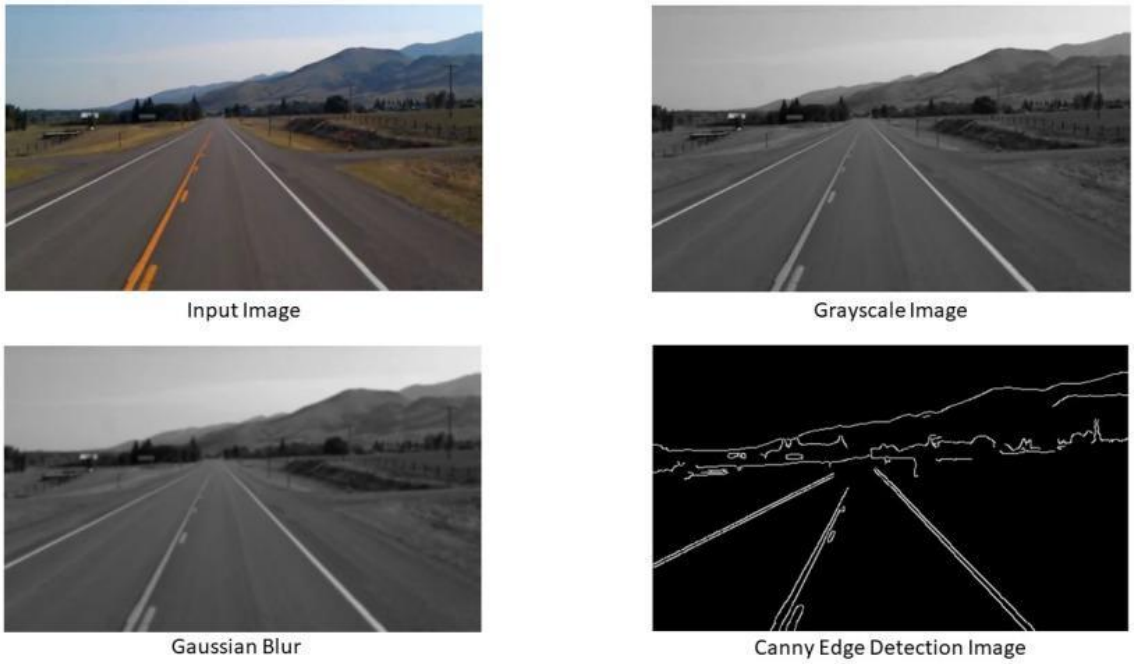


Figure 4 Grayscale and Canny Edge Image[10]

2.1.2 Lane Colour

Every lane marking has a different colour depending on the location or type of road, the edge detection is also been done on colour-based feature detection. Chiu and Lin introduced a method for lane detection using the intensity of grayscale pixels [12]. By transforming images into grayscale and analyzing the pixel intensity distribution in a histogram, they were able to set an adaptive threshold that effectively identifies lane markings. This technique demonstrated enhanced reliability in recognizing lanes that have faded over time. Additionally, Jongin and Hunjae introduce a methodology which involves analyzing pixel intensity within an image to identify lane markers[13]. For white lane markers, a pixel's intensity must rank within the highest 3% of the Y-component's cumulative histogram. Conversely, a yellow lane marker is identified if a pixel's intensity falls within the lowest 1% of the Cb-component's cumulative histogram. Combining binary maps of white and yellow markers through an OR operation produces the final binary map, delineating the lane markings.

2.2 Model Fitting

Once the visual characteristics of lane markers are captured from an image, a model fitting process is initiated to create a simplified, high-level representation of the lane. This process typically involves taking detected points along the lane markers as input.

2.2.1 Hough Transform

First Hough Transform technique is utilized on an array of potential lane marker coordinates to derive the parameters characterizing the identified lane lines. Hough Transform is a technique used to detect straight lines, which can be particularly useful for identifying lanes on a road[14]. It works by transforming each point in the image space into a curve in a parameter space and then finding intersections in this parameter space that indicate the presence of a line with those parameters in the image as seen in Figure 5. Each point in the image space corresponds to a sine wave curve in the Hough space, and the intersecting curves point to the line represented in the image space as seen in Figure 6. This figure illustrates how individual points (in different colors) on a line in the coordinate space correspond to curves in Hough space, and their intersection represents the line's parameters (slope and intercept).

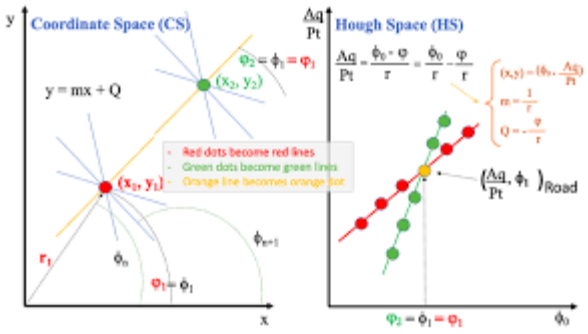


Figure 5 Hough Space [15]

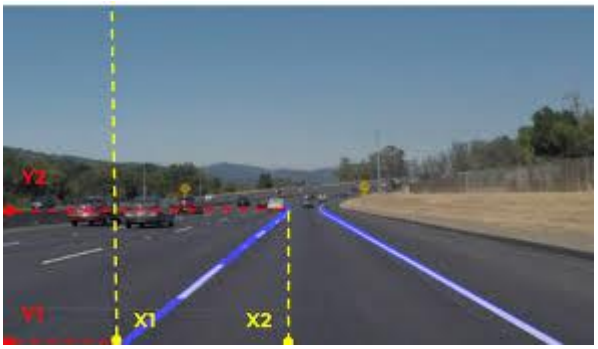


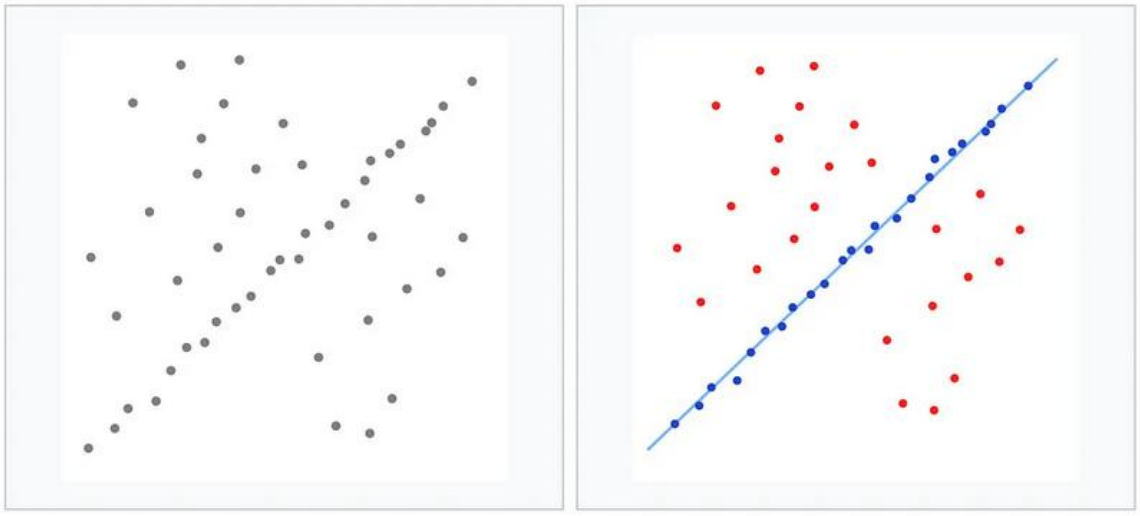
Figure 6: Intersection with lanes [16]

The steps to Hough Transform are as follows:

1. **Hough Space Mapping**: Each detected edge point in the image is projected onto a parameter space to explore potential lines that might pass through it. Each spot in this space represents a candidate line in the original scene.
2. **Accumulator Array**: Within the Hough space, we employ an accumulator to tally the convergence of these potential lines. The points with higher counts suggest a greater likelihood of a true line's presence.
3. **Threshold Application**: We then apply a predefined threshold to the accumulator's counts. Points surpassing this threshold are indicative of a line's presence and are thus confirmed as lines in the image.

2.2.2 RANSAC

While the Hough Transform can detect various shapes, including circles and curves, it’s not the most efficient method for complex shapes. Instead, using spline models combined with the RANSAC algorithm offers a more effective solution for identifying such features. RANSAC, or Random Sample Consensus, is a robust statistical method used in data analysis to identify and mitigate outliers within a dataset. It works by repeatedly choosing a random subset of the original data to fit a model and then determining how many other data points fit this model within a predefined tolerance level as seen in Figure 7[17].



A data set with many outliers for which a line has to be fitted.

Fitted line with RANSAC; outliers have no influence on the result.

Figure 7: RANSAC gradient [17]

To utilize RANSAC, certain key parameters must be determined:

- The smallest number of points necessary to define the model.
- The number of algorithmic repetitions.
- A cut-off value to distinguish outliers from inliers.
- The least count of inliers needed for the model to be acceptable.

The procedure is as such:

1. From the dataset, a minimal subset is chosen randomly to infer the model parameters.
2. Data points are then classified as inliers or outliers based on a predefined distance metric.
3. Should inliers exceed a certain count, refine the model with these points and conclude the process.
4. Iterate the previous steps, each time considering the model with the most inliers as the best fit.

RANSAC offers a foundational approach to detecting lanes by discerning patterns and discarding anomalies, but it doesn't always cope well with the variability of real-world conditions like

different road surfaces, weather changes, or varying light conditions. Machine learning, on the other hand, brings a sophisticated layer of analysis, with the capacity to learn from diverse scenarios and improve predictive accuracy. This makes it a vital tool for robust lane detection, able to adjust to a wide range of conditions and ensure reliable performance in a dynamic driving environment. In the next section we talk about how machine learning is vital to lane detection.

2.3 Machine learning

Machine learning is a branch of artificial intelligence that involves the use of data and algorithms to imitate the way humans learn, incrementally improving its accuracy [18]. It's about constructing systems and methodologies that can learn from data to make decisions with minimal human intervention. This technology is the driving force behind many modern conveniences, including web search results, real-time ads, and network intrusion detection systems, to name a few. It empowers software applications to become more accurate in predicting outcomes without being explicitly programmed to do so.

2.3.1 Types of Machine Learning

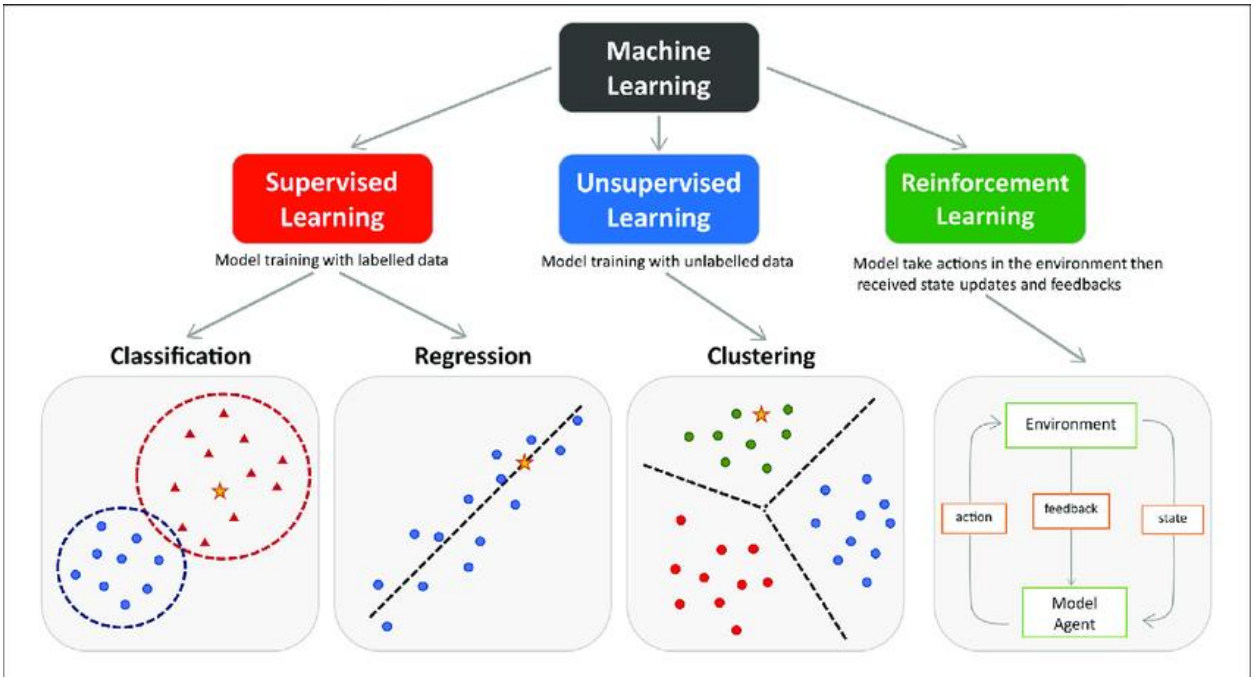


Figure 8: Types of Machine Learning [19]

Machine Learning are categorized into 3 different types as seen in Figure 8:

1. Supervised Learning

Supervised learning is a type of machine learning where models are trained using labeled data. In this approach, the model is provided with input data along with the corresponding output labels. The goal is for the model to learn the mapping from inputs to outputs, enabling it to make predictions on new, unseen data [18]. Supervised learning is widely used for applications such as classification, where input data is categorized into labels, and regression, where the model predicts a continuous output.

