Aalto-yliopisto
Aalto-universitetet
Aalto University

Master's Programme in ICT Innovation: Visual Computing and Communication

Real-Time Lane Detection on Embedded Systems for Control of Semi-Autonomous Vehicles

Applying Deep Learning on Embedded Systems

Jacob Young

Master's Thesis
2022

Aalto-yliopisto
Aalto-universitetet
Aalto University

Author Jacob R. Young

Title of thesis Real-Time Lane Detection on Embedded Systems for Control of Semi-Autonomous Vehicles

Programme ICT Innovation

Major Visual Computing and Communication

Thesis supervisor Dr. Juho Kannala

Thesis advisor(s) Dr. Vahid Abrishami & Dr. Timo Tossavainen

Collaborative partner Basemark Oy

Date 21.07.2022 **Number of pages** 63 + 9 **Language** English

Abstract

Car accidents are the leading cause of death and injuries in most countries. advanced driving assistance systems and intelligent autonomous vehicles aim to improve road safety, traffic issues, and the comfort of passengers. Lane detection is a pivotal element in advanced driving assistance systems as lane understanding is essential in maneuvering the car safely on roads. Detecting lanes in real-world scenarios is challenging due to adverse weather, lighting conditions, and occlusions. However, as the computational budget available for lane detection in the systems above is limited, a lightweight, fast and accurate lane detection system is crucial.

This thesis proposes a simple, lightweight, end-to-end deep learning-based lane detection framework following the row-wise classification approach. The inference speed is significantly increased by reducing the computational complexity and using a light backbone. In contrast to other systems, the proposed method can handle lane-changing scenarios by offering three lane candidates within the model. Additionally, we introduced a second-order polynomial fitting method and Kalman filter for tracking lane points as post-processing steps to improve the overall accuracy and stability of the system.

The proposed lane detection method can provide over 500 frames per second on an Nvidia GTX 3080 notebook with our lightweight model and a median 48 frames per second on an Nvidia Jetson AGX Xavier while producing comparable accuracy to most of the state-of-the-art approaches.

Keywords: Lane Detection, Deep Learning, Embedded Systems, ADAS, Autonomous Driving, Machine Learning

Contents

Preface

I want to thank Dr. Juho Kannala and my advisors at Basemark, Dr. Vahid Abrishami & Dr. Timo Tossavainen for their guidance and to thank my partner Sarah Van Sicklen for listening to me talk about the same subject for months on end. I would also like to thank my Father for his support and push to pursue a master's degree. Finally to the rest of my family (The Blys, The Morrisons & The Youngs).

Otaniemi, 21 July 2022
Jacob Young

Symbols and abbreviations

Symbols

$H[u, v]$	A 2D filter, kernel or image
μ	The *mean* or *average* of some values
σ	The standard deviation of the mean of a normal distribution
θ	An angle *theta* in radians

Operators

exp(x)	The exponential function e^x
$*$	Convolution operator
∇f	The gradient of f
Σ_i	Sum over index *i*
$\frac{\partial f}{\partial x}$	Partial derivative of *f* with respect to variable *x*
$\|\|x\|\|$	The magnitude of some variable x

Abbreviations

ADAS	Advanced Driving Assistance System
FPS	Frames-per-Second
SOC	System-On-a-Chip
ML	Machine Learning
DL	Deep Learning
NN	Neural Network
CNN	Convolutional Neural Network
LSTM	Long short term memory
ResNet	Residual Network
IPM	Inverse Perspective Mapping

1. Introduction

1.1 Motivation

Motor vehicle based transportation is critical for individual participation in the modern world, yet over one million people die each year in motor vehicle related accidents. Vehicle related injuries are the leading cause of death for children and young adults (5-29 years old) [1]. According to the National Highway Transportation Safety Administration (NHTSA), human error accounts for an overwhelming majority of all auto related accidents [2]. For these reasons, research institutions and private companies have sought to make driving safer by eliminating human error.

In previous decades, reducing the impact of crashes with physically based safety features has been the main method of making driving safer. This approach has saved countless lives, but is reaching its limit and does not account for human error. Another approach is to give vehicles the ability to perceive their environment, potentially eliminating human error. Advanced Driving Assistance Systems (ADAS) features such as automatic braking, blind spot detection and backup cameras have the potential to reduce crashes by up to 40% [3, 4]. The European Road Safety Observatory (ERSO) defines ADAS as "vehicle-based intelligent safety systems which could improve road safety in terms of crash avoidance, crash severity mitigation and protection and post-crash phases" [5]. We seek to develop a lane detection model that performs accurately and fast enough to be used as the basis for control systems on a System-On-a-Chip (SOC).

Lane detection is one of the most fundamental and safety-critical tasks in autonomous driving. Its application ranges from ADAS features such as lane-keeping to higher-level autonomy tasks such as fusion with high-definition maps and trajectory planning. Detecting lanes in real-world scenarios is challenging due to adverse weather, lighting conditions, and occlusions. Moreover, lane detection algorithms should be computationally efficient and runnable on embedded systems limited to small memory and processor resources. Light image processing-based methods are considered a solution for this limit. However, these methods can not overcome the challenges above. New approaches based on Deep Neural Networks (DNNs) can overcome many of these challenges but demand higher computational power, memory, and battery power requirements, making them useless for this application. Although many light models have been introduced for the lane detection task recently, the problem is there for SOCs with minimal resources.

In this thesis, we proposed a light DNN model for the lane detection task, which is runnable on SOCs with limited resources. The proposed model comprises a light backbone for feature extraction and three heads, one for predicting coordinates, another for classifying points as under or above the horizon, and one for proposing lane candidates. For performance sake, this method uses some fixed row anchors to predict the position of lane points. The detection accuracy is further improved by fitting a second-order polynomial to the line points and tracking them using a Kalman filter as post-processing. The model was tested on two SOCs: Nvidia Jetson AGX Xavier and Jetson Nano which provided 100 and 46 FPS, respectively. In addition to the performance, the proposed method can correctly predict lane position when changing lanes and also detect the curvatures.

1.2 ADAS and Automated Driving Systems

Manufacturers have slowly but surely been automating some of the previously most dangerous parts of driving. From Lane Departure Warnings (LDW) first invented in 1992 by Mitsubishi, to driver assistance coming about from Toyota in 2004 [6]. A big leap towards defining and standardizing ADAS and Automated Driving Systems (ADS) came in 2014, when the Society of Automotive Engineers (SAE) released a document known as SAE J3016 *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. This document sets a guideline for ADS which performs sustained Dynamic Driving Tasks (DDT). The SAE defines 6 degrees of automation in motor vehicles, which range from no driving automation (level 0) to full driving automation (level 5) [7]. The levels are detailed as below:

Level 0: No Driving Automation - The driver performs all DDT (i.e. full manual driving) while the ADS provides warnings or momentary emergency intervention.

Level 1: Driver Assistance - Driver performs all DDT not performed by the ADS and supervises features performed by it. The ADS is capable of performing *either* longitudinal *or* lateral vehicle motion control.

Level 2: Partial Driving Automation - Driver performs all DDT not performed by the ADS and supervises features performed by it. The ADS is capable of performing *both* longitudinal *and* lateral vehicle motion control.

Level 3: Conditional Driving Automation - The ADS performs all DDT within its Operational Design Domain (ODD) while engaged and will disengage after requesting the driver to intervene when it detects a situation outside of its ODD or when the driver requests to drive.

Level 4: High Driving Automation - The driver does not need to supervise the ADS while it is engaged. The ADS will determine if the condition is safe enough for the user to drive upon request.

Level 5: Full Driving Automation - The vehicle can drive everywhere (i.e. on roadways public and private) in all conditions completely autonomously.

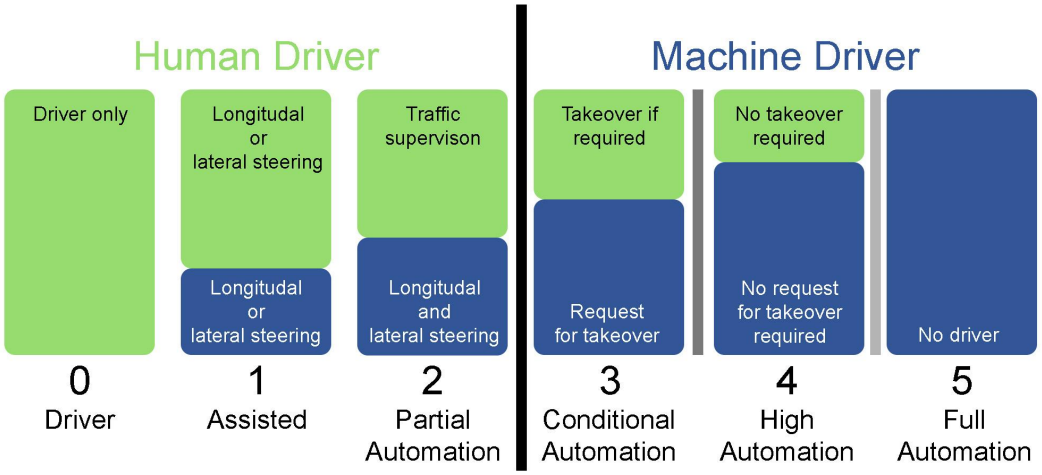


Figure 1.1: The 6 SAE levels of automation.