2. Unsupervised Learning

Unsupervised learning involves training models on data without labels, allowing the model to identify patterns and structures within the data itself. This method is useful for clustering, where the model groups similar data points together, and for dimensionality reduction, which simplifies data to make it easier to analyze [18]. Unsupervised learning helps uncover hidden patterns in data when we don't have predefined categories or outcomes.

3. Reinforcement Learning

Reinforcement learning is a type of machine learning where an agent learns to make decisions by taking actions in an environment to achieve some goals [20]. It learns from the outcomes of its actions, rather than from being taught explicitly. The state represents the current situation of the agent, the action is what the agent decides to do, the reward is the feedback from the environment based on the action's effectiveness, and the environment is the world in which the agent operates as seen in Figure 9 [21]. Over time, the agent learns to make better decisions that lead to more rewards.

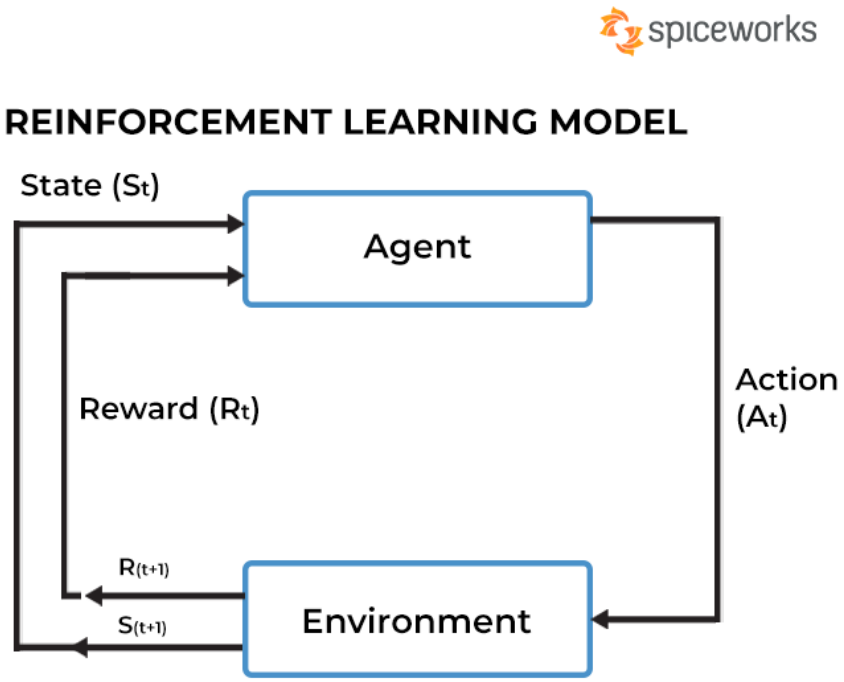


Figure 9: Reinforcement Learning

2.3.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a class of deep learning models primarily used for processing data with grid-like topology, such as images. A CNN would process an input image through multiple layers designed to detect various features, from simple edges to complex objects like lane markings. [23]. CNN uses a series of layers to process input images and identify lanes as seen in Figure 10:

1. **Input Layer**: Receives the raw pixel data from the road images.
2. **Convolutional Layers**: These layers use filters to detect lane line features, like edges and curves, from the input image.
3. **Pooling Layers**: These reduce the dimensionality of the data by downsampling, making the detection of features more robust to noise and variations.

4. **Fully Connected Layers**: These layers act as a classifier on top of the features extracted from the convolutional and pooling layers to determine the presence and position of lanes.
5. **Output Layer**: Produces the final result, indicating detected lane lines which can be used to guide autonomous vehicles or assist drivers.

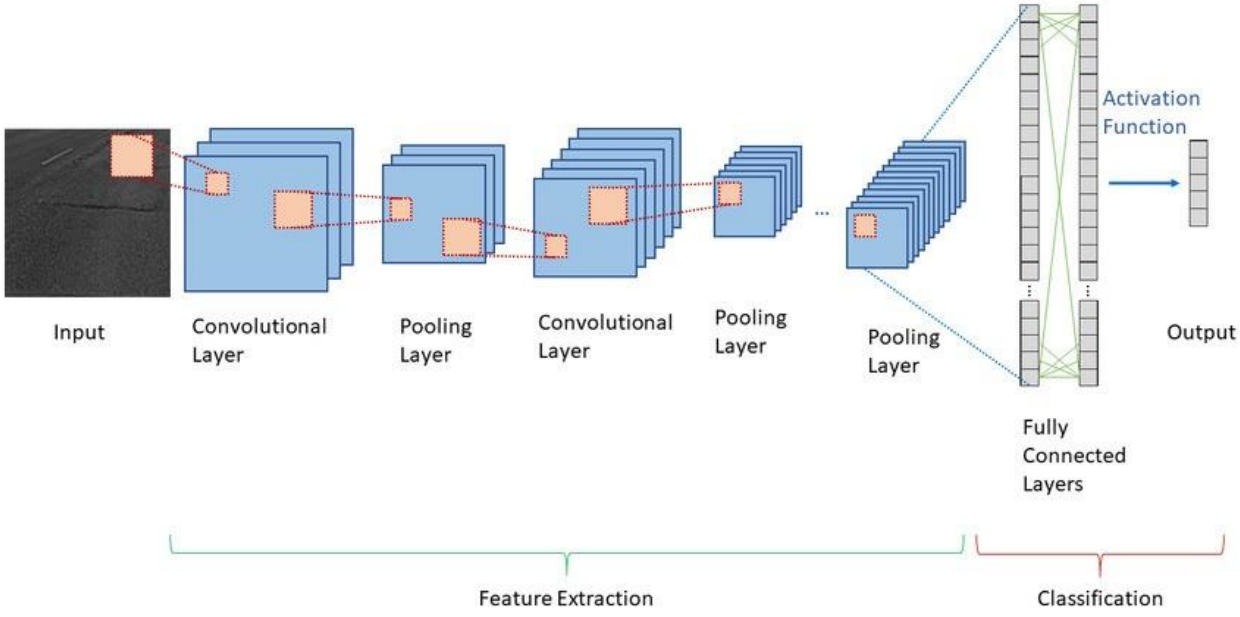


Figure10: Convolution Neural Network Flowchart [22]

2.3.2.1 Convolutional Layer

Convolutional layers are the core building blocks of a CNN where it handles the main bulk of computation. In a convolutional layer, small grids called kernels slide across the input image to process overlapping areas simultaneously. Each kernel, although smaller in width and height compared to the input image, extends through the full depth of the input's channels such as RGB. It performs element-wise multiplication with the input's local region it covers and aggregates the results into a feature map, which helps the model understand various features within the image for tasks such as identifying lanes on a road[24].

In convolutional neural networks, a kernel, also known as a filter, moves over an image's width and height, processing sections to create a feature map that shows the kernel's responses at various

image positions. There are 3 main components that control the size of the output volume of the Convolutional Layer:

1. Depth: The depth of a convolutional layer's output, a key design choice, corresponds to the quantity of filters utilized [25]. Each filter is trained to detect distinct features within the input. These various filters within a single convolutional layer are specialized to identify different aspects of a road, such as lane boundaries, road edges, or specific lane markings based on their orientation and color.
2. Stride: Stride refers to the number of pixels by which the filter kernel moves across the image. A stride of 1 moves the filter one pixel at a time, while a larger stride reduces the spatial dimensions of the output volume, as the kernel skips over pixels of the input image [26]. Using a smaller stride in convolutional operations results in overlapping fields of view across the image, which tends to enlarge the dimensionality of the output data. In contrast, a larger stride reduces this overlap and diminishes the output size. We use a 5x5 input image on the left along with a Lapacian kernel on the right image as seen in Figure 11:

95	242	186	152	39
39	14	220	153	180
5	247	212	54	46
46	77	133	110	74
156	35	74	93	116

0	1	0
1	-4	1
0	1	0

Figure 11: 5x5 input image with Lapacian kernel [26]

When the stride parameter is set to 1, the filter methodically traverses the entire image one pixel at a time both horizontally and vertically, ensuring every pixel is covered, which creates an expansive output as seen in the left image on Figure(). Contrastingly, a stride set to 2 causes the filter to move two pixels over with each step, effectively skipping every other pixel as seen in the right image on Figure 12. This action results in a reduced size of the resulting output because fewer

positions are calculated. This stride manipulation is a common technique in image processing to adjust the granularity of feature detection.

692	-315	-6
-680	-194	305
153	-59	-86

692	-6
153	-86

Figure 12: *Left:* Output of convolution with 1×1 stride. *Right:* Output of convolution with 2×2 stride.

3. Zero-Padding: Zero padding in convolutional neural networks refers to the practice of adding layers of zeros around the input image before applying the convolution operation. This is done to control the spatial size of the output feature maps, allowing the network to preserve the spatial dimensions of the input through the layers.

2.3.2.2 Pooling Layer

Pooling layers are used within CNN to reduce the spatial dimensions of the input volume for the next convolution layer. It is a form of non-linear down-sampling [26].

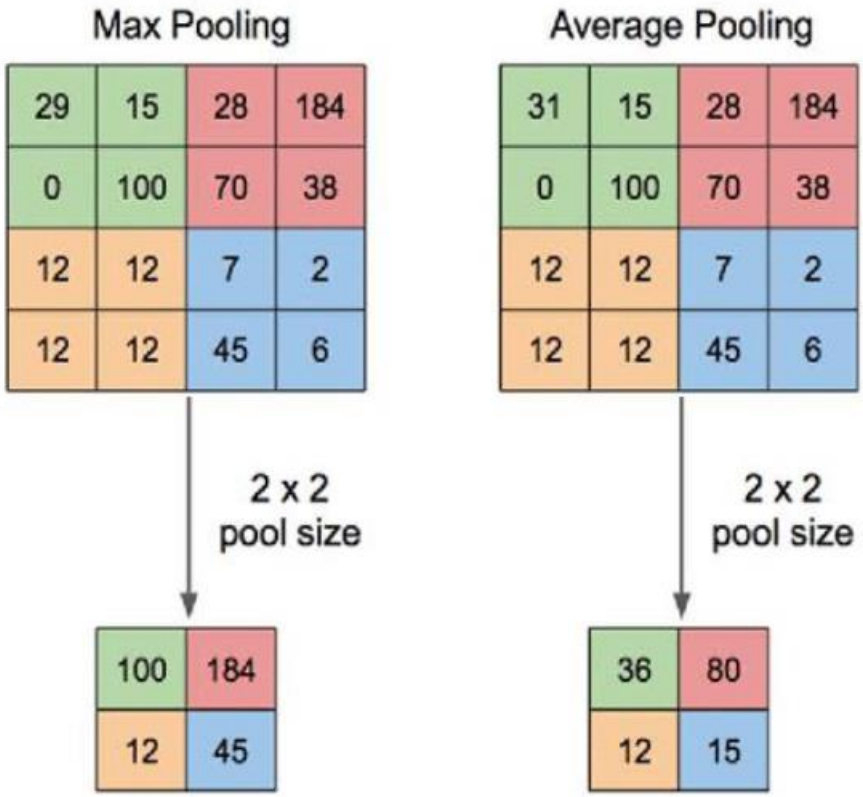


Figure 13: Max Pooling and Average Pooling [27]

There are two main types of Pooling as seen in Figure 13:

1. Max Pooling:

   Max pooling selects the highest value from a specified area of the feature map, covered by its window or kernel. In max pooling, the pooling layer divides the input into a set of non-overlapping rectangles and outputs the maximum value for each subregion. For the 2x2 pool size in Figure 13, we look at each 2x2 block in the input matrix and take the largest value. This operation reduces the dimensions of the input by half if the stride is equal to the pool size.

   In the provided example, the top-left 2x2 block (with values 29, 15, 0, 100) gets reduced to a single value of 100 because 100 is the maximum number within that block.

   The process repeats for other blocks, ultimately resulting in a 2x2 output from the original 4x4 input. The mathematical formula of max pooling is:

$$\text{MaxPooling}(X)_{i,j,k} = \max_{m,n} X_{i \cdot s_x + m, j \cdot s_y + n, k}$$

Figure 14: Formula of Max Pooling [31]

2. Average Pooling:

   Average pooling calculates the mean value of the data points within the area of the feature map that the filter spans. Average pooling also divides the input into non-overlapping rectangles, but instead of taking the maximum value, it computes the average of the values in each subregion. With the same 2x2 pool size, it calculates the average for each block. For the top-left block in the example, the average is (29+15+0+100)/4 = 36. This downsampling process also results in a 2x2 output from the 4x4 input, with each value representing the average of a corresponding block. The mathematical formula of average pooling is:

$$\text{AvgPooling}(X)_{i,j,k} = \frac{1}{f_x \cdot f_y} \sum_{m,n} X_{i \cdot s_x + m, j \cdot s_y + n, k}$$

Figure 15: Formula of Average pooling [31]

2.3.2.3 Fully Connected Layers

Fully connected layers in a neural network are where all neurons from the previous layer connect to each neuron in the current layer [29]. These layers are critical for learning complex patterns by combining features from the previous layers. In essence, fully connected layers integrate learned features for the final classification or regression tasks. This layer is a combination of Affine function and Non-Linear function.

The Fully Connected layer operates on a flattened input where each input dimension is considered. The process begins by applying an affine transformation, followed by a non-linear transformation through functions like Affine and Non-Linear, comprising a single Fully Connected layer as seen in Figure 16 [30]. Multiple layers can be stacked to deepen the model, contingent on the complexity needed for the dataset. The output from the last layer undergoes a Softmax or Sigmoid function to derive a probability distribution across the possible classes.