In SAE levels 0 through 2, the human must constantly supervise the vehicle, even when semi-autonomous features are operating. Conversely, in levels 3 through 5, the human passenger is not considered driving, except in level 3 when an autonomous feature requests the human must drive. In higher levels of automation, the user's role may switch between driver and passenger. In all cases, the person who may be designated as the driver is responsible for checking the vehicle is in working condition.

Features we are interested in for this thesis are LDW, considered SAE level 0, and Lane Keeping Assistance (LKA), which is SAE level 1 or 2 depending on if it is performed on a sustained basis [7]. In the April 2021 revision J3016_202104, LKA is excluded from the scope of the DDT because it only provides momentary intervention [8]. Although LKA is difficult to fit within one SAE driving level, we seek to develop a model capable of SAE level 2++ autonomy such that our lane detection algorithm

can serve as a foundation to provide other more advanced features in the future.

To create an autonomous vehicle, it is first necessary to measure the environment through various sensors, such as camera, radar, IMU and GPS. Combining output from multiple sensors into a single system is called *sensor fusion*. Using the information from multiple sensors enables the development of perception algorithms such as lane and obstacle detection and other methods of perceiving the environment. Using these algorithms, it is possible to perform trajectory planning and path navigation in the control system. Vehicles with the capability of environmental perception can keep passengers safe by minimizing the potential for human error. With these foundational algorithms, the autonomous vehicle can perform lane keeping, obstacle avoidance and other safety features.

1.3 Classic Image Processing versus Deep Learning

Methods for lane detection can be classified into two categories: image processing-based and neural network-based methods. The former category uses image processing techniques like edge detection and line detection along with color space information to detect the lanes, while the latter uses a model to learn lanes based on some internal extracted features. Currently, image processing-based methods are used more in the industry since these methods are computationally less expensive, but deep learning is quickly becoming widely adopted as advances in network efficiency are becoming better understood. The literature review will overview different lane detection methods and dive deeper into them in sections 2.1 and 2.2.

Before going further into classic lane detection methods and their evolution into using deep learning, we will briefly cover their differences (figure 1.2).

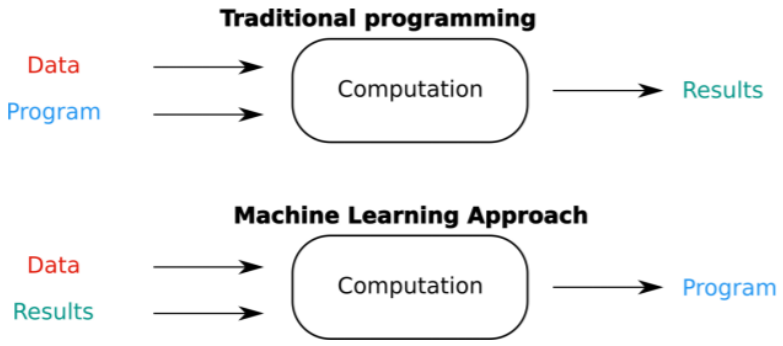


Figure 1.2: Traditional programming compared to machine learning [9].

Traditional programming requires the engineer to write a program to formulate rules with which data will be processed to produce some desired outcome. This approach is advantageous when we know exactly what rules a program should follow, which is not always the case in real life. However, it is almost impossible to develop rules for solving complex problems (e.g. classification) and that is exactly where machine learning comes to learn rules from existing examples.

In contrast, the machine learning approach automatically formulates the rules and generates a program by observing some example inputs and their expected outputs via a process called training. Typically the entire dataset is split up into three parts: a training, test and validation set. The purpose of this is to have enough data for the network to learn an approximation of the entire dataset while having unseen data which is useful for evaluating the effectiveness of the model.

During training, the machine learning model learns a linear/non-linear mathematical function which maps inputs to proper outputs. A loss function measures the quality of the model by quantifying the difference between the expected and predicted outcomes. Based on this difference, the model updates its parameters.

Deep Learning is a specialized subset of machine learning, where the statistical model takes the form of a deep neural network (DNN). DNNs take the form of a graph structure, typically composed of layers of computational units called neurons (figure 2.6). A Neuron is a combination of weights and biases input into a non-linear activation function (figure 2.5).

During training, the network's weights are tuned by using *backpropagation* and *gradient descent* to map inputs to an output prediction. Backpropagation is an algorithm which automatically computes derivatives for the entire network from the output layer to the input layer using the chain rule. Next, gradient descent uses the derivatives (i.e. gradient) output by backpropagation to adjust the weights in a way which moves the loss function towards a local minimum. These two algorithms occur iteratively for all data in the training set until the network is trained. Each pass through all data in the training set is called an *epoch*. Once trained, the weights of the network are frozen then used to make predictions in a process called *inference*.

The goal of training is to have the network learn general features present in the dataset to correctly predict features in unseen data. The downside of this process is that the model is only as good as the input data and its loss function. Feeding a model random data will not result in a useful program. It often takes many thousands to millions of well labeled

data points for the model to learn complex tasks. This process can be very laborious and expensive. Loss functions must be designed to minimize the error between the data and the label gradually. These cannot just be any function and are not “one size fits all”. Which loss function to choose depends on the situation and has a huge impact on how the model learns. They can be biased with domain-specific knowledge to aid the model in learning faster.

In image processing and computer vision, features are often detected through a process called convolution [10]. It works by crafting some kernel or filter then applying it to the image to detect some feature. Classic methods require hand-crafting of filters for detecting different features. These filters are extremely difficult to design and understand as it often requires multiple filters in conjunction to actually create a useful process for detecting things in an image.

In deep learning for computer vision, we often use a type of neural network biased towards image data, called a Convolutional Neural Network (CNN). This type of network was originally called a neocognitron, proposed by *K. Fukushima* in 1980 [11]. CNNs use image data to learn the filter required for detection, rather than hand crafting it. This gives networks the ability to learn a sequence of filters which are capable of detecting a hierarchy of features. For instance, a hand-crafted filter might be able to differentiate what makes up a human face, but a CNN can learn what exactly makes up a specific individual's face. The main reason deep learning has become so popular in computer vision and lane detection is that the series of filters the machine learns to generalize lane detection across many scenarios would be extremely difficult, if not impossible to craft by hand. Now that the two paradigms have been overviewed, we begin by diving deeper into classic methods and see how they eventually evolved into neural networks to more capably complete the same tasks.

2. Literature Review

2.1 Image Processing Based Methods

2.1.1 Edge Detection and Hough Transform

Before the advent of big data and deep learning for solving problems, lane detection was constructed as a line detection problem. Because of this, traditional methods of edge detection and line fitting, most notably the Hough transform [12] and Canny edge detection [13] were employed to detect lanes in a few specific cases [14, 15]. These methods were effective for detecting straight lines, which is useful in ideal cases such as driving on the highway with low traffic during a sunny day. Many improved iterations of the Hough transform were attempted including [16, 17, 18].

Each of these methods utilized a combination of preprocessing steps to obtain a cleaner detection with Hough transformation or they would modify parameters of Hough transform directly. Nonetheless, the underlying method remained the same. Fundamentally, the Hough transform is a voting technique that can be used to determine which line features present in the image are most important. The main idea is that, for a given edge in an image, it "votes" for other compatible lines. Once voting is finished, search for lines with the highest number of votes.

The Hough transform expects an edge detection image as input to identify potential lines. The Canny edge detection algorithm is a well known method which uses hand-crafted filters to detect steep changes in intensity. It works by filtering an image using two filters (i.e. kernels) which compute the derivative of the image with respect to the X and Y directions then applying additional post-processing steps detailed later in this section.

These operations rely on an image processing technique called *spatial filtering*, *that* extracts meaning from an image by analyzing spatial relationships between pixels. *Linear spatial filtering* finds features in images using *convolution*, or more specifically *2D discrete convolution*. Convolution is a linear translation invariant function from a finite sized filter applied to an image. This process uses a sliding window to compute a weighted sum of the current pixel and its neighbors based upon the filter when repeated for every pixel. The result of this is a filtered image or *feature map* which retains semantic information about the image. Mathematically, 2D discrete convolution can be defined as:

$$G[i,j] = H * F = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u, j-v] \tag{1}$$

Where, *G* is the resulting filtered image, *H* is the filter, *F* is the image and $*$ denotes the convolution operation. In practice convolution is implemented as an NxN matrix *H*, which represents a window that moves across each horizontal row of the image and replaces the value of each pixel with a weighted combination of surrounding pixels, resulting in a filtered image. Given some image F(x,y) in figure 2.1, a filter of size 3x3 is used, which accounts for a region of 9 pixels.