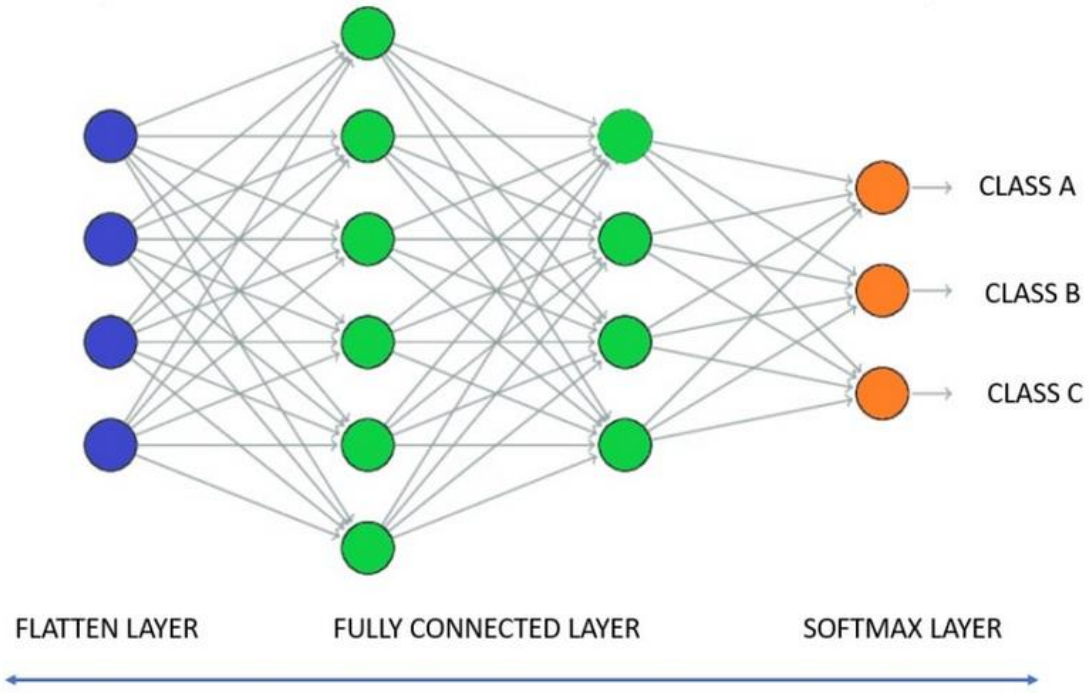


Figure 16: Flatten Layer to Fully Connected Layer

3. Methodology

3.1 Implementation Environment

3.1.1 Software

This project uses mainly Python 3.9.6 as the main environment system using Conda. Within Python these are the list of major libraries that I used:

Libraries	**Description**
Pytorch	PyTorch is an open-source framework for machine learning, developed with Python and based on the Torch library. Originally created for deep neural network research, Torch, which PyTorch draws from, is coded in Lua.
NumPy	NumPy is a core Python library for scientific computing, offering a powerful multidimensional array object and tools for a wide range of fast array operations. NumPy is mainly use for array and matrices manipulation
MatPlotLib	Matplotlib is a versatile library for creating static, interactive, and animated visualizations in Python.
OpenCV	OpenCV is an open-source library focused on computer vision and machine learning. Its creation aimed to provide a common infrastructure to accelerate the incorporation of machine perception capabilities in commercial products through computer vision applications.
JSON (JavaScript Object Notation)	JSON is a format for exchanging data that is designed to be both human-readable and machine-friendly. In this project, it was

	utilized for the task of parsing JSON strings, facilitating smooth data handling and processing.

3.1.2 Hardware

For the hardware component of my final year project, I utilized my laptop equipped with an RTX 1650 graphics card. This powerful piece of hardware was instrumental in handling the computationally intensive tasks required by my project, such as running machine learning models and processing large datasets. The RTX 1650's robust performance capabilities enabled efficient experimentation and development, significantly contributing to the project's success by offering a blend of speed and accuracy in data processing and model training.

3.2 Pipeline

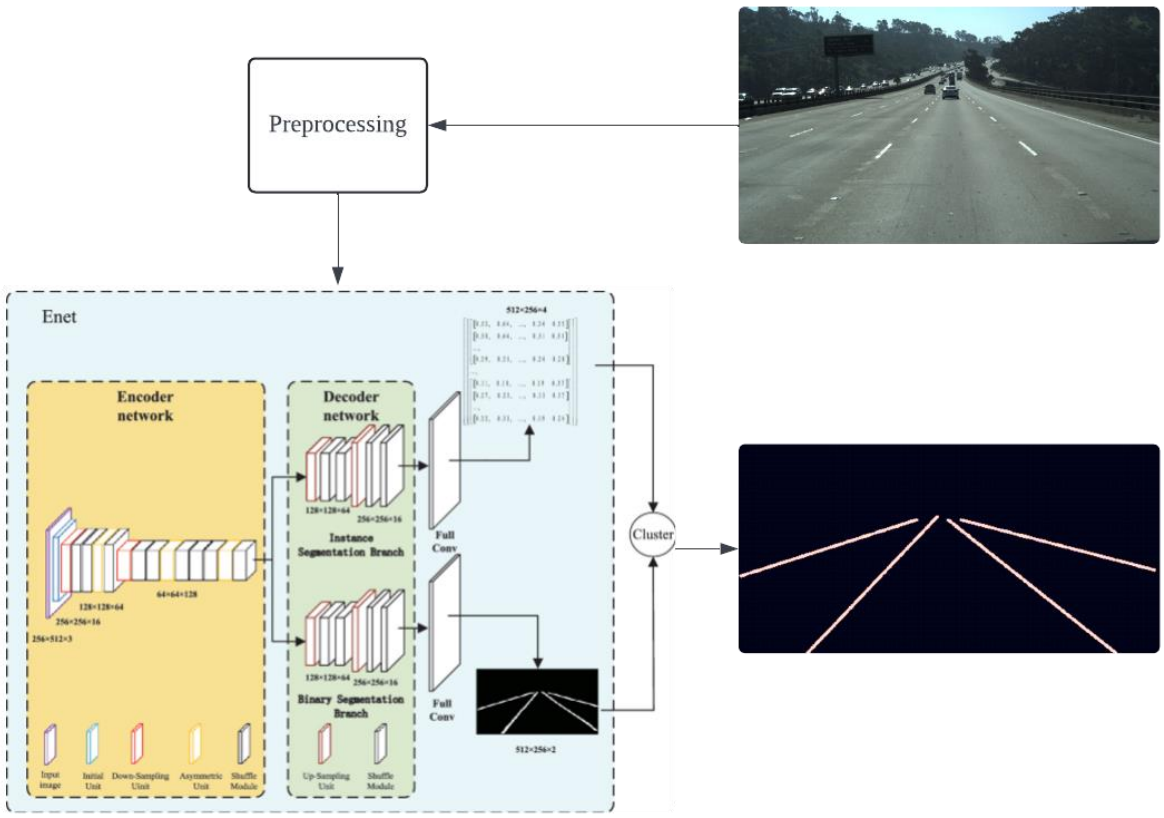


Figure 17: Pipeline of lane detection

Figure 17 shows the pipeline of this lane detection project proposed in this paper. This pipeline has different stages where images are first pre-processed to prepare them for analysis, the images are then trained under ENet machine learning model which will be decoded through the instance segmentation branch and binary segmentation branch.

3.2.1 Labelling lane detection images

For the creation of a comprehensive dataset for training lane detection algorithms, each image needs to be labeled with the coordinates of lane markings. These coordinates are critical as they serve as ground truth for supervised learning, enabling the model to learn the visual patterns associated with lanes on various roadways. I use JSON to label each of the different image markings as JSON provides a lightweight data interchange format that is easy for humans to read and write, as well as easy for machines to parse and generate. JSON allows data to be organized and structured in a clear and logical way.

3.2.1.1 Labeling Detail


```
{"lanes": [[-2, -2, -2, -2, -2, -2, -2, -2, -2, -2, 648, 636, 626, 615, 605, 595, 585, 575, 565, 554, 545, 536, 526, 517, 508, 498, 489, 480, 470, 461, 452, 442, 433,
424, 414, 405, 396, 386, 377, 368, 359, 349, 340, 331, 321, 312, 303, 293, 284, 275, 265, 256, 247, 237, 228, 219], [-2, -2, -2, -2, -2, -2, -2, -2, -2, -2, 681, 692,
 704, 716, 728, 741, 754, 768, 781, 794, 807, 820, 834, 847, 860, 873, 886, 900, 913, 926, 939, 952, 966, 979, 992, 1005, 1018, 1032, 1045, 1058, 1071, 1084, 1098,
 1111, 1124, 1137, 1150, 1164, 1177, 1190, 1203, 1216, 1230, 1243, 1256, 1269], [-2, -2, -2, -2, -2, -2, -2, -2, -2, -2, 713, 746, 778, 811, 845, 880, 916, 951, 986,
 1022, 1057, 1092, 1128, 1163, 1198, 1234, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2],
 [-2, -2, -2, -2, -2, -2, -2, -2, -2, -2, 754, 806, 858, 909, 961, 1013, 1064, 1114, 1164, 1213, 1263, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2,
 -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2]], "h_samples": [160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260, 270, 280, 290, 300,
310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 410, 420, 430, 440, 450, 460, 470, 480, 490, 500, 510, 520, 530, 540, 550, 560, 570, 580, 590, 600, 610, 620, 630,
640, 650, 660, 670, 680, 690, 700, 710], "raw_file": "test_set/clips/0530/1492626760788443246_0/20.jpg"}
```


Figure 18: JSON Labelling of lane details

Figure 18 shows an example of the labelling process that involves marking the positions of lane lines in the images in a JSON format. Here's an overview of the labeling structure based on the sample JSON:

1. "lanes": An array of arrays, each representing a single lane. Within these sub-arrays, the y-coordinates of the lane markings are listed at fixed x-coordinates defined in the "h_samples" array. The value -2 is used as a placeholder when a lane marking is not present or detectable at that particular horizontal position.

2. "h_samples": This array contains the x-coordinates corresponding to the vertical positions in the image where the lanes are sampled. These coordinates are consistent across all images in the dataset, allowing for a standardized comparison and evaluation across different images.

3. "raw_file": Specifies the path to the image file within the dataset directory. This enables direct correlation between the labeled data and the image it pertains to.

The labeled coordinates can be visualized on the image by plotting points or drawing lines at the specified y-coordinates for each of the **"h_samples"** x-coordinates. This visualization aids in verifying the accuracy of the labeling and provides a clear indication of the algorithm's performance during training and validation phases.

3.2.2 Image PreProcessing

Before training the images into machine learning, the images in TUSimple are pre-processed first. Figure 19 provides the main Pre-Processing code for our TUSimple images where 4 main stages will be explained:

1. Data acquisiton and resizing
2. Dimension Adjustment
3. Image Segmentation
4. Tensor Conversion and Permutation

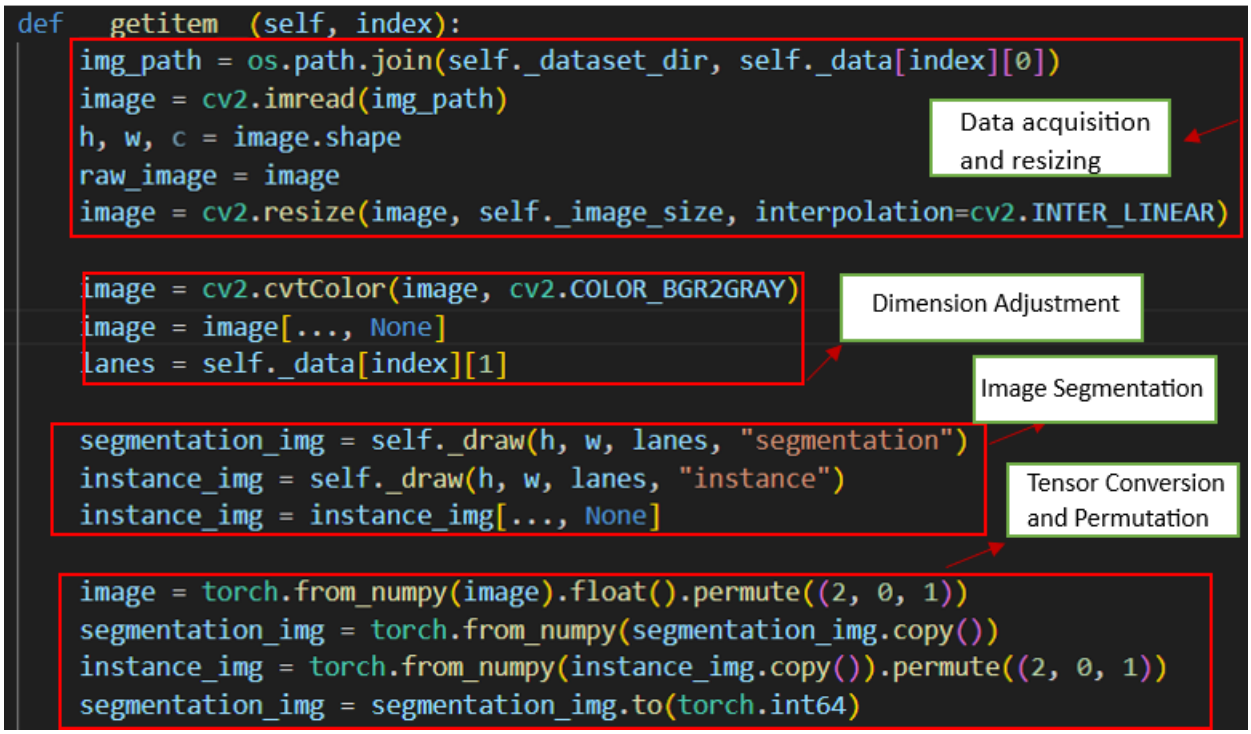


Figure 19: Image Pre-Processing Code

3.2.2.1 Data Acquisition and resizing

```
img_path = os.path.join(self._dataset_dir, self._data[index][0])
image = cv2.imread(img_path)
h, w, c = image.shape
raw_image = image
image = cv2.resize(image, self._image_size, interpolation=cv2.INTER_LINEAR)
```

Figure 20: Data Acquisition and resizing Code

The preprocessing begins with the retrieval of an image from the dataset based on the provided index in Figure 20. The image is located within the dataset directory, with its path constructed using **os.path.join**. Once the image is loaded using OpenCV's **cv2.imread**, it is resized to the predetermined dimensions specified by **self._image_size**. This ensures uniformity in input size, a common requirement for deep learning models where consistent image dimensions are necessary for batch processing.

3.2.2.2 Dimension Adjustment for Neural Network Compatibility

```
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
image = image[..., None]
lanes = self._data[index][1]
```

Figure 21: Dimension adjustment code

First the image is converted to grayscale. This step simplifies the images by reducing it from three color channels (Red, Green, Blue) to a single channel, and at the same time reduces computational complexity. This single-channel grayscale image is then expanded with an additional singleton dimension at the end to comply with the input convection '[height, widith, channels]' which is the required format by Pytorch. Figure 21 shows the dimension adjustment code.

3.2.2.3 Image Segmentation Preparation

```
segmentation_img = self._draw(h, w, lanes, "segmentation")
instance_img = self._draw(h, w, lanes, "instance")
instance_img = instance_img[..., None]
```

Figure 22: Image Segmentation Code

To prepare the target outputs for the model, the segmentation and instance images are generated using the **_draw** function. This function creates a mask where the lanes are drawn onto blank images, segmenting them from the rest of the image. In the case of the segmentation image, lanes

are marked with a single color, while for the instance image, each lane is marked with a unique color to distinguish between different lanes. Figure 22 shows the Image Segmentation Code

3.2.2.4 Tensor Conversion and Permutation

```
image = torch.from_numpy(image).float().permute((2, 0, 1))
segmentation_img = torch.from_numpy(segmentation_img.copy())
instance_img = torch.from_numpy(instance_img.copy()).permute((2, 0, 1))
segmentation_img = segmentation_img.to(torch.int64)
```

Figure 23: Tensor Conversion and Permutation Code

Finally, the grayscale image and the segmentation masks are converted from NumPy arrays to PyTorch tensors. This conversion is necessary because PyTorch models expect input data to be in tensor format. Additionally, the dimensions of the tensors are permuted to match the **[channels, height, width]** format. The segmentation image is also cast to a **torch.int64** tensor, which is typically required for classification tasks where the target is expected to be of a discrete nature. Figure 23 shows the Tensor Conversion and Permutation Code

3.3 Model Construction

The Machine Learning Model use is mainly Efficient Neural Network(Enet) model is specifically designed for tasks requiring efficient processing and fast execution, such as real-time segmentation in autonomous vehicles or mobile applications. It is especially efficient on images so hence Enet was chosen. Figure 24 shows how the overall Enet architecture is like.

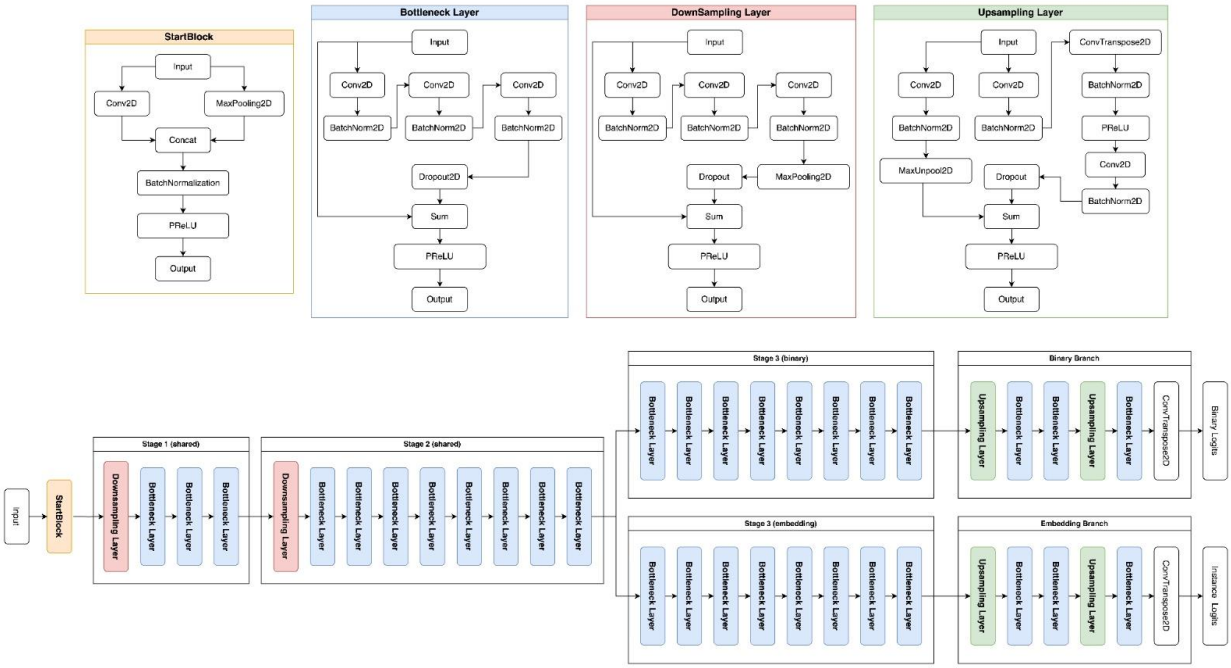


Figure 24: Pipeline of ENet architecture

3.3.1 Components of Enet

First I will explain on the 4 layers of ENet: StartBlock, Bottleneck, DownSampling and Upsampling Layer before going through how these layers play a part in the architecture sequence.

3.3.1.1 StartBlock

```python
class StartBlock(nn.Module):
    def __init__(self,
                 input_channels,
                 output_channels,
                 use_bias=False,
                 use_relu=True):
        super().__init__()

        if use_relu:
            activation_func = nn.ReLU
        else:
            activation_func = nn.PReLU

        self.main_conv = nn.Conv2d(
            input_channels,
            output_channels - 1,
            kernel_size=3,
            stride=2,
            padding=1,
            bias=use_bias)
        self.ext_pool = nn.MaxPool2d(3, stride=2, padding=1)
        self.batch_norm_layer = nn.BatchNorm2d(output_channels)
        self.output_activation = activation_func()

    def forward(self, input_tensor):
        main_output = self.main_conv(input_tensor)
        ext_output = self.ext_pool(input_tensor)
        combined_output = torch.cat((main_output, ext_output), 1)
        normalized_output = self.batch_norm_layer(combined_output)
        return self.output_activation(normalized_output)
```

Figure 25: StartBlock code

The StartBlock serves as the entry point of the Efficient Neural Network (ENet) architecture. Its primary function is to perform an initial transformation on the input data, preparing it for the subsequent layers of the network. This block is carefully designed to immediately reduce the spatial dimensions of the input, which aids in decreasing the computational load for the following operations. Figure 25 shows the StartBlock code

Initial Parameters: The block takes in **input_channels**, representing the number of channels in the input tensor, and **output_channels**, indicating the desired number of channels in the output tensor. The **use_bias** parameter allows toggling the use of bias in convolutional operations, and **use_relu** chooses between two types of activation functions, ReLU for simple non-linearity and PReLU for a learnable non-linear function.

Convolution and Pooling:

- The block employs two parallel paths:

- A **main_conv** convolutional layer with a kernel size of 3x3, a stride of 2 for downsampling, and padding to preserve spatial dimensions. This layer produces **output_channels - 1** feature maps, reducing the input's spatial size while extracting features.
- An **ext_pool** max pooling layer with identical stride and padding settings, further downsampling the input tensor and providing robustness to input variations.

Combination and Normalization:

The outputs of both paths are concatenated along the channel dimension to merge feature representations. This combination leverages the detailed features extracted by the convolution and the abstracted features from the pooling. A batch normalization layer, **batch_norm_layer**, then normalizes this combined output, which helps in accelerating the training process and stabilizing the network.

3.3.1.2 Bottleneck Layer

The Bottleneck Layer is a pivotal component of the ENet architecture, designed for semantic segmentation tasks. This layer aims to balance computational efficiency with the capacity to process and refine feature representations extracted from my input data by manipulating the depth of the feature maps through a series of convolutions with a bottleneck design—reducing the channels, processing them, and then restoring the original depth. Figure 26, 27 and 28 shows the Bottleneck Layer code.

```
class BottleneckLayer(nn.Module):
    def __init__(self,
                 dilation=1,
                 asymmetric=False,
                 dropout_prob=0,
                 use_bias=False,
                 use_relu=True):
        super().__init__()

        if ratio <= 1 or ratio > input_channels:
            raise RuntimeError("Invalid ratio value. Ratio should be in the range (1, {0}]. Got: {1}"
                               .format(input_channels, ratio))

        internal_channels = input_channels // ratio

        if use_relu:
            activation = nn.ReLU
        else:
            activation = nn.PReLU

        self.conv1 = nn.Sequential(
            nn.Conv2d(
                input_channels,
                internal_channels,
                kernel_size=1,
                stride=1,
                bias=use_bias), nn.BatchNorm2d(internal_channels), activation())
```

Figure 26: Bottleneck layer code part 1

```
    if asymmetric:
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                internal_channels,
                internal_channels,
                kernel_size=(kernel_size, 1),
                stride=1,
                padding=(padding, 0),
                dilation=dilation,
                bias=use_bias), nn.BatchNorm2d(internal_channels), activation(),
            nn.Conv2d(
                internal_channels,
                internal_channels,
                kernel_size=(1, kernel_size),
                stride=1,
                padding=(0, padding),
                dilation=dilation,
                bias=use_bias), nn.BatchNorm2d(internal_channels), activation())
    else:
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                internal_channels,
                internal_channels,
                kernel_size=kernel_size,
                stride=1,
                padding=padding,
                dilation=dilation,
                bias=use_bias), nn.BatchNorm2d(internal_channels), activation())
```

Figure 27: Bottleneck layer code part 2

```
        self.conv3 = nn.Sequential(
            nn.Conv2d(
                internal_channels,
                input_channels,
                kernel_size=1,
                stride=1,
                bias=use_bias), nn.BatchNorm2d(input_channels), activation())

        self.dropout = nn.Dropout2d(p=dropout_prob)

        self.final_activation = activation()

    def forward(self, x):
        main_output = x
        ext_output = self.conv1(x)
        ext_output = self.conv2(ext_output)
        ext_output = self.conv3(ext_output)
        ext_output = self.dropout(ext_output)
        final_output = main_output + ext_output
        return self.final_activation(final_output)
```

Figure 28: Bottleneck layer code part 3

The Bottleneck Layer operates on input feature maps and employs a series of operations:

1. **Dimensionality Reduction**: The initial **conv1** layer uses a 1x1 convolution to reduce the number of input channels by a factor defined by **ratio**. This step concentrates the information and reduces computation for the subsequent layers.

2. **Feature Processing**: Depending on the **asymmetric** flag, **conv2** applies either a set of asymmetric convolutions or a regular convolution with the provided **kernel_size**, **padding**, and **dilation**. Asymmetric convolutions break down a larger convolutional operation into two smaller ones, reducing the number of parameters and computational complexity.

3. **Dimensionality Restoration**: A subsequent 1x1 convolution in **conv3** brings the number of channels back to the original depth. This step reconstructs the feature maps to their initial dimensions, ready for addition to the input.

4. **Regularization**: **Dropout2d** is applied to prevent overfitting by randomly setting a portion of the feature maps to zero during training, controlled by **dropout_prob**. Figure 29 shows how dropout reduces number of paths

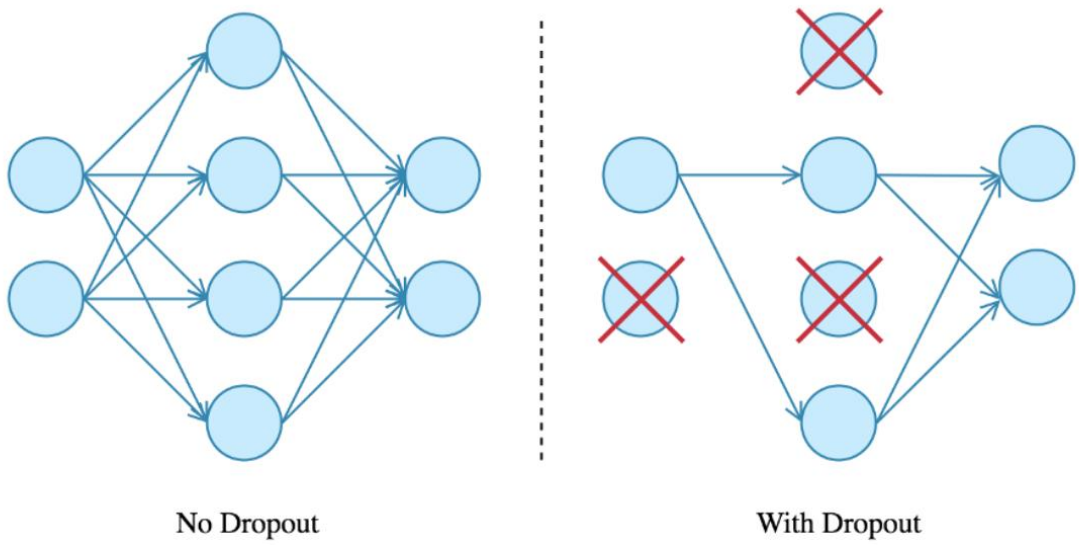


Figure 29:Dropout effect [32]

5. **Normalization and Combining Features**: The processed feature maps are then added back to the input tensor, creating a residual connection that helps mitigate the vanishing gradient problem and promotes feature reuse. Batch Normalization:

$$y = \gamma \left(\frac{x - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}} \right) + \beta$$

where μ_{batch} is the mean of the batch, σ^2_{batch} is the variance, γ is a scale factor, β is an offset, and ϵ is a small constant added for numerical stability.