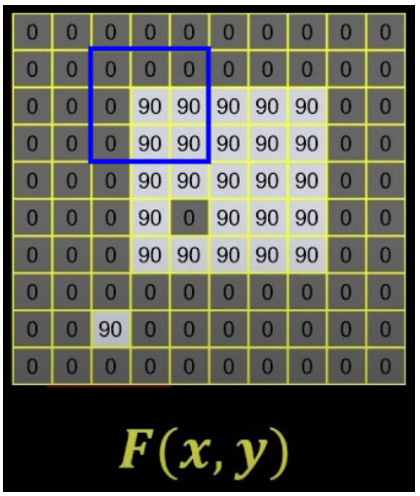
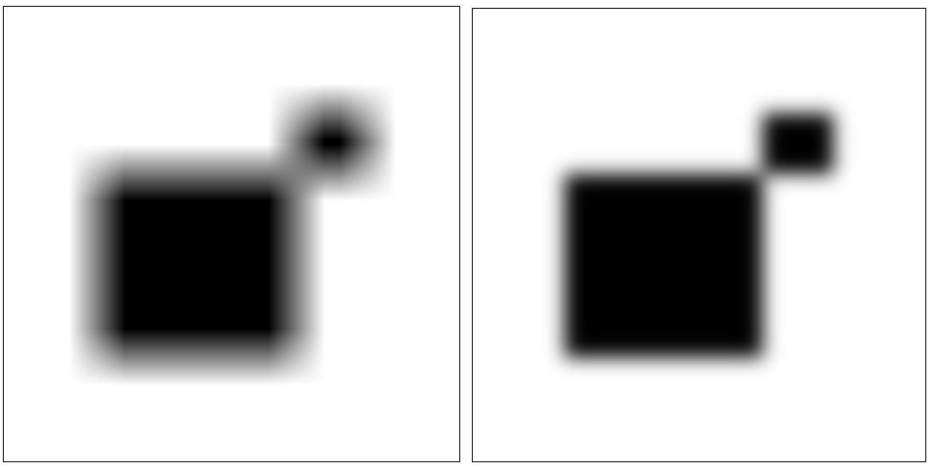


Figure 2.1: Convolution visualized [19].



(a) (b)

Figure 2.2: Box filter on an image (a), Gaussian filter on an image (b).

A simple example of filtering an image would be setting the center pixel of the filter to the average of surrounding pixels. Repeating this process for each pixel results in a blurred image with harsh artifacts (see figure 2.2a). There are better methods for blurring images, most notably using the Gaussian function (equation 2) which performs a circularly symmetric blur in figure 2.2b.

$$h(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{\sigma^2}} \tag{2}$$

Where σ is the standard deviation of the mean of a normal distribution.

Now that we have covered basic filtering, we will see how different functions can produce dramatically different results. The resulting filtered image is often called a *Feature Map*. We will see why this is the case later, as we use filtering to detect features in the image. Following through the Hough transform, images are pre-processed using Canny edge detection. Edges in images can be found by detecting steep changes in intensity throughout the image. Change in this intensity can be found using the image derivative. The Gaussian blurring effect detailed above is essential to the process, as Canny edge detection first applies this filter to smooth out noise in the image making it easier to detect steep changes in intensity. If smoothing is not performed, the noise present in the image will be amplified when taking the derivative, making it difficult to distinguish edges. Next the derivatives in the *X* and *Y* directions are calculated using another filter, combining both directional derivatives is referred to as the *gradient*. The derivative theorem of convolution states that it is possible to apply this filter directly to the previous Gaussian filter first, then filter the original image. The *gradient* is defined as:

$$\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] \tag{3}$$

We produce the gradient using two filters (one for each derivative direction). This is usually computed with two Sobel filters, defined in equations 4a and 4b:

$$h_x(u,v) = \frac{1}{8} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{4a}$$

$$h_y(u,v) = \frac{1}{8} * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{4b}$$

We apply these filters to the image separately, then take the magnitude of the two image derivatives. The *gradient magnitude* is defined as:

$$||\nabla f|| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2} \tag{5}$$

Afterwards, threshold the result using non-maximum suppression. This thins out the lines present in the gradient magnitude image. Non-maximum suppression works by filtering out neighboring pixels which do not exceed some threshold. In practice the *gradient orientation* is used to detect the peak intensity along an edge, this is defined as:

$$\theta = \tan^{-1}(\frac{\partial f}{\partial x} / \frac{\partial f}{\partial y}) \tag{6}$$

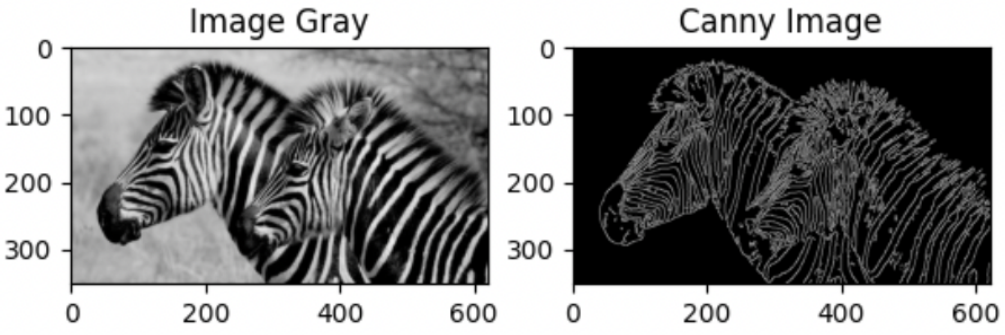


Figure 2.3: A Canny edge image.

Convolving an image with a different filter can give dramatically different feature maps. These maps are useful for detecting different things in the

image. Now that filtering has been covered, we return back to the first step of the Hough transform: Voting.

The key concept to perform voting is Hough Space. Lines in the image are transformed into points in Hough Space and vice versa. By dividing the image in Hough Space into bins, each line in that bin casts a vote. Counting up the votes of bins in Hough Space corresponds to candidate lines in the image.

Some difficulties encountered in using the Hough transform to detect lines in an image, is that there are many parameters to optimize, depending on factors such as noise. Relying on Canny edge detection means that Hough Transform images will have widely varying results depending on lighting conditions. This means that even with many modifications which yield better results, the narrow number of situations where these are useful, limits the number of scenarios in which this method of lane detection can be used. Due to this limitation, Hough transform based lane detection is restricted to daytime highway driving. Using it for vehicle control may not be safe in suboptimal illumination. Ultimately, the pervasiveness of complex lighting conditions and the inability to easily detect curves led to researchers developing other techniques.

Use of black and white color (i.e. *image intensity*) information has proven to be very useful in methods such as Canny edge detection and the Hough Transform. *J. Gonzalez and U. Ozguner* [20] built upon the idea of using intensity information by effectively leveraging the intensity histogram of the image. A histogram shows the frequency of light intensities present in the image. They couple this information with the use of a classic machine learning technique called a Decision Tree to detect and analyze lane markers. Their system creates structures that define lane boundaries in real time, resulting in images that can be used for preprocessing steps in lane and object detection [20]. This method resembles a rudimentary version of deep learning techniques used today.

2.1.2 Color-Based Methods:

The previous methods rely solely on image intensity, which overlooks integrating color information in the detection process. Using color to detect lanes begins with the simple premise that roads and lane lines are usually marked or have some consistent range of color (ex: lane lines are usually white or yellow, roads are very dark). Humans rely on color information to distinguish types of lane boundaries and therefore having programs capable of understanding this information should be incorporated into their function. Aside from filter based approaches, the rise of statistical methods

in lane detection provided solutions previously out of reach. Through this research we will show how the combination of filter and statistical methods naturally leads to using deep learning.

K. Chiu and S. Lin [21] propose a system which uses statistics to extract color thresholds based upon a region of interest in colored images. Their method uses this threshold to distinguish potential lane boundaries from the road. They present a fast and efficient way of estimating the geometric structure of the lane boundary that is robust in complex environments and various lighting conditions [21].

The use of classic machine learning techniques continued to emerge in lane detection as the role of statistical information became more understood in the importance of detecting lanes. Building on *K. Chiu and S. Lin's* approach, years later *J.W. Lee and J.S. Cho* [22] proposed a method which uses both color detection and edge orientation to minimize the error of a Bayesian classifier. This classifier is capable of distinguishing if a pixel belongs to the lane class or not. They estimate the probability distribution functions based upon classified features to adapt their model to various road conditions and lane types. Using this strategy, they are able to detect a single lane in real time [22].

As lane detection systems improved, more ADAS features went from academia to application. Early implementations of Lane Departure Warning (LDW) and Adaptive Cruise Control (ACC) features needed algorithms capable of dealing with changes present in the natural environment.

2.1.3 Inverse Perspective Mapping

Inverse perspective mapping is a classic method, which relies on the geometric camera calibration to create an inverse perspective map [23, 24, 25]. Effectively this converts the front camera image into a top down view, also called a bird's eye view image. After the image is converted, utilizing the Hue, Saturation and Value (HSV) color model, it can be used to isolate colored lane pixels such as safety boundaries. Next, edge detection filtering with thresholding (i.e. Canny edge detection) is performed to create a binary map of the lane pixels. Finally, edge pixels in the binary image are split into two halves, which then have a polynomial curve fit to each half of the image. The advantages of this approach over the Hough Transform is that it can create curved lines up to the extent of the horizon. This method of lane detection also runs in real time.

Figure 2.4: Transforming an image into bird's eye view using IPM [24].