Figure 30: Batch Normalization Equation

6. **Activation**: Finally, a non-linear activation function, either ReLU or PReLU depending on **use_relu**, is applied to introduce non-linearity into the model, allowing it to learn more complex features.

3.3.1.3 Downsampling Layer

The Downsampling Layer in ENet is an essential component engineered for reducing the spatial dimensions of the input feature maps. This reduction is pivotal in expanding the receptive field of

the network and decreasing the computational load for the subsequent layers, enabling more efficient processing. Figure 31, 32 and 33 shows the Downsampling Layer overall code

```
class DownsamplingLayer(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 internal_ratio=4,
                 return_indices=False,
                 dropout_prob=0,
                 use_bias=False,
                 use_relu=True):
        super().__init__()

        self.return_indices = return_indices

        if internal_ratio <= 1 or internal_ratio > in_channels:
            raise RuntimeError("Value out of range. Expected value in the "
                               "interval [1, {0}], got internal_scale={1}. "
                               .format(in_channels, internal_ratio))

        internal_channels = in_channels // internal_ratio

        if use_relu:
            activation = nn.ReLU
        else:
            activation = nn.PReLU

        self.max_pooling = nn.MaxPool2d(
            2,
            stride=2,
            return_indices=return_indices)
```

Figure 31: Downsampling Layer Code Part 1

```
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels,
                internal_channels,
                kernel_size=2,
                stride=2,
                bias=use_bias), nn.BatchNorm2d(internal_channels), activation())

        self.conv2 = nn.Sequential(
            nn.Conv2d(
                internal_channels,
                internal_channels,
                kernel_size=3,
                stride=1,
                padding=1,
                bias=use_bias), nn.BatchNorm2d(internal_channels), activation())

        self.conv3 = nn.Sequential(
            nn.Conv2d(
                internal_channels,
                out_channels,
                kernel_size=1,
                stride=1,
                bias=use_bias), nn.BatchNorm2d(out_channels), activation())

        self.dropout = nn.Dropout2d(p=dropout_prob)

        self.activation_out = activation()
```

Figure 32: Downsampling Layer Code Part 2

```
def forward(self, x):
    if self.return_indices:
        main, max_indices = self.max_pooling(x)
    else:
        main = self.max_pooling(x)

    ext = self.conv1(x)
    ext = self.conv2(ext)
    ext = self.conv3(ext)
    ext = self.dropout(ext)

    n, ch_ext, h, w = ext.size()
    ch_main = main.size()[1]
    padding = torch.zeros(n, ch_ext - ch_main, h, w)

    if main.is_cuda:
        padding = padding.cuda()

    main = torch.cat((main, padding), 1)
    out = main + ext
    return self.activation_out(out), max_indices if self.return_indices else None
```

Figure 33: Downsampling Layer Code Part 3

3.3.1.3.1 Initialization and Key Components

Upon initialization, the Downsampling Layer is configured with parameters that determine its structure and behavior:

- **in_channels**: The number of input channels.
- **out_channels**: The number of output channels after downsampling.
- **internal_ratio**: Determines the degree of channel reduction before the convolution operations.
- **return_indices**: A flag indicating whether to return the max pooling indices, which is useful for later upsampling operations.
- **dropout_prob**: The probability of an element to be zeroed, introducing regularization.
- **use_bias**: Whether the convolution layers will include a bias term.
- **use_relu**: Decides between using the ReLU or PReLU activation function.

The layer consists of:

1. Max Pooling:

A max pooling operation with a kernel size of 2 and stride of 2, halving the dimensions of the input tensor. If return_indices is True, the indices of the max values are stored for later use in upsampling.

2. Sequential Convolutions:

- conv1: Reduces the number of channels according to the internal_ratio using a 2x2 convolution with stride 2.
- conv2: A 3x3 convolution with padding maintains the spatial dimensions while processing the internal channels.
- conv3: A 1x1 convolution increases the number of channels to out_channels.
- Mathematical Formula of the 3 conv:

$$Y_{ij} = \sum_{m,n,c} (W_{mn} \cdot X_{(i+m),(j+n),c} + b)$$

where W is the convolutional kernel, b is the bias term, X is the input tensor, and c iterates over the channels.

Figure 34: Convolutional mathematical formula

3. Batch Normalization and Activation:

- Each convolutional step is followed by batch normalization and an activation function, standardizing outputs and introducing non-linearity.
- Batch Normalization:

$$Y = \gamma \left(\frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta$$

where X is the input to the batch normalization, μ is the mean of X, σ^2 is the variance, γ is a scale parameter, β is a shift parameter, and ϵ is a small constant for numerical stability.

Figure 35: Batch Normalization formula

4. Dropout:

- Applied for regularization purposes to help prevent overfitting during training.

3.3.1.3.2 Forward Pass Process

During the forward pass, the layer executes the following operations:

1. The input **x** goes through max pooling, which reduces its spatial dimensions and optionally returns indices for max values.

2. The input **x** also passes through **conv1**, **conv2**, and **conv3** in sequence, reducing its resolution and adjusting channel depth while maintaining spatial dimensions.
3. Dropout is applied to the output of **conv3**.
4. Padding is added to the max-pooled feature maps to match the channel dimensions of the convolved output.
5. The padded max-pooled feature maps are combined with the convolved output, and the final activation function is applied.
6. The layer outputs the downsampled feature maps, with an option to include max pooling indices if **return_indices** is **True**.

3.3.1.4 UpSampling Layer

The Upsampling Layer in ENet is designed to increase the spatial dimensions of the feature maps. Its primary function is to reverse the effect of previous downsampling operations, a critical process in tasks such as semantic segmentation where the final output needs to be of the same resolution as the input. Figure 36 and 37 shows the overall UpSampling Layer code.

```
class UpsamplingLayer(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 internal_ratio=4,
                 dropout_prob=0,
                 use_bias=False,
                 use_relu=True):
        super().__init__()

        if internal_ratio <= 1 or internal_ratio > in_channels:
            raise RuntimeError("Value out of range. Expected value in the "
                               "interval [1, {0}], got internal_scale={1}. "
                               .format(in_channels, internal_ratio))

        internal_channels = in_channels // internal_ratio

        if use_relu:
            activation = nn.ReLU
        else:
            activation = nn.PReLU

        self.main_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=use_bias),
            nn.BatchNorm2d(out_channels))

        self.main_unpool = nn.MaxUnpool2d(kernel_size=2)

        self.ext_conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels, internal_channels, kernel_size=1, bias=use_bias),
            nn.BatchNorm2d(internal_channels), activation())
```

Figure 36: Upsampling Layer Code Part 1

```
        self.ext_tconv1 = nn.ConvTranspose2d(
            internal_channels,
            internal_channels,
            kernel_size=2,
            stride=2,
            bias=use_bias)
        self.ext_tconv1_bnorm = nn.BatchNorm2d(internal_channels)
        self.ext_tconv1_activation = activation()

        self.ext_conv2 = nn.Sequential(
            nn.Conv2d(
                internal_channels, out_channels, kernel_size=1, bias=use_bias),
            nn.BatchNorm2d(out_channels), activation())

        self.ext_dropout = nn.Dropout2d(p=dropout_prob)

        self.out_activation = activation()

    def forward(self, input_tensor, max_indices, output_size):
        main_out = self.main_conv(input_tensor)
        main_out = self.main_unpool(
            main_out, max_indices, output_size=output_size)

        ext_out = self.ext_conv1(input_tensor)
        ext_out = self.ext_tconv1(ext_out, output_size=output_size)
        ext_out = self.ext_tconv1_bnorm(ext_out)
        ext_out = self.ext_tconv1_activation(ext_out)
        ext_out = self.ext_conv2(ext_out)
        ext_out = self.ext_dropout(ext_out)

        final_out = main_out + ext_out

        return self.out_activation(final_out)
```

Figure 37: Upsampling Layer Code Part 2

3.3.1.4.1 Construction and Initialization

Upon instantiation, the Upsampling Layer sets up several components, with key parameters to govern its behavior:

- **in_channels** and **out_channels** define the number of input and output channels.
- **internal_ratio** controls the compression and expansion within the layer.
- **dropout_prob** provides regularization.
- **use_bias** determines whether biases are used in convolutional layers.

- **use_relu** selects between ReLU and PReLU activation functions.

3.3.1.4.2 Layer Components

1. **Main Convolution**: Applies a 1x1 convolution (**main_conv**) to refine the channel dimensions from **in_channels** to **out_channels**, followed by batch normalization.

   Convolution is a mathematical operation that involves a kernel or filter W sliding over the input X to produce an output Y through an element-wise multiplication and addition: $Yi,j =\sum m,nWm,n\cdot Xi+m,j+n+b$

2. **Unpooling Operation**: Uses **main_unpool**, which is a MaxUnpool2d layer that performs the inverse operation of max pooling using the recorded indices from the corresponding downsampling step, to restore the spatial dimensions of the feature maps.

3. **Expansion Path**:

   - **ext_conv1**: A 1x1 convolution reduces the depth to **internal_channels**.
   - **ext_tconv1**: A transposed convolution (also known as deconvolution) with a kernel size of 2 and a stride of 2 increases the spatial dimensions.
   - **ext_tconv1_bnorm** and **ext_tconv1_activation**: Apply batch normalization and activation, respectively, to the output of the transposed convolution.

4. **Output Path**:

   - **ext_conv2**: Another 1x1 convolution adjusts the channel dimensions back to **out_channels**, with subsequent batch normalization and activation.
   - **ext_dropout**: Implements dropout for regularization.

3.3.1.4.3 Forward Pass

- The input tensor is processed through **main_conv** and **main_unpool** to generate **main_out**, upsampled and channel-adjusted.
- Simultaneously, the input tensor goes through the expansion path (**ext_conv1**, **ext_tconv1**, **ext_tconv1_bnorm**, **ext_tconv1_activation**) to generate **ext_out**, also upsampled.
- The **ext_out** is further processed by **ext_conv2** and **ext_dropout**.

- The outputs of the main and expansion paths (**main_out** and **ext_out**) are then combined through element-wise addition to fuse the upsampled features.
- The fused output is activated by **out_activation** to produce the final output tensor of the layer.

3.3.2 Comprehensive Analysis of ENet's Architecture and Processing Flow

This section delves into the nuances of ENet's architecture, highlighting the orchestrated flow of data through its initial blocks, shared stages, binary, and embedding branches, culminating in a detailed description of its forward propagation mechanism.

3.3.2.1 Stage 1: Shared Encoding

```
# Stage 1 share
self.downsample1_0 = DownsamplingLayer(16, 64, return_indices=True, dropout_prob=0.01, use_relu=encoder_relu)
self.regular1_1 = BottleneckLayer(64, padding=1, dropout_prob=0.01, use_relu=encoder_relu)
self.regular1_2 = BottleneckLayer(64, padding=1, dropout_prob=0.01, use_relu=encoder_relu)
self.regular1_3 = BottleneckLayer(64, padding=1, dropout_prob=0.01, use_relu=encoder_relu)
self.regular1_4 = BottleneckLayer(64, padding=1, dropout_prob=0.01, use_relu=encoder_relu)
```

Figure 38: Shared Encoding Code

The ENet architecture model begins with stage 1 shared encoding where it serves as a critical phase for condensing the spatial dimensions of the input while simultaneously enriching its feature representation.

From Figure 38, The stage begins with a DownSampling Layer which halves the spatial dimensions of the input by moving from a height and width of 'H x W' to 'H/2 x W/2' and at the same time increasing the depth of the feature maps from 16 to 64 channels. This is achieved through a combination of max pooling and parallel convolution operation.

Following the initial DownSampling, a series of four BottleNeck Layers are employed. Each of these layers executes three steps:

1. Compression: The feature maps are first compressed using a 1x1 convolution, reducing the depth of the channels to a fraction which lessons the computational load for subsequent operations

2. Feature Transformation: The compressed feature maps are processed by a 3x3 convolution which is crucial to learning the spatial hierarchies of the features within the reduced feature map dimensions
3. Expansion and Regularization: The transformed features are expanded back to the original depth with another 1x1 convolution

3.3.2.2: Stage 2: Shared encoding

```
# Stage 2 share
self.downsample2_0 = DownsamplingLayer(64, 128, return_indices=True, dropout_prob=0.1, use_relu=encoder_relu)
self.regular2_1 = BottleneckLayer(128, padding=1, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated2_2 = BottleneckLayer(128, dilation=2, padding=2, dropout_prob=0.1, use_relu=encoder_relu)
self.asymmetric2_3 = BottleneckLayer(128, kernel_size=5, padding=2, asymmetric=True, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated2_4 = BottleneckLayer(128, dilation=4, padding=4, dropout_prob=0.1, use_relu=encoder_relu)
self.regular2_5 = BottleneckLayer(128, padding=1, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated2_6 = BottleneckLayer(128, dilation=8, padding=8, dropout_prob=0.1, use_relu=encoder_relu)
self.asymmetric2_7 = BottleneckLayer(128, kernel_size=5, asymmetric=True, padding=2, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated2_8 = BottleneckLayer(128, dilation=16, padding=16, dropout_prob=0.1, use_relu=encoder_relu)
```

Figure 39: Shared encoding code part 1

```
# Stage 2 share
stage2_input_size = x.size()
x, max_indices2_0 = self.downsample2_0(x)
x = self.regular2_1(x)
x = self.dilated2_2(x)
x = self.asymmetric2_3(x)
x = self.dilated2_4(x)
x = self.regular2_5(x)
x = self.dilated2_6(x)
x = self.asymmetric2_7(x)
x = self.dilated2_8(x)
```

Figure 40: Shared encoding code part 2

Stage 2 delves deeper into the refinement and expansion of the feature representation. Building on the compressed and enriched features from Stage 1, this phase further abstracts the input data through increased channel depth and diversified convolutional techniques, targeting more complex patterns and broader contextual information. Figure 39 and 40 shows the shared encoding code.