Although this method improves upon the Hough transform, it still has several downsides. At least the following issues can be identified: IPM requires camera calibration which can be difficult or impractical in a real world setting. This is because the IPM assumes flat terrain, which is required for a linear transformation from the front camera view to a bird's eye view. In a perspective view, less pixels represent the lane as it draws closer to the horizon, therefore making curve fitting after projection more difficult. Moreover, the width of the lanes is set manually, so additional care must be taken when working with datasets containing varying road-widths. Given these factors, it may be unreliable to use IPM for autonomous vehicle control in less than ideal conditions.

A major limiting factor of the classic methods mentioned above is their reliance on canny edge detection. Unfortunately this introduces several limiting factors, as lane boundaries are not always present and detection of the lane is heavily influenced by lighting conditions. Fundamentally this filter cripples these approaches from being robust.

The most important takeaway from these methods is that the function (i.e. filter) which we convolve an image with is useful for detecting different features present in said image. Applying several different convolutional filters to an image can allow us to detect multiple features. Some examples of this is detecting that an image contains a dog versus figuring out which dog breed is shown through the more specific features. Later we will see that the foundation of convolutional neural networks (CNNs) is the ability to learn the filters necessary to detect features for any type of detection problem. The ability to utilize multiple learned filters in conjunction for detection is a huge advantage of CNNs as these filters are able to detect features which would be extremely difficult for humans to come up with the mathematical function by hand. The nature of neural networks being *universal function approximators* [26] means that they can

learn any possible filter which is capable of detecting any object or class of objects. This extremely powerful tool enables us to focus more on higher level concepts rather than on intricate details of why some given filter detects some type of feature.

2.2 Neural Network Based Methods

2.2.1 Basics of Neural Networks

Perceptrons: A simple explanation of neural networks starts with a classic machine learning component called a *perceptron*. Neural networks have been around for a long time, with perceptrons being detailed in *The Perceptron, A perceiving and recognizing automation* [27]. Originally believed to be capable of learning anything, the perceptron was hailed by Rosenblatt "Yet we are about to witness the birth of such a machine – a machine capable of perceiving, recognizing and identifying its surroundings without any human training or control" [27].

The perceptron is a linear, binary classifier, considered the most fundamental building block of the majority of neural networks [28]. Perceptrons work by taking a number of inputs (*x1, .., xn*), each assigned a weight (*w1, .., wn*) and producing a single output prediction from multiplying the input vector (*x*) with the weight vector (*w*) then adding a bias term (*b*) and summing the results. This weighted sum of the inputs is referred to as the *activation function* of the neuron. The weight vector which differentiates the input classes is a vector learned by the perceptron. Originally initialized randomly, this weight vector starts off with poor results.

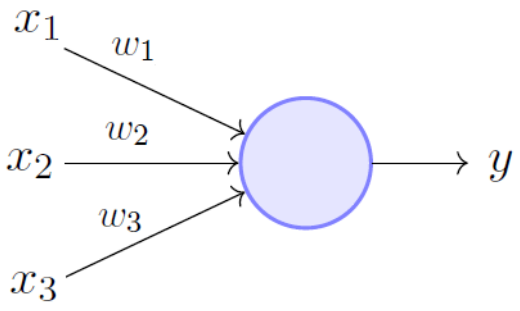


Figure 2.5: Perceptron Model [29].

Ultimately the perceptron learns a decision boundary, i.e. a line, also called a *hyperplane*, which separates two classes or two features, provided the classes are linearly separable. After learning from a sufficient number of

examples, the perceptron can predict the class of new instances. The learning process of the perceptron works by first feeding it a piece of data with which a prediction is made and compared to the ground truth label of the data. For instance, if the neuron is fed information of a dog, but it predicts a cat. The difference between the prediction and label is called the error, which is used to update the weight vector in a process called *Stochastic Gradient Descent* (SGD) [30]. SGD is similar to normal gradient descent except that the gradient is recalculated per data point rather than after training on the entire dataset. More on optimization algorithms will be covered at the end of this section.

Much later detailed in the 1969 book *Perceptrons* [29], the authors discovered that perceptrons are incapable of learning the "exclusive or" (XOR) function. This crashing discovery sewed disbelief in the ability of learning algorithms. The reason for this is that the perceptron is only capable of linearly separating classes, while the XOR is a nonlinear function. Although there are many different kinds of activation functions, the original perceptron algorithm used a linear function. Linear functions are incapable of learning decision boundaries between classes that can not be separated by a straight line [31].

Mutli-layer Perceptrons: The solution to separating nonlinear functions (such as XOR) was to use multiple perceptrons strung together with each using nonlinear activation functions. The most common activation function used in modern neural networks is the Rectified Linear Unit (ReLU) which employs a *Max(0, output)* function [32]. These multi-layer perceptrons (MLPs) (figure 2.6) are the basis of nearly all deep learning models. They consist of an input layer of perceptrons which feed their prediction to some number of hidden layers of perceptrons which are then summed into the output as a single value prediction or vector of predictions. The connections between each layer are densely connected, meaning that every input in the previous layer is connected to every output in the next layer.

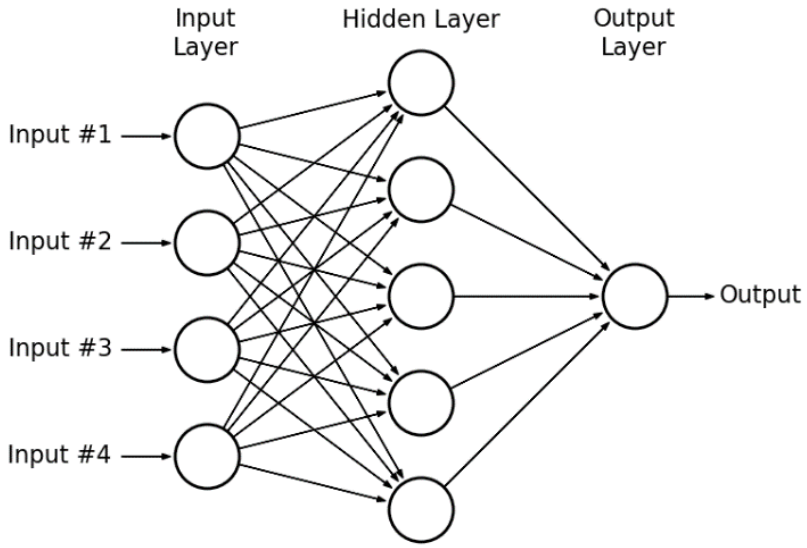


Figure 2.6: A multi-layer perceptron network [33].

Going further into detail on this issue, non-linear transformations are critical for a network to actually learn, because with linearity, the combination of all perceptrons in a network could be modeled by a single perceptron as a linear sum of any number of neurons is still a linear function. Most problems in the real world are nonlinear and thus require a nonlinear function to approximate their prediction solution. As a simple example using fluffiness and color as input to determine what the animal is would not have linear separation, as many different animals may be equally fluffy and the same color. This process of determining which features differentiates the classification most optimally is called *feature selection* [34].

As mentioned, classic computer vision methods of lane detection work in a limited number of scenarios, making them insufficient for the wide range of situations encountered in fully autonomous driving. MLPs are extremely useful for generating accurate predictions given sufficient data, but their input is often just a vector of values, while images are usually a 2D matrix of vectors. Spatial information in MLPs is not encoded, meaning information relative to some other bit of information is not carried across. This is because MLPs work using flattened 1D vectors, which loses the ordering of pixels [35]. CNNs address this problem, as local pixel groups are usually very important to understanding a portion of the image (i.e. spatial information is important for understanding features in an image).

One of the most useful ways for using images in neural networks is using a CNN, where rather than using a perceptron to transfer data from

one layer to another, learned weighted filters pass information forward through the network. These filters are initialized randomly and eventually come to approximate some function given sufficient data. After the demonstration of a powerful CNN called AlexNet in 2012 the pursuit of lane detection with machine learning and deep learning techniques took off [36].

Convolutional Neural Networks differ from MLPs in that they leverage spatial structure in an image to predict features. This way of structuring a network can be seen as biasing a network (i.e. specializing it to efficiently handle certain data structures [images])[37, 38]. Effectively, CNNs can be viewed as a special subset of the more general MLP.

By combining concepts of spatial information, feature understanding through filters (as seen in Canny edge detection) and MLPs, we can use CNNs to learn a hierarchical set of features ranging from low detailed information called local features (such as edges) to high level (global) features (ex: eyes, facial structure). The learned features present in some dataset can then be generalized, allowing the network to correctly classify new data.

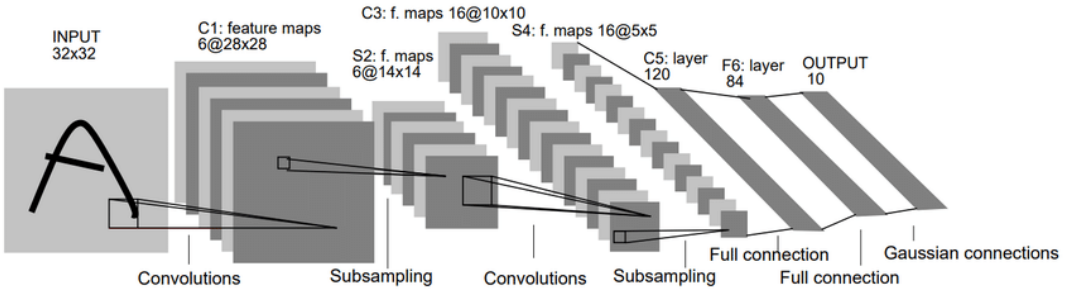


Figure 2.7: A CNN depicting the LeNet-5 architecture [39].

As demonstrated in figure 2.7, CNNs have four main operations. Namely, convolution, non-linear activation functions, subsampling (pooling) and full connection (classification). The network architecture specifies how many of these operations will be used and in what order. Once set they do not change.

During training, the goal of the network is to learn a set of feature maps which generate a high response when classifying data. Learning is done by adjusting the weights of the filters until they converge to the correct classification. Each pass through the network, the input data convolved with the filters to create feature maps. These feature maps are then passed to further layers, where low-level features are synthesized into higher level features. At the end of the network, feature maps are flattened into a feature vector, which outputs a prediction for the given data point. A loss function

is used to calculate the error between the output prediction and the labeled data. This error is then used in the backpropagation step to calculate gradients which determine how the filters should be adjusted using gradient descent. Once trained, the loss function and backpropagation steps are no longer used. The network maintains a set of filters which will generate a high response to new data of the same classes which the network was trained on.

Going into detail, convolution uses filters to create feature maps that preserve the spatial relationship between a neighborhood of pixels. Since convolution is a linear operation, a non-linear activation function must be applied after the convolution step for the network to learn the filters. Each new data point fed into the model adjusts the filters to better classify features present in the image. The network learns a hierarchy of features given sufficient data. Filters earlier in the network correspond to lower level features, while filters later in the network take previous feature maps to learn higher level features. Depending on the size of the filter and the padding around the input image, convolution may reduce spatial dimensionality while increasing feature dimensionality (e.g, 36x36x3 → 28x28x6). Decreasing input and filter size can optimize the prediction process, saving computational time. Another way to do this is using pooling.

Pooling is a downsampling operation that reduces image resolution while preserving spatial relationships between image regions. The typical type of pooling used is called *max pooling*, which takes the maximum value in some given patch of the original image and uses it as the pixel intensity in the downsampled image. Pooling passes only the most important features through the network.

Once these actions have been performed many times, the final layer contains several small feature maps that describe high level details. These are then flattened and *fully connected* like an MLP which enables the network to generate a classification vector or value prediction. The fully connected layer also has the advantage of learning non-linear combinations of high level feature maps. Typically the prediction is passed to a softmax function which converts the output into a statistical distribution which sums to one [31]. As an example using the softmax function in equation 7, the prediction may output a classification vector which contains the values (0.75, 0.1, 0.05, 0.03, 0.0.2) which means the network predicts the animal contained in the image has a 75% chance to be classified as an ermine, 10% cat, 5% dog, 3% squirrel and 2% bear. Softmax provides a clean way of understanding the prediction when multiple classes are present in the same image.

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{7}$$

Encoder-Decoder Networks: The output prediction of the network does not necessarily have to be a single value or vector of probabilities. By performing upsampling on the learned features at the end of the network, the network can also learn the class of each pixel in the image, in a process called *semantic segmentation*. We will see this process implemented later on in both Lanenet and Robust Lane Detection. Often this type of architecture is called an *Encoder-Decoder* network. Transposed convolution is an operation which scales a feature map up by overlapping kernel strides and summing the result. It is performed in the decoding portion to upsample features into the image.

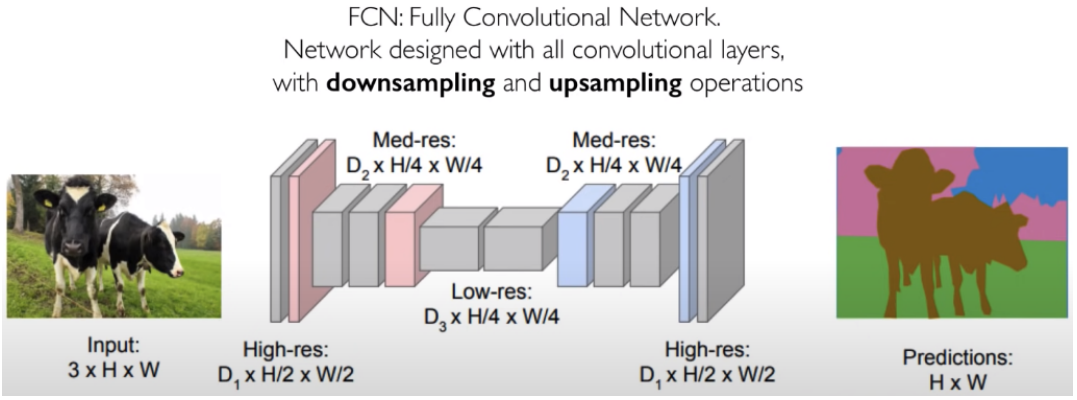


Figure 2.8: An Encoder-Decoder Network [37].

Residual Networks: An important network which we use as a backbone for feature extraction is ResNet. ResNet34 is a simplified version of the residual network architecture, a commonly used deep residual neural network (RNN) which combines a series of convolutional layers for feature extraction in images [40]. Residual networks are not just Convolutional Neural Networks (CNNs) in that, they additionally pass the next layer a residual function by skipping certain activation functions, which passes lower level features further to the next layer. Residual networks are composed of residual blocks, also called skip connections, which pass features from the previous layer over some portion of the network.

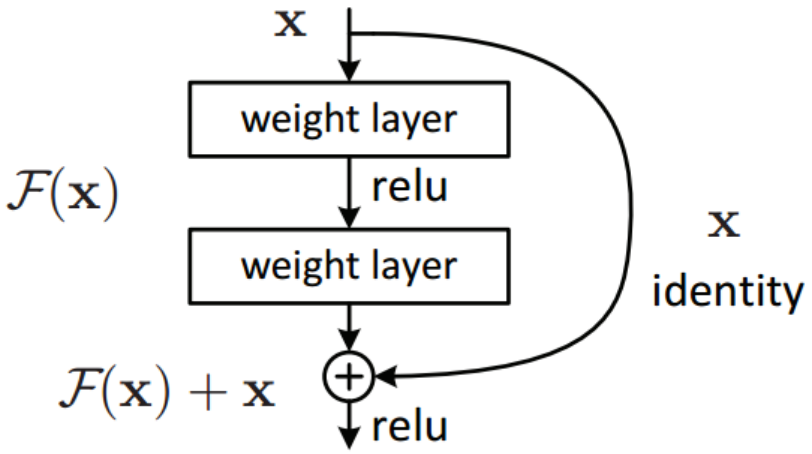


Figure 2.9: A residual block [41].

By passing lower level features throughout the network, Residual networks are able to overcome the infamous *"vanishing gradient problem"* [41] which prevents backpropagation from updating network weights earlier in the network. This innovation enabled new networks to be constructed with significantly more layers, increasing feature detection capabilities.

Methods of teaching neural networks: Random initialization of weights does not always lead to learning something useful. Sometimes these weights propagated through gradient descent based optimization algorithms can converge to non-optimal local minima. The topic of finding out how to ensure the network most optimally learns a separate and useful feature is an open research problem.

There are many approaches to assist the network in converging to a more optimal solution. Optimization functions like gradient descent come in many different flavors which affect the speed of convergence. Stochastic Gradient Descent [30] is a well known algorithm which can be improved by a concept called *momentum* to increase the speed of convergence [42, 43]. This can effectively be thought of as an additional velocity term which allows the learning process to overcome non-optimal local minimums. It works by taking a weighted moving average of the network's gradients to smooth out the oscillations introduced by SGD when updating after each training example. Another improvement to the optimization process is to introduce a variable learning rate, rather than inching forward with a small rate. Adam (Adaptive Moment Estimation) is an optimization algorithm which takes into account both momentum and adaptive learning rates to speed up convergence [44].

Additional techniques which examine separate parts of the training process can have a large impact on how well the network learns. Typically these techniques seek to address improving the model through adjusting the data (*normalization*) or adjusting the model (*regularization*).

Normalization is a technique in which output from one neuron layer is normalized before being the input to another layer of neurons. Normalization generally works by computing both a *mean* μ and *variance* σ values to which all neurons on a layer are normalized against [45, 46]. Lower numerical values generated by normalization have the effect of increasing the speed and accuracy of networks. There are two main types of normalization, being *batch* and *layer* normalization.

Batch normalization calculates the mean of the current neuron layer from all batches of data sampled [45]. This attempts to find a mean and standard deviation which represent the entire dataset. It then normalizes the outputs of all neurons in each layer by each layer's calculated mean.

Layer normalization calculates a mean value for each neuron layer for each data sample. It then normalizes the output of the entire neuron layer with this mean value. This means that each normalized layer in the network has the same mean and variance.