This stage is introduced with a DownSampling Layer (self.downsample2_0) which further increase the feature depth from 64 to 128 channels while reducing the spatial dimensions. A sequence of eight BottleNeck Layers follows to explore the feature space with specific convolutional strategies:

1. ‘self.regular2_1’: Utilize standard convolution to consolidate and refine features within the new depth space

2. ‘self.dilated2_2’ to self.dilated2_8’: Employs dilated convolutions with progressively increasing dilation rates(2,4,8,16), allowing the network to grasp larger contextual scopes without the expense of additional parameters
3. ‘self.asymmetric2_3’ and ‘self.asymmetric2_7’: Applies asymmetric convolutions, breaking down the conventional square kernel into a two-step process involving rectangular kernels horizontally first then vertically, optimizing parameter efficiency.

3.3.2.3 Stage 3: Binary and Embedding

After Stage 2, it is split into two process, the binary process and the embedding process.

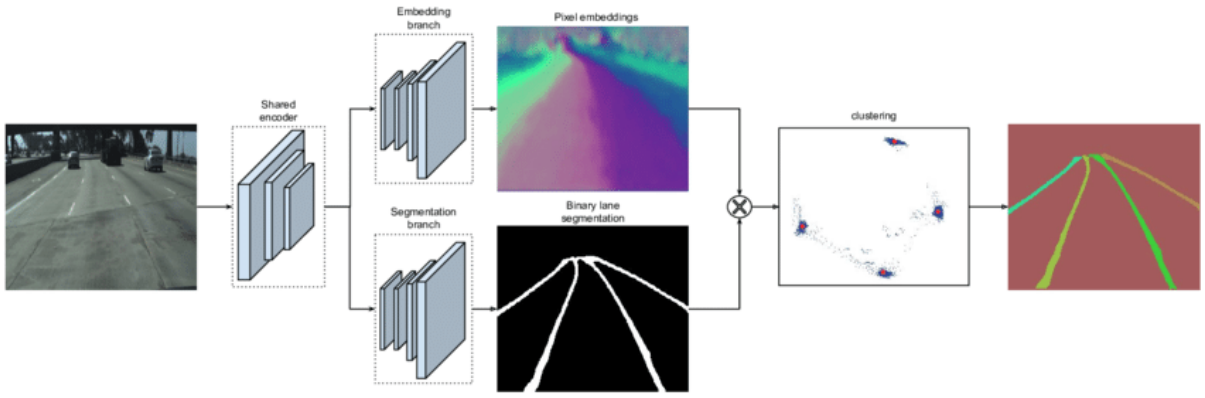


Figure 41: Binary and Embedding Process Images [33]

3.3.2.3.1 Binary Processing

```
# binary branch
self.upsample_binary_4_0 = UpsamplingLayer(128, 64, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_binary_4_1 = BottleneckLayer(64, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_binary_4_2 = BottleneckLayer(64, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.upsample_binary_5_0 = UpsamplingLayer(64, 16, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_binary_5_1 = BottleneckLayer(16, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.binary_transposed_conv = nn.ConvTranspose2d(16, binary_seg, kernel_size=3, stride=2, padding=1, bias=False)
```

Figure 42: Binary Processing Code Part 1

```
# binary branch
x_binary = self.upsample_binary_4_0(x_binary, max_indices2_0, output_size=stage2_input_size)
x_binary = self.regular_binary_4_1(x_binary)
x_binary = self.regular_binary_4_2(x_binary)
x_binary = self.upsample_binary_5_0(x_binary, max_indices1_0, output_size=stage1_input_size)
x_binary = self.regular_binary_5_1(x_binary)
binary_final_logits = self.binary_transposed_conv(x_binary, output_size=input_size)
```

Figure 43: Binary Processing Code Part 2

Stage 3 Binary Processing is dedicated to refining the segmentation masks for binary classification tasks, such as separating road from non-road in driving scenes. Figure 42 and 43 shows the overall Binary Processing Code

Layer Configuration:

- **Regular and Dilated Bottleneck Layers:** These layers form the backbone of Stage 3 binary processing, where regular bottleneck layers maintain feature resolution and dilated bottleneck layers expand the receptive field without increasing the number of parameters significantly. This is crucial for capturing broader contextual information without loss of detail, which is essential for accurate segmentation.
- **Asymmetric Bottleneck Layers:** These are specialized layers designed to efficiently process features with asymmetric fields of view, which helps in handling orientations and shapes more effectively, crucial for detailed areas of the input image.
- **Dropout:** Applied consistently across the layers to mitigate overfitting, enhancing the model's ability to generalize well to unseen data.

Flow Through the Layers:

1. **Initial Processing:** The feature map from the previous stage is first processed by a regular bottleneck layer to stabilize the features before more complex transformations.
2. **Expansion of Receptive Field:** Following the initial processing, a series of dilated bottleneck layers progressively increase the receptive field to capture wider contextual information without increasing the spatial dimensions of the feature maps.
3. **Asymmetric Processing:** Interspersed with dilated layers, asymmetric bottlenecks handle features with complex orientations, enhancing the model's ability to discern irregular patterns and shapes.
4. **Final Combination:** The last dilated layer in the sequence further expands the field to encompass the broadest context within the feature map, ensuring that the subsequent upsampling layers have rich, detailed information to refine.

3.3.2.3.2 Embedding Process

```
# stage 3 embedding
self.regular_embedding_3_0 = BottleneckLayer(128, padding=1, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated_embedding_3_1 = BottleneckLayer(128, dilation=2, padding=2, dropout_prob=0.1, use_relu=encoder_relu)
self.asymmetric_embedding_3_2 = BottleneckLayer(128, kernel_size=5, padding=2, asymmetric=True, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated_embedding_3_3 = BottleneckLayer(128, dilation=4, padding=4, dropout_prob=0.1, use_relu=encoder_relu)
self.regular_embedding_3_4 = BottleneckLayer(128, padding=1, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated_embedding_3_5 = BottleneckLayer(128, dilation=8, padding=8, dropout_prob=0.1, use_relu=encoder_relu)
self.asymmetric_bembedding_3_6 = BottleneckLayer(128, kernel_size=5, asymmetric=True, padding=2, dropout_prob=0.1, use_relu=encoder_relu)
self.dilated_embedding_3_7 = BottleneckLayer(128, dilation=16, padding=16, dropout_prob=0.1, use_relu=encoder_relu)
```

Figure 44: Embedding Process Code Part 1

```
# stage 3 embedding
x_embedding = self.regular_embedding_3_0(x)
x_embedding = self.dilated_embedding_3_1(x_embedding)
x_embedding = self.asymmetric_embedding_3_2(x_embedding)
x_embedding = self.dilated_embedding_3_3(x_embedding)
x_embedding = self.regular_embedding_3_4(x_embedding)
x_embedding = self.dilated_embedding_3_5(x_embedding)
x_embedding = self.asymmetric_bembedding_3_6(x_embedding)
x_embedding = self.dilated_embedding_3_7(x_embedding)
```

Figure 45: Embedding Process Code Part 2

Stage 3 Embedding Processing is similar to the Binary processing stated above but focuses on generating dense pixel-level embeddings. Figure 44 and 45 shows the Embedding Process COde

Layer Configuration:

- The configuration is similar to the binary processing stage, utilizing a mix of regular, dilated, and asymmetric bottleneck layers.
- **Dilation Rates and Asymmetric Convolutions:** These are employed to effectively capture multi-scale and directionally varied features, which are crucial for distinguishing between closely positioned objects in the scene.

Flow Through the Layers:

1. **Embedding Initialization:** Starting with regular bottlenecks to stabilize initial embeddings.
2. **Contextual Embedding Expansion:** Through dilated and asymmetric layers, embeddings are enriched with broad contextual data, crucial for instance segmentation.

3. **Final Detail Enhancement:** The highest dilation rates in the final layers ensure that even the most minute details are captured in the embeddings, critical for accurate instance **separation.**

3.3.2.4 *Binary Branch Post-Processing*

```
# binary branch
self.upsample_binary_4_0 = UpsamplingLayer(128, 64, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_binary_4_1 = BottleneckLayer(64, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_binary_4_2 = BottleneckLayer(64, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.upsample_binary_5_0 = UpsamplingLayer(64, 16, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_binary_5_1 = BottleneckLayer(16, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.binary_transposed_conv = nn.ConvTranspose2d(16, binary_seg, kernel_size=3, stride=2, padding=1, bias=False)
```

Figure 46: Binary Branch Post-Processing Code Part 1

```
# binary branch
x_binary = self.upsample_binary_4_0(x_binary, max_indices2_0, output_size=stage2_input_size)
x_binary = self.regular_binary_4_1(x_binary)
x_binary = self.regular_binary_4_2(x_binary)
x_binary = self.upsample_binary_5_0(x_binary, max_indices1_0, output_size=stage1_input_size)
x_binary = self.regular_binary_5_1(x_binary)
binary_final_logits = self.binary_transposed_conv(x_binary, output_size=input_size)
```

Figure 47: Binary Branch Post-Processing Code Part 2

Following the processing in Stage 3 for binary segmentation, the binary branch of ENet focuses on generating a high-resolution binary segmentation output that delineates specific features or areas of interest, such as road surfaces or obstacles in autonomous driving applications. This branch refines and upsamples the encoded features from earlier network stages to produce finely detailed segmentation maps. Figure 46 and 47 shows the Binary Branch Post-Processing Codes

First an Upsampling Layer is applied which increase the map resolution from 128 channels to 64 channels to 64 channels. Following upsampling, two consecutive BottleNeck Layers are applied to enhance details and uses standard convolution and batch normalization to ensure the quality of feature maps. Another Upsampling Layer is then applied which transitions the feature maps from 64 to 16 channels to approach the original dimensions of the input image. A final BottleNeck Layer is then applied to ensure that the subtle features are enhanced and captured. The last layer in the binary branch ‘self.binary_transposed_conv’ is a transposed convolutional layer that projects the 16-channel feature maps back to the image resolution, producing the final binary segmentation map. This layer uses a stride of 2 and appropriate padding to effectively double the spatial dimensions, achieving an output size that matches the original input. The logits output from this layer represents the probability of each pixel belonging to a particular class.

3.3.2.5 *Embedding Branch Processing*

```
# embedding branch
x_embedding = self.upsample_embedding_4_0(x_embedding, max_indices2_0, output_size=stage2_input_size)
x_embedding = self.regular_embedding_4_1(x_embedding)
x_embedding = self.regular_embedding_4_2(x_embedding)
x_embedding = self.upsample_embedding_5_0(x_embedding, max_indices1_0, output_size=stage1_input_size)
x_embedding = self.regular_embedding_5_1(x_embedding)
instance_final_logits = self.embedding_transposed_conv(x_embedding, output_size=input_size)
```

Figure 48: Embedding Branch Processing Code Part 1

```
# embedding branch
self.upsample_embedding_4_0 = UpsamplingLayer(128, 64, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_embedding_4_1 = BottleneckLayer(64, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_embedding_4_2 = BottleneckLayer(64, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.upsample_embedding_5_0 = UpsamplingLayer(64, 16, dropout_prob=0.1, use_relu=decoder_relu)
self.regular_embedding_5_1 = BottleneckLayer(16, padding=1, dropout_prob=0.1, use_relu=decoder_relu)
self.embedding_transposed_conv = nn.ConvTranspose2d(16, embedding_dim, kernel_size=3, stride=2, padding=1, bias=False)
```

Figure 49: Embedding Branch Processing Code Part 2

In the ENet architecture, the embedding branch is crucial for producing feature embeddings that are spatially dense and semantically rich. This branch specifically targets the generation of embeddings that can be used for instance segmentation. Figure 48 and 49 shows the Embedding Branch Processing Code.

Similar to the Binary Branch workflow, the Embedding Branch first employs an Upsampling Layer, which increases the resolution of feature maps from 128 channels to 64. This step is followed by two Bottleneck Layers that refine and enhance the features. Another Upsampling Layer then transitions the feature maps from 64 to 16 channels, moving closer to the original dimensions of the input image. A final Bottleneck Layer applies additional refinements. The last layer in the Embedding Branch, **self.embedding_transposed_conv**, uses a transposed convolutional layer to upscale the 16-channel embeddings back to the full input resolution. This process creates dense, pixel-wise embeddings that are finely detailed and ready for further processing or direct application in segmentation tasks.

4. Experiment & Discussion

4.1 Dataset

For this project, TuSimple was used for evaluating the model. The TuSimple dataset comprises 6,408 images of US highways, each with a resolution of 1280×720. It is divided into 3,626 training images, 358 validation images, and 2,782 test images where labels were in a json file format.

4.2 Evaluation

```
binary_accuracy_val = correct_predictions_binary / total_pixels_count
binary_total_false_val = total_pixels_count - correct_predictions_binary
binary_precision_val = (true_positives_count) / (true_positives_count + false_positives_count)
binary_recall_val = (true_positives_count) / (true_positives_count + binary_total_false_val - false_positives_count)
binary_f1_score_val = 0 if (binary_recall_val == 0) or (binary_precision_val == 0) else 2 / (1/binary_precision_val + 1/binary_recall_val)
```

Figure 50: Evaulation Code

The following metrics provide a comprehensive assessment of how well the model predicts the presence of lane markings in images as seen in Figure 50:

1. **Binary Accuracy**: This metric is fundamental in assessing the overall effectiveness of the model in classifying pixels correctly as lane or non-lane. High binary accuracy indicates that the model is generally successful in distinguishing lane markings from the surrounding environment. It is calculated using the formula:

$$Binary\ Accuracy = \frac{Correct\ Precision}{Total\ Pixel\ Count}$$

This metric is straightforward and gives a quick snapshot of model effectiveness across the entire dataset.

2. **Binary Precision**: Precision in lane detection is critical to minimize false positives (FP)—situations where the model incorrectly identifies non-lane elements as lanes. True Positives (TP) are detected lanes with intersection over Union (IOU) above 0.5. This is vital for avoiding misleading lane guidance:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

High precision is particularly important in cluttered where non-lane road markings or other artifacts are present.

3. **Binary Recall**: Recall measures the model's ability to identify all actual lane markings. High recall reduces false negatives, where actual lanes are missed, ensuring that the vehicle recognizes all relevant lane information:

$$Binary\ Recall = \frac{True\ Positive}{True\ Positve + Binary\ Total\ False\ Val - False\ Positive}$$

False Negatives Count is adjusted for non-lane elements falsely marked as lanes. Environments

4. **Binary F1 Score**: The F1 Score harmonizes precision and recall, providing a single measure that balances both sensitivity and specificity. This is crucial when precision and recall are equally important:

$$Binary\ F1\ Score = 2 \times \frac{Binary\ Precision\ \times Binary\ Recall}{Binary\ Precision + Binary\ Recall}$$

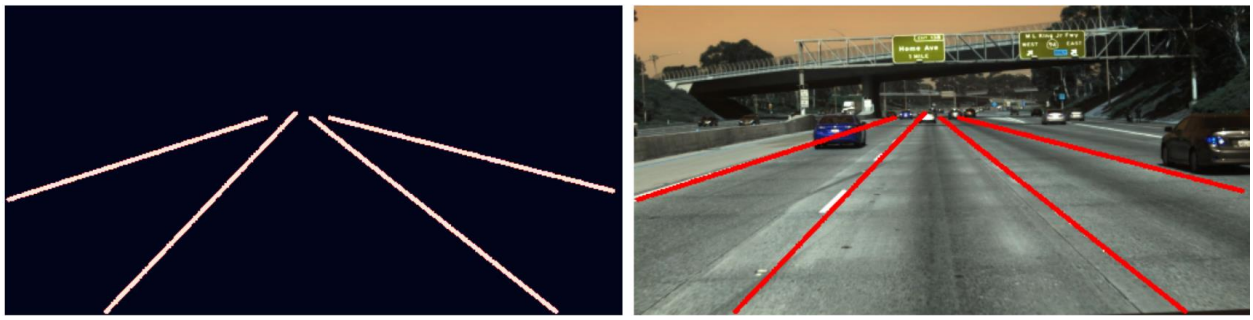
A high F1 score indicates a robust model that accurately detects lanes with minimal errors.

5. **Mean Losses**: The average losses (binary and instance) during training give insight into the model's learning progress and convergence. This metric helps in tuning the model during the training phase to minimize error:

4.3 Results

4.3.1 Types of Road

In the provided images, we can observe the lane detection model's performance across different road types. On straight roads, the model demonstrates a strong ability to track and project the course of the lane markings consistently, providing a clear path for vehicle navigation as seen in Figure 51. This is essential as maintaining lane integrity in well-defined, straight segments is foundational for autonomous driving.



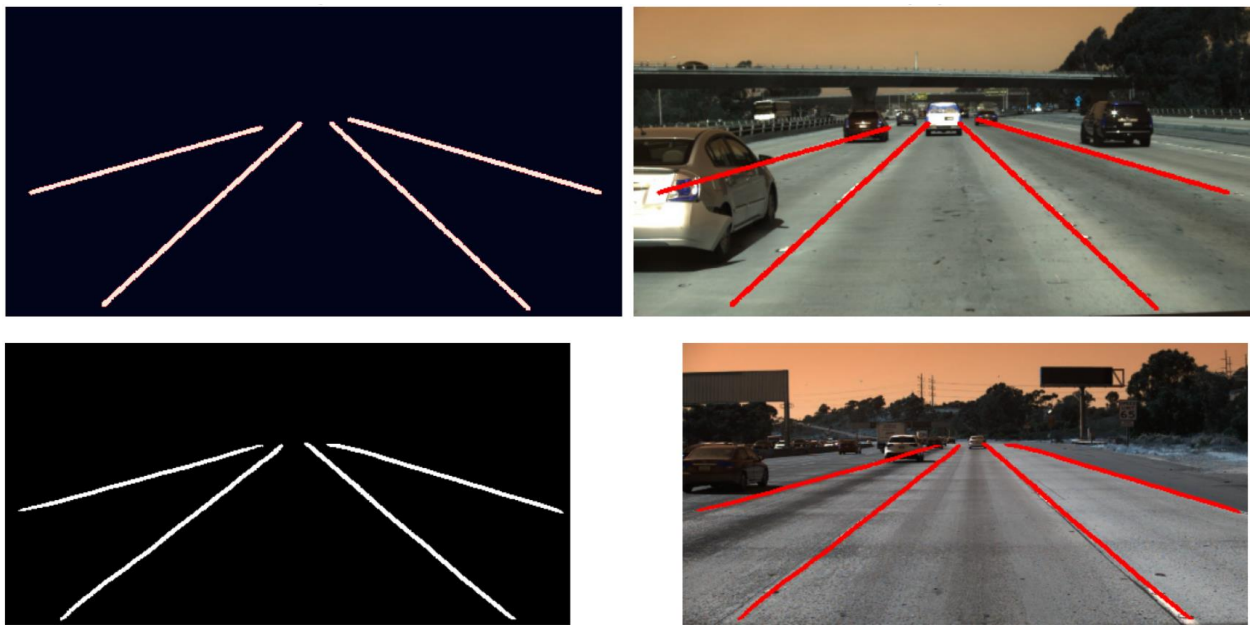


Figure 51: Straight Road lane detection result

On curved roads, which present a more complex challenge due to the varying radii and potential occlusions, the model appears to adapt well, tracing the curvature of the lanes accurately as seen in Figure 52. The robustness of the model in handling curves is particularly notable as it suggests a comprehensive understanding of lane features and the ability to predict their trajectory.

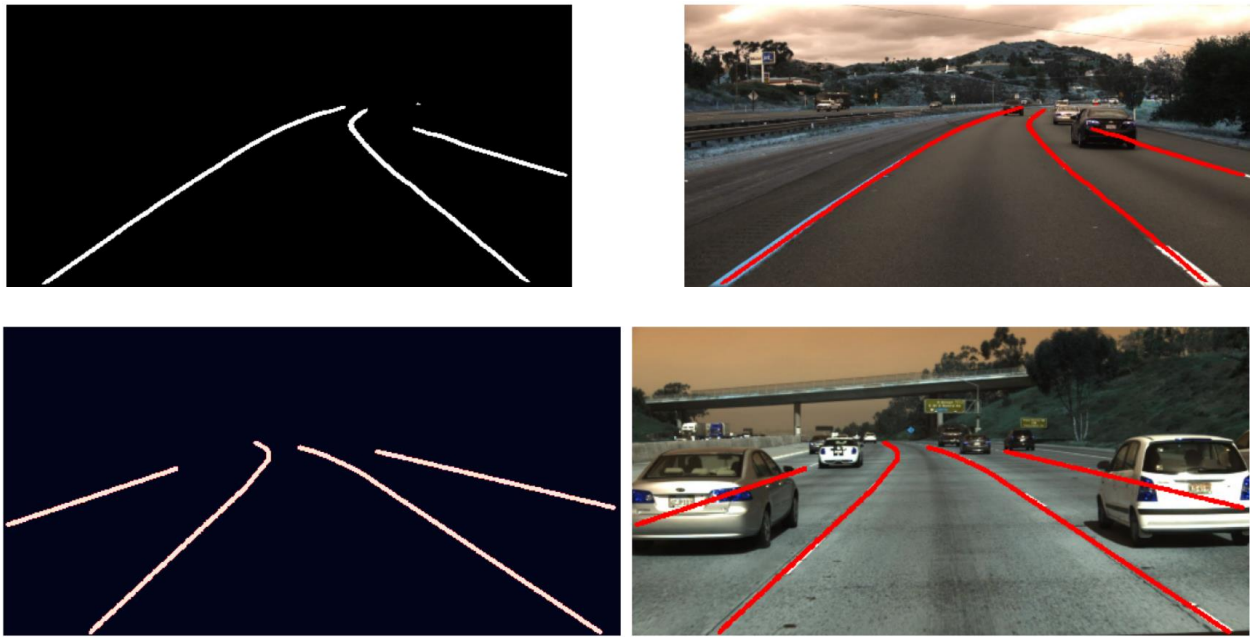


Figure 52: Curved Road lane detection result

Across all road types, the model's proficiency in delineating lanes contributes to safer and more reliable guidance for autonomous systems, ensuring that vehicles stay within lane boundaries under diverse driving conditions.

4.3.2 Machine Learning Result

The lane detection model based on the Enet architecture was subjected to training over 100 epochs to evaluate its efficacy in accurately delineating lane markings from road images. This training was aimed at minimizing both binary and instance losses as well as enhancing both accuracy and F1-score. The graphs below shows the graphs of Binary Loss vs Epoch(Figure 53), Instance Loss vs Epoch (Figure 54), Accuracy vs Epoch(Figure 55), and F1-score vs Epoch(Figure 56).

Initial Epochs: The training commenced with its first epoch with a binary loss of 0.1376 and a significantly higher instance loss of 0.9994, indicative of the model's initial struggle with feature extraction and lane differentiation. Despite these high losses, the binary accuracy was relatively high at 97.17% with a low F1-score of 0.0183. However second epoch show rapid improvement in loss reduction where binary score decrease to 0.0875 and instance loss decrease to 0.4792 and F1-score increment greatly increase to 0.0494. This shows the model quick adaptation to the nuances of lane detection

By the 20^{th} iteration, the binary losses has reduce to 0.0493, instance losses reduces to 0.0719, accuracy increased to 0.9826 and F1-score increase to 0.6592. From this iteration onwards, the reduction of the binary and instances losses as well as the increase in accuracy and F1-score has started to stabilize. This mark a significant advancement in the odel's ability to generalize from the training data, reducing error substantially.

Peak performance: The model achieved its best F1-score of 0.7233 by the 69^{th} iteration, with a binary loss of 0.037, instance loss of 0.0232, binary loss of 0.0327 and accuracy of 0.9856.This peak performance highlights the model's enhanced predictive capabilities and its efficiency in recognizing and predicting lane markings accurately.

Consistency and Convergency: Past the 69^{th} iteration, the model displayed a consistency in maintaining a high level of accuracy and F1-score which stabilizes towards the final epochs.