Layer normalization is an improvement over batch normalization and provides several advantages when working with large datasets. One advantage of layer normalization is that it is free from dependencies of batch size. This means the same number of calculations are performed in training time and test time. These two times differ in batch normalization, because performing inference during training in larger batches often utilizes stronger hardware which may not be available in test time. Another advantage is that layer normalization can be used on sequence data enabling the performance increase previously only available to non sequential networks [46]. Although, layer normalization is not as effective on CNNs, which is why our network only uses layer normalization on the fully connected layers.

Regularization works to prevent a model from overfitting to the dataset. Two important types of regularization are L1 and L2 regularization. Both techniques penalize networks if weights are too high by adding the weights into the loss calculation. This is because weights with uniquely high values often exaggerate the importance of some feature in the training set, leading to overfitting. The key difference is that *L1 regularization* (i.e. *L1 norm*) encourages network weights to be sparse while *L2 regularization* (i.e. *ridge regression*) penalizes weights with high values, positive or negative, which keeps network's weights from being high, but not pushing them to zero.

Another regularization technique called *drop out* prevents the model from settling on a local minimum by randomly deactivating neurons in the network. This will prevent the model from relying on some subset of neurons storing the majority of information used for classification. Other techniques such as *data augmentation* are covered in section 4.3.5. Overall experimentation with *hyperparameters* (i.e. the values used to tune the learning process such as learning rate, batch size, etc), architectures, feature selection or better datasets are the best way to ensure a model converges to the most important features for a classification task.

Applying neural networks in autonomous driving has revolutionized the field, making reliable control of the vehicles now possible. The downside is that many of these powerful networks rely on heavy and expensive hardware. This research focuses on networks which have the potential to work on light-weight embedded systems which are useful for cost-efficient consumer vehicles. This research focuses on three papers which yield good results. Namely, Lanenet lane segmentation (2018) [47], Robust Lane Detection (2020) [48] and Ultra-fast Lane Detection (2020) [49] which is based on a highly efficient CNN technique referred to as SCNN (2018) [50].

2.2.2 LaneNet

LaneNet approaches lane detection as an instance segmentation problem. The goal is to label each lane line as belonging to a separate instance of the Lane class. The architecture is a two branch Encoder-Decoder CNN which shares an encoder comprising two out of three stages of E-Net [47] used as the *backbone* (i.e. using a feature extractor network as a foundation for another network). The decoder is split into two branches, one for binary segmentation and the other for instance segmentation.

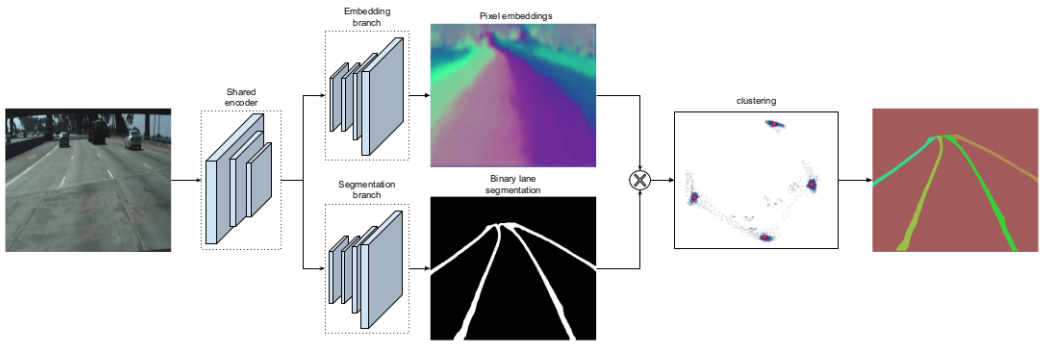


Figure 2.10: LaneNet architecture [47].

The binary segmentation encoder separates lane lines out, coloring lane pixels white, with all other pixels colored black. This binary segmentation

map is used in conjunction with the pixel embeddings output to mask out non-lane pixels, making prediction of lane instance in the next step less computationally expensive. This first branch uses standard cross-entropy loss defined in equation 20. The instance segmentation branch has the goal of outputting an N dimensional pixel embedding which separates the pixels into different instances of the lane class. Pixel embeddings are created by using an iterative clustering loss function which pulls together nearby lane pixels and pushes away lane pixels of other instances. This process and the loss function are detailed in *De Brabandere et al.* [56].

The combination of the binary segmentation and instance segmentation set the foundation for inverse perspective transformation and curve fitting performed by the second Neural Network detailed in the paper. In previous research, a fixed homography matrix (an inverse perspective matrix) would be used to warp the perspective image into a top-down image also called a bird's eye view image [23, 25]. Homography means a perspective mapping of one (image) plane to another. It requires that enough image points present in one plane must be known in the other. This technique allows us to recover the depth of image points which are normally lost in projection. The homography matrix is typically calculated only once using known camera parameters and is error prone under non-flat terrain, which can result in lane points in other images being projected to infinity [25]. LaneNet resolves this issue with a second network referred to as H-Net, that predicts a *conditioned homographic matrix* which transforms points into a top-down view where lane fitting is performed (figure 2.11).

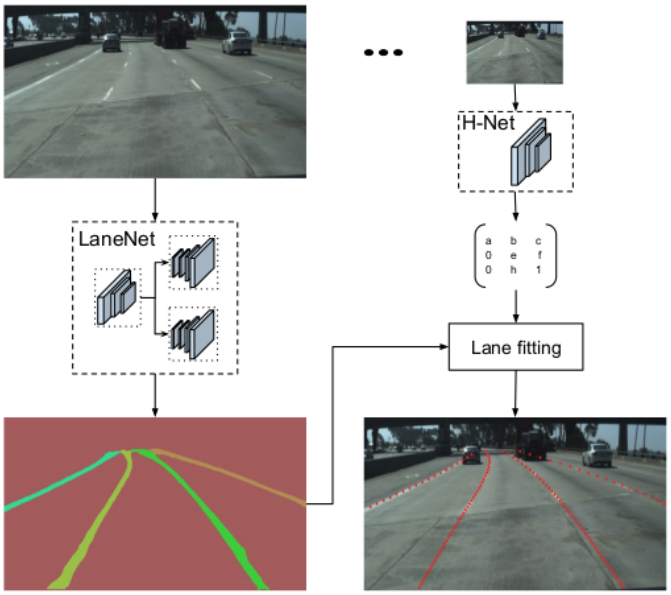


Figure 2.11: LaneNet and H-Net.

The authors specify a custom loss function for H-Net to predict a transformation matrix which optimally fits a polynomial curve to lane pixels. Given *N* ground-truth lane points, $\mathbf{p_i} = [x_i, y_i, 1]^T \in \mathbf{P}$ these points are transformed by the output of H-Net:

$$\mathbf{P}' = \mathrm{H}\mathbf{P} \tag{8}$$

where $\mathbf{p_i'} = [x_i', y_i', 1]^T \in \mathbf{P}'$ and H is a 3x3 matrix containing 6 variables (i.e. 6 degrees of freedom) and 3 constants.

$$\mathrm{H} = \begin{bmatrix} a & b & c \\ 0 & d & e \\ 0 & f & 1 \end{bmatrix} \tag{9}$$

Once points are projected, a polynomial $f(y') = \alpha y'^2 + \beta y' + \gamma$ is fit using the least squares closed-form solution:

$$\mathbf{w} = (\mathbf{Y}^T\mathbf{Y})^{-1}\mathbf{Y}^T\mathbf{x}' \tag{10}$$

with $\mathbf{w} = [\alpha, \beta, \gamma]^T, \mathbf{x}' = [x_1', x_2', \ldots, x_N']^T$ and $\mathbf{Y}$ is a Nx3 matrix containing the vertical pixel positions of the polynomial variables with N being the order of the polynomial.

$$\mathbf{Y} = \begin{bmatrix} y_1'^2 & y_1' & 1 \\ \vdots & \vdots & \vdots \\ y_N'^2 & y_N' & 1 \end{bmatrix} \tag{11}$$

Regarding the case using a 2nd order polynomial, the fitted polynomial is evaluated at every y_i' location $x_i'^* = f(y_i')$ yielding a horizontal pixel position prediction $x_i'^*$. Afterwards the prediction is projected back by the inverse of H and compared with the label in image space x_i using least squares. The sum of the difference between the original and the fitted point transformed back into image space is used as the measurement for the loss.

$$Loss = \frac{1}{N}\sum_{i=1,N}(x_i^* - x_i)^2 \tag{12}$$

These constants used for ensuring horizontal lines in the original image remain horizontal in the warped image. The output matrix is then used to warp a downsampled original image into bird's eye which works under variable terrain, ensuring that all lane points are correctly warped onto the image. The reason for this is that it is easier to fit lower order polynomial curves onto the warped images as their points are spread equidistant, making curves wider and therefore easier to approximate. The fitted polynomials are then warped back into the original image space, correctly separating lane lines. Ultimately using both LaneNet and H-Net in tandem results in accurately fitted curves produced at 50 fps on an Nvidia 1080 ti. With an accuracy of 96.4%, LaneNet achieved 4th place in accuracy on the TuSimple dataset. Speed of the other winning models is not provided.

2.2.3 Robust Lane Detection

Another paper critical to our research is Robust Lane Detection. Their research team's key insight was to pivot from single image lane detection to a continuous image sequence based detection. The main theory behind their approach is that lanes are fundamentally continuous structures and therefore having only a single image to predict a continuous structure does not provide enough information. By incorporating information from previous frames, the model will be capable of filling in the blank when the lanes (the continuous structures) are obscured or occluded. The model should be capable of predicting lane structure from past information when it is obstructed in the current frame. Due to this idea, *Zou et al.* [48] structure their model architecture as follows:

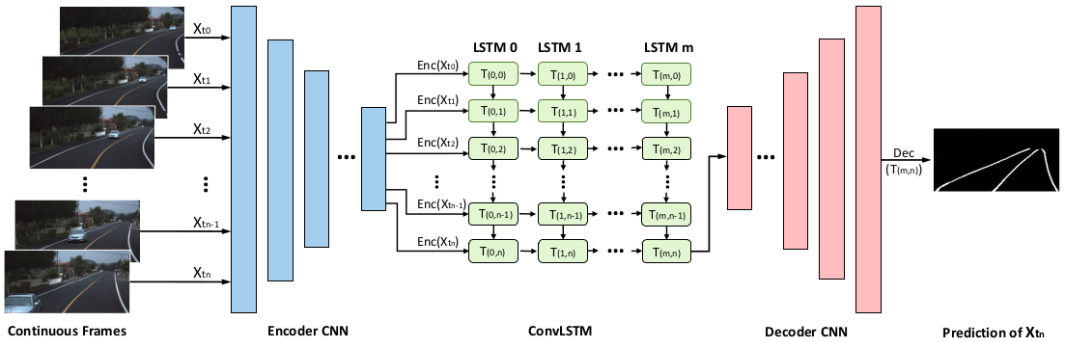


Figure 2.12: Robust Lane Detection Architecture [48].

The architecture involves a fusion of a classic Encoder-Decoder CNN for semantic segmentation, with the center being a Long-Short-Term Memory

(LSTM) Recurrent Neural Network (RNN). Using RNNs on images previously would be extremely costly as images represented as vectors would be of size WxH so even small images (ex: 512x256) would create too large vectors for passing information. The encoder resamples images to size 256x128 and this size is replicated in the decoder, producing the same size output prediction.

U-Net and SegNet introduced here [57] combined with ConvLSTM act as the backbone for the model. ConvLSTM was introduced in [58] and is different from traditional LSTMs in that all matrix operations on the gates are converted to convolution operations, decreasing the size of computation. Instead, by attaching the LSTM to the end of the decoder, the LSTM works given image features of size 8x16x512 when using UNet-ConvLSTM or 4x8x512 when using SegNet-ConvLSTM. Since this process is repeated for a continuous image sequence (in this case, 5 images at a time), the LSTM is able to incorporate multiple lane features from the images and make a robust prediction. The loss function is based on the weighted cross entropy for the purposes of solving discriminative segmentation tasks. It is defined as:

$$\varepsilon_{loss} = \sum_{\mathbf{x}\in\Omega} w(\mathbf{x}) log(p_{l(\mathbf{x})}(\mathbf{x})) \tag{13}$$

Where, L : Ω → {1, . . . , K} is the true label of each pixel and w : Ω → R is a weight for each class. Additionally p represents the softmax function, defined in equation 7.

The authors of RLD contributed to the TuSimple lane dataset by additionally labeling every 13th image, where the previous dataset only had every 20th image labeled. Additionally, the authors expanded the dataset by introducing 1,148 sequences of rural roads with the goal of increasing the diversity of the dataset.

The results of the dataset demonstrate that the combination of U-Net-ConvLSTM produces the most robust results as opposed to SegNet-ConvLSTM. U-Net-ConvLSTM achieves validation accuracy of 98.52, test accuracy of 98.43 in rural scenes and 98.00 accuracy on highway scenes. Other metrics include Precision of 0.857, Recall of 0.958 and F1 of 0.904. These metrics are defined in section 4.2.

The authors are able to achieve state of the art performance on images which include obstruction, occlusion and heavy shadows while running at 5.8ms on inference on two GeForce GTX TITAN-X GPUs. While the results of Robust Lane Detection are worth noting, the speed of

inference is significantly slower on embedded systems, pushing the model out of use for production purposes. With this constraint in mind, we instead opted to leverage Robust Lane Detection and clustering for auto-labeling frames to use on smaller models. This process will be explained in further detail in the methodology section.

2.2.4 Ultra Fast Lane Detection

Seeking to solve the "no visual clue" problem, Ultra-Fast Lane Detection [49] opts to make strategic optimizations to reduce computational cost on their model. It uses two core approaches to speed up lane detection while achieving state of the art performance in accuracy. The first is using row-based selection to predict positions of lanes in the image along pre-set row anchors, rather than using a more powerful encoder-decoder architecture to segment the entire image. By manually biasing where the model has the highest probability to find the lane markings, computation can be saved instead of using semantic segmentation across the entire image.

Row-based selection works by setting a predefined number of horizontal anchors h across the image which are divided into grid cells w that are searched to detect C lanes. This method drastically reduces the size of the model to search through compared to image-segmentation which searches the entire image. In practice on the TuSimple dataset, UFLD creates 55 row anchors placed every 10 vertical pixels. Row anchors are divided into 100 grid cells with up to 5 lanes. The second method is using SCNN's strategy of cutting out any 0 multiplication calculations, which are often a product of the ReLU activation function [50]. The combination achieves over 300 fps on an Nvidia 1080Ti with their lightest model. The model is trained and tested on both the CULane and TuSimple Datasets.

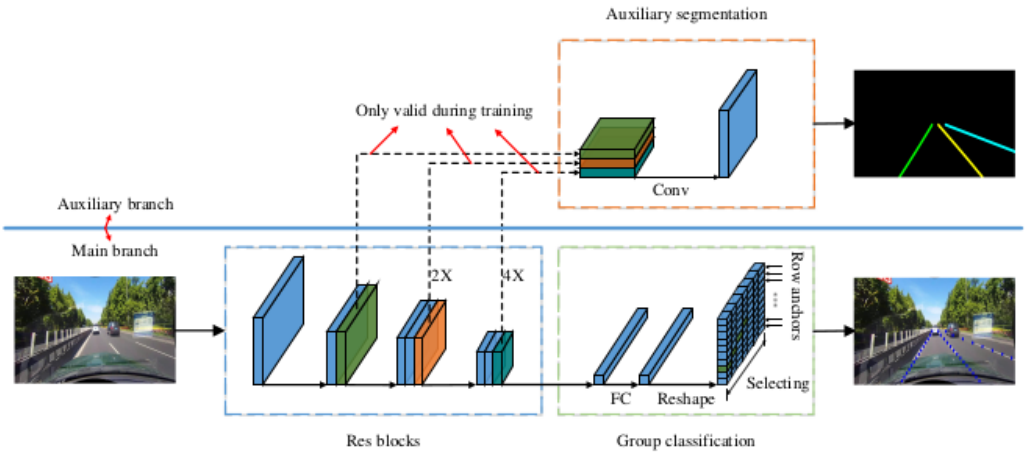


Figure 2.13: Ultra-Fast Lane Detection model architecture [49].

Multiple loss functions are used throughout different parts of the model architecture to hierarchically predict different features of lane detection. Their classification loss is the sum of the cross entropy loss *LCE* across *C* (the maximum number of lanes) and *h* (the number of row anchors).

$$L_{cls} = \sum_{i=1}^{C} \sum_{j=1}^{h} L_{CE}(P_{i,j,:}, T_{i,j,:}) \tag{14}$$

Where P i,j,: is the (w + 1)-dimensional vector representing the probability of selecting (w + 1) gridding cells for the i-th lane, j-th row anchor, while T i,j,: is one label of correct locations.

Their second loss function, Lane Structural Loss, is composed of two loss functions: Similarity Loss (*Lsim*) and Shape Loss (*Lshp*). Similarity Loss:

$$L_{sim} = \sum_{i=1}^{C} \sum_{j=1}^{h-1} ||P_{i,j,:} - P_{i,j=1,:}||_1 \tag{15}$$

addresses the fact that lanes are continuous structures and works by constraining the distribution of classification vectors over adjacent row anchors. Shape Loss:

$$L_{shp} = \sum_{i=1}^{C} \sum_{j=1}^{h-2} ||(Loc_{i,j} - Loc_{i,j+1}) - (Loc_{i,j+1} - Loc_{i,j+2})||_1 \tag{16}$$

constraints lane shape to the second order difference equation, being 0 when lanes are straight. Finally, they introduce an auxiliary segmentation loss (*LSeg*), which is just cross entropy loss, giving the total loss:

$$L_{Total} = L_{cls} + L_{str} + L_{seg}. \tag{17}$$

With this combination of model optimizations, loss functions and a modified Resnet18 backbone, UFLD accurately predicts multiple lanes in challenging scenarios (including curved roads, crowded streets and night driving) in 3.2ms on a single Nvidia GTX 1080 Ti. The model has a classification accuracy of 95.77 - 96.06 on the TuSimple benchmark.

2.2.5 Summary of Neural Network Based Methods

Lane detection methods for the most part share the common strategy of defining lane detection as an image segmentation problem. Even in the case of Ultra-Fast Lane Detection, the authors still use some degree of (auxiliary) segmentation. With this in mind, it means that the architectures rely on a modified form of the "Encoder-Decoder" architecture. Ultimately each experiment uses either Semantic Segmentation, Instance Segmentation or Auxiliary Segmentation.