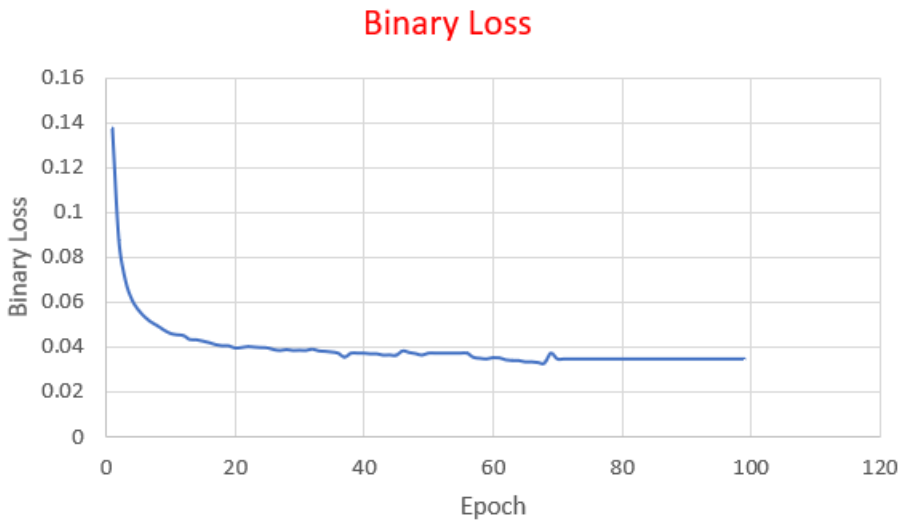


Figure 53: Binary Loss vs Epoch

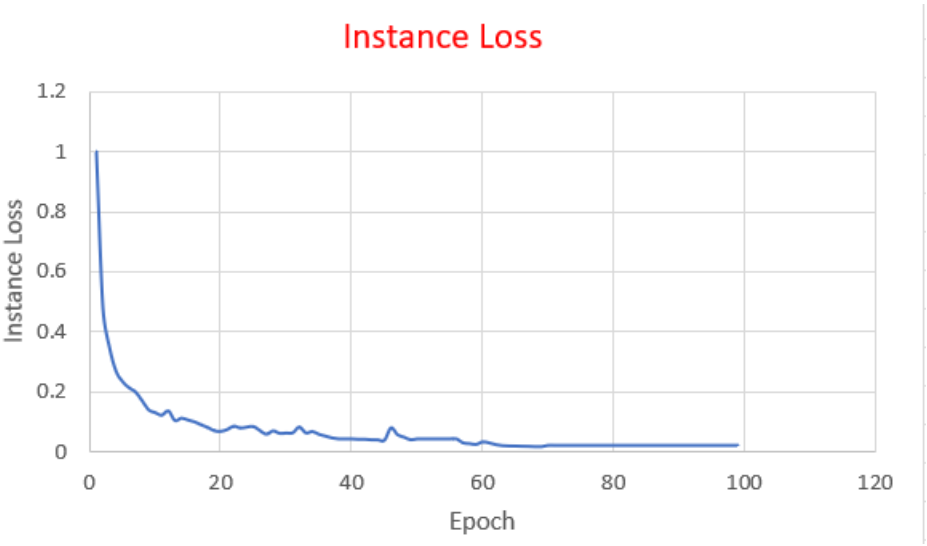


Figure 54: Instance Loss vs Epoch

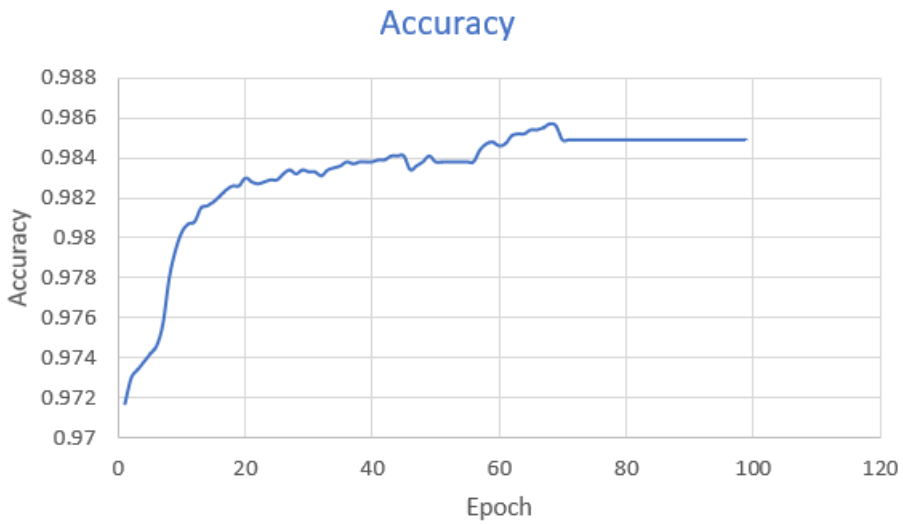


Figure 55: Accuracy vs Epoch

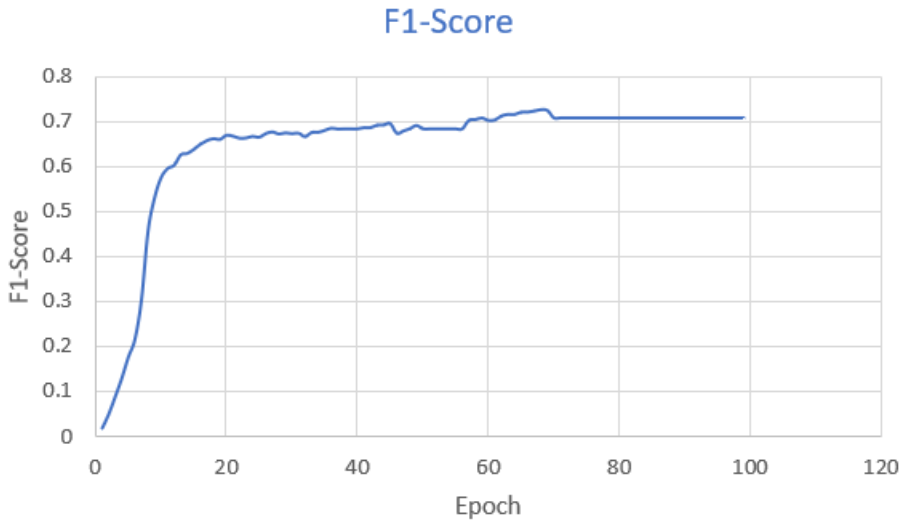


Figure 56: F1-score vs Epoch

5. Future Work and Considerations

Although the lane detection model on Enet had a reasonable F1-score in this project, it is considered an old Neural Network which was introduced in a research paper by Adam Paszke in 2016[34]. Some future work would be:

1. Utilization of newer Machine Learning Model

   Currently there is a newer CNN known as EfficientNetV2 that is founded in 2018. It is able achieve 87.3% of top accuracy while training at a much faster rate, being able to distinguishing between different types of road lanes under varying lighting and weather conditions.

2. Tackling more challenging road scenarios

   As autonomous driving technology continues to evolve, addressing challenging road scenarios remains a top priority. Future iterations of our model will focus on enhancing performance under conditions that currently pose significant challenges for lane detection systems. This includes, but is not limited to, poor lighting conditions (Figure 57) such as driving at twilight or pre-dawn hours, in tunnels, or facing the glare of oncoming headlights.

(a) W/O ALLE (b) W/ ALLE

Figure 57: Low light situation

3. Integrating Object Detection in lane detection

   Incorporating object detection such as YOLO (You Only Look Once) or SSD (Single Shot MultiBox Detector) into lane detection systems represents a pivotal advancement in the development of autonomous vehicles and advanced driver assistance systems (ADAS). The successful integration of object detection with lane detection not only enhances the safety features of autonomous vehicles but also significantly boosts their operational domain. It allows vehicles to perform safely and efficiently in complex urban environments, paving the way for widespread adoption of autonomous driving technology.

6. Conclusion

The development and evaluation of the lane detection model, particularly using the ENet architecture, demonstrates significant progress in the field of autonomous driving and advanced driver assistance systems (ADAS). This report has detailed the implementation, testing, and performance of the lane detection system across various road conditions, showcasing its robustness and accuracy in identifying lane markings under typical operating scenarios.

The ENet model, adapted for lane detection, effectively processed high-resolution road images, successfully delineating lane boundaries with high precision and recall rates. This was evidenced by the F1 score of 72% peak and accuracy metrics obtained during the testing phases. Using hough transform model fitting, the model is able to perform lane detection with curved and straight roads with relatively high reliability.

7. Reference

1. World Health Organization, “Road traffic injuries,” *World Health Organization*, Dec. 13, 2023.https://www.who.int/news-room/fact-sheets/detail/road-traffic injuries
2. “Toyota Safety Sense (TSS) | Toyota (Country),” *www.toyota.com.sg*. https://www.toyota.com.sg/discover-toyota/toyota-safety-sense#eye-on-safety (accessed Apr. 19, 2024).
3. “What Image Processing Techniques Are Actually Used in the ML Industry?,” *neptune.ai*, Nov. 12, 2020. https://neptune.ai/blog/what-image-processing-techniques-are-actually-used-in-the-ml-industry
4. “NV5 Geospatial Docs Center,” *www.nv5geospatialsoftware.com*. https://www.nv5geospatialsoftware.com/docs/FXExampleBasedTutorial.htmlhttps://www.nv5geospatialsoftware.com/docs/FXExampleBasedTutorial.html (accessed Apr. 19, 2024).
5. “FeatureExtraction,” *www.mathworks.com*. https://www.mathworks.com/discovery/feature-extraction.html#:~:text=Feature%20extraction%20refers%20to%20the
6. V. DaSilva, “Computer Vision for Busy Developers: Convolutions,” *HackerNoon.com*, Jun. 26, 2019. https://medium.com/hackernoon/cv-for-busy-developers-convolutions-5c984f216e8c
7. “Sobel Filter - an overview | ScienceDirect Topics,” *www.sciencedirect.com*. https://www.sciencedirect.com/topics/computer-science/sobel-filter
8. Kang & Atul, “sobel operator,” *TheAILearner*, May 24, 2019. https://theailearner.com/tag/sobel-operator/
9. Ashish, “Understanding Edge Detection (Sobel Operator),” *Medium*, Sep. 26, 2018. https://medium.datadriveninvestor.com/understanding-edge-detection-sobel-operator-2aada303b900
10. “OpenCV | Real Time Road Lane Detection,” *GeeksforGeeks*, Dec. 12, 2019. https://www.geeksforgeeks.org/opencv-real-time-road-lane-detection/
11. “OpenCV: Canny Edge Detection,” *docs.opencv.org*. https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

12. K.-Y. Chiu and S.-F. Lin, “Lane detection using color-based segmentation,” *IEEE Xplore*, Jun. 01, 2005. https://ieeexplore.ieee.org/abstract/document/1505186 (accessed Mar. 19, 2023).
13. J. Son, H. Yoo, S. Kim, and K. Sohn, “Real-time illumination invariant lane detection for lane departure warning system,” *Expert Systems with Applications*, vol. 42, no. 4, pp. 1816–1824, Mar. 2015, doi: https://doi.org/10.1016/j.eswa.2014.10.024.
14. “Image Transforms - Hough Transform,” *homepages.inf.ed.ac.uk*. https://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm
15. Gabrielli, Alessandro & Alfonsi, Fabrizio & Annovi, Alberto & Camplani, Alessandra & Cerri, Alessandro. (2021). Hardware Implementation Study of Particle Tracking Algorithm on FPGAs. Electronics. 10. 2546. 10.3390/electronics10202546.
16. N. Ferdinand, “A Deep Dive into Lane Detection with Hough Transform,” *Medium*, May 05, 2020. https://towardsdatascience.com/a-deep-dive-into-lane-detection-with-hough-transform-8f90fdd1322f
17. C. P. Bathula, “Machine Learning Concept 69: Random Sample Consensus (RANSAC),” *Medium*, Apr. 14, 2023. https://medium.com/@chandu.bathula16/machine-learning-concept-69-random-sample-consensus-ransac-e1ae76e4102a
18. IBM, “What Is Machine Learning?,” *IBM*, 2023. https://www.ibm.com/topics/machine-learning
19. Peng, Junjie & Jury, Elizabeth & Dönnes, Pierre & Ciurtin, Coziana. (2021). Machine Learning Techniques for Personalised Medicine Approaches in Immune-Mediated Chronic Inflammatory Diseases: Applications and Challenges. Frontiers in Pharmacology. 12. 10.3389/fphar.2021.720694.
20. J. Carew, “What is reinforcement learning? A comprehensive overview,” *SearchEnterpriseAI*, Feb. 2023. https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning4
21. V. Kanade, “What Is Reinforcement Learning? Working, Algorithms, and Uses,” *Spiceworks*, Sep. 29, 2022. https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-reinforcement-learning/

22. “Decoding the CNN Architecture: Unveiling the Power and Precision of Convolutional Neural Networks - Part I,” *www.linkedin.com*. https://www.linkedin.com/pulse/decoding-cnn-architecture-unveiling-power-precision-neural-moustafa/
23. M. Mandal, “CNN for Deep Learning | Convolutional Neural Networks (CNN),” *Analytics Vidhya*, May 01, 2021. https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/
24. M. Mishra, “Convolutional Neural Networks, Explained,” *Medium*, Aug. 27, 2020. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939
25. “CS231n Convolutional Neural Networks for Visual Recognition,” *Github.io*, 2012. https://cs231n.github.io/convolutional-networks/
26. A. Rosebrock, “Convolutional Neural Networks (CNNs) and Layer Types,” *PyImageSearch*, May 14, 2021. https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/
27. N. Srivastava, G. Hinton, A. Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014, Available: http://jmlr.org/papers/v15/srivastava14a.html
28. J. Raitoharju, “Convolutional neural networks,” *Elsevier eBooks*, pp. 35–69, Jan. 2022, doi: https://doi.org/10.1016/b978-0-32-385787-1.00008-7.
29. D. Unzueta, “Fully Connected Layer vs Convolutional Layer: Explained | Built In,” *builtin.com*, Oct. 18, 2022. https://builtin.com/machine-learning/fully-connected-layer
30. N. J. Zakaria, M. I. Shapiai, R. A. Ghani, M. N. M. Yassin, M. Z. Ibrahim, and N. Wahid, “Lane Detection in Autonomous Vehicles: A Systematic Review,” *IEEE Access*, vol. 11, pp. 3729–3765, 2023, doi: https://doi.org/10.1109/ACCESS.2023.3234442
31. G. Nanos, “Neural Networks: Pooling Layers,” *Baeldung*. https://www.baeldung.com/cs/neural-networks-pooling-layers
32. P. S, “Convolution Neural Network - Better Understanding!,” *Analytics Vidhya*, Jul. 16, 2021. https://www.analyticsvidhya.com/blog/2021/07/convolution-neural-network-better-understanding/

33. Neven, Davy & Brabandere, Bert & Georgoulis, Stamatios & Proesmans, Marc & Van Gool, Luc. (2018). Towards End-to-End Lane Detection: an Instance Segmentation Approach. 286-291. 10.1109/IVS.2018.8500547.
34. A. Paszke, Abhishek Chaurasia, S. Kim, and E. Culurciello, "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation," Jun. 2016, doi: https://doi.org/10.48550/arxiv.1606.02147.
35. M. Tan and Q. Le, "EfficientNetV2: Smaller Models and Faster Training." Available: https://arxiv.org/pdf/2104.00298.pdf