Due to their similar segmentation approach, both LaneNet and Robust Lane Detection have the same issues of slower speed at the trade-off for higher accuracy. LaneNet has an advantage over Robust Lane Detection in the sense that lanes can be differentiated by their class instance. Maintaining a consistent instance for each lane allows the autonomous vehicle to change lanes while keeping a stable detection. In the end, Ultra Fast Lane Detection comes out on top, as it is capable of comparable accuracy to LaneNet and Robust Lane Detection while having many multiples the speed of inference. Additionally UFLD is capable of classifying the lane lines in a similar manner to LaneNet.

2.3 Implementing Lane Detection Methods on Embedded Systems

In research, lane detection is typically simulated on personal computers rather than directly on embedded hardware due to tighter constraints than what is typically available in academia. The problem with this is that it leads to cutting edge robust lane detentions being too slow to be practical for commercial use [51]. This means that older, less robust detection systems which rely heavily on hand-crafted feature filters have to be used, limiting the range of scenarios vehicles can drive safely. Research such as the real time mobile lane detection system by *M.J. Jen et al.* [52] resorts to running hand-crafted filters on embedded hardware. Other improvements including PathMark by *Q. Ju et al.* [51] continued to improve detection using image intensity and geometric matching of detected lane segments to full lanes. Gradual advances in hardware brought classic machine learning techniques back into the picture. *N. Mechat et al.* [53] used Support Vector Machines (SVMs) to classify lanes and then fit curves for better predictions. Often relying on a single camera for lane detection, sensor fusion techniques such as the Kalman filter were used to give the control system better perception [53]. Additional uses of combining both the Kalman filter and IPM techniques include Y. Lee and H. Kim's [25] research which achieves real

time performance for highway and urban driving scenarios. This technique uses a single camera and focuses on ego-lane detection only. M. Neito et al. [54] further improves classification method based lane detection by providing an end-to-end lightweight pipeline which is built to combine many different computer vision techniques for maintaining stable detection. They focus on geometry based methods that are capable of detecting multiple lanes.

Recent improvements in embedded hardware computational speed has opened up the opportunity to use lightweight neural network based lane detection. Inspired by UFLD's row-wise classification technique, SwiftLane by *O. Jayasinghe et al.* [55] tackle lane detection on the Nvidia Jetson AGX Xavier embedded system. They further increase inference speed by reducing the number of false detection, which manages to achieve 56 fps, rivaling our solution.

3. Methodology

3.1 Background

In 2019 the European Commision enacted a law requiring new automotive safety features to be implemented in cars launched after July 5th of 2024 [59]. Safety features required by this mandate include Advanced Emergency Braking (AEB) and Lane-Keeping Assistance (LKA). Our team developed a pilot for a major European OEM which included a convolutional neural network based lane detection model as it is the foundation for *AEB* and *LKA* features.

The main requirements of the pilot are a model that runs in real time on an automotive-grade SOC and meets the Operational Design Domain (ODD) standard of L2+ ADAS. OEMs need small systems that can easily fit in the dashboard. This places a significant constraint on the amount of computation that can be used for reliable lane detection models. We examined these constraints on hardware and software to find a balance which meets the requirements of the pilot.

In regards to hardware, we took a similar approach to [55] by choosing the Nvidia Jetson AGX Xavier as our SOC. It is a powerful embedded system useful for deploying end-to-end AI and robotics applications [60]. We plan to downscale further to the Nvidia Jetson Nano [61]. After testing we expected a 10-12x downscaling in fps when testing models on an Nvidia GTX 3080 notebook moving to the Xavier and 15x downscaling to the Nano.

Inference speed is a key factor that eliminates any model incapable of running at very high FPS. This is because the model needs to predict lanes fast enough that the control system can steer the vehicle safely at high speeds. Downscaling to an embedded system slows down inference significantly. Making a control robust system requires more than just lane detection. 3D vehicle detection, pedestrian detection, objects on road detection and other features are critical to maintaining a safe driving experience for passengers.

In regards to the models presented in the literature review, we found that LaneNet and RLD require a combination of powerful GPUs while only attaining slightly higher than real-time fps. LaneNet sought to address the issue of poor prediction when switching lanes using instance segmentation; we found that given sufficient data, UFLD is more than capable of reliably switching lanes. UFLD most notably significantly outperforms these other methods and will most likely scale down well. It was rigorously tested in several scenarios that feature high curvature, missing lane markings and

night driving. Although the accuracy will vary in these scenarios, we believe it will remain high enough for autonomous control. Moreover, when testing UFLD on an Nvidia GTX 3080 notebook we were able to obtain over 500 fps in inference when using the ResNet 18 backbone. Expecting to achieve between 50-80 fps with ResNet 18 on the selected embedded systems, this still leaves room for additional tweaks and modifications to increase performance for our purpose.

For the reasons above, our team adopted UFLD's methodology and tuned its architecture to fit our key purposes. See section 2.2.4 Ultra Fast Lane Detection for more details on the representation of lanes and use of loss functions to learn lane structure. How lanes are labeled is covered in section 3.4. Using row-based selection of lanes as a foundation, we further optimize for speed by reducing the total number of row anchors for the network to learn from 48 to 14. One important thing to note is that, unlike UFLD, our goal is only to predict the ego-lane (i.e. the lane the vehicle is driving in) for lane-keeping assistance features. UFLD detects a maximum of five lanes, while we reduce the number to three, which is the minimum for changing lanes. Experiments with different tiers of ResNet will likely improve performance while pushing fps down closer to just above real-time inference.

Although we chose not to use LaneNet or RLD, the models were not entirely disregarded. Although RLD was capable of running at real-time on high end systems, downscaling the model would yield sub-real time speeds, which eliminated the model from being used on the chosen embedded system for the pilot. RLD was found to have a very reliable detection and segmentation processes that could be used to label many images very quickly in a process referred to as auto-labeling. This process will be detailed further after the section on manual labeling.

3.2 Model Architecture

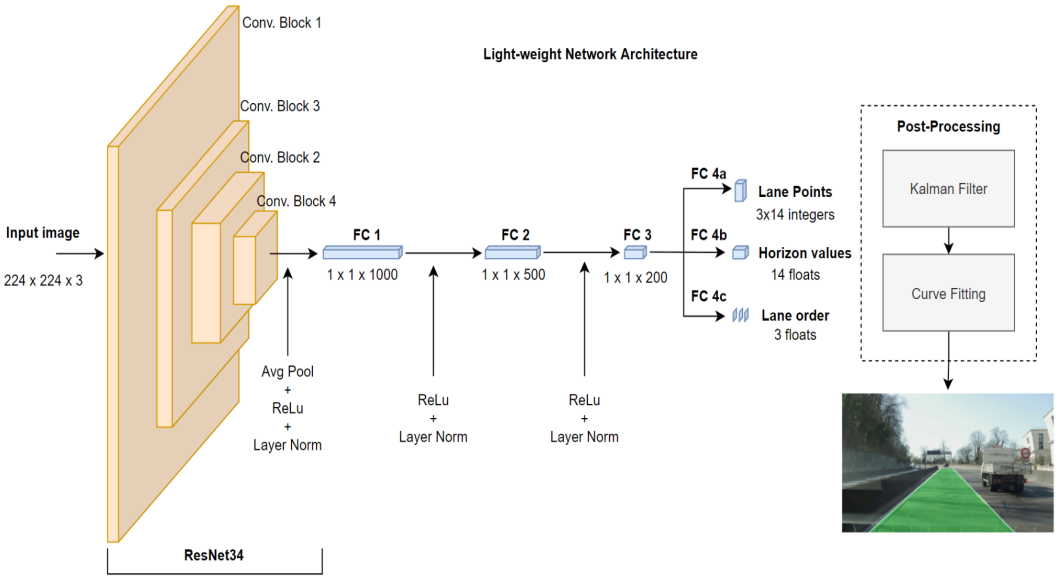


Figure 3.1: VahidNet model architecture.

As noted in section 3.1 we based our model architecture on a simplified version of UFLD. Overall we reduce the input image size, number of lanes predicted and number of lane points to speed up computation. The model's architecture can be summarized as follows:

- Input a 3 channel RGB image of size (224x224x3)
- Backbone (ResNet18 or ResNet34) → outputs a 1000 fc layer. #1
- ReLU and layer normalization (1000)
- Fully connected layer 2 (1000 → 500) → transforms into 500 length vector
- ReLU and Layer normalization 2 (500)
- Fully connected layer 3 (500 → 200)
- ReLU and Layer normalization 3 (200)
- Fully connected layer 4 (Multi-headed output):
    - Part A: size 200 → size (3x14) lane points
    - Part B: size 200 -> size (14) horizon values
    - Part C: size 200 -> size (3) lane order

Beginning with the model backbone, we choose to build upon the ResNet18 and ResNet34 [62] architectures, which as noted in UFLD are able to significantly enhance prediction accuracy while maintaining a good degree of speed. We chose these two backbones due to speed constraints and found

that both are able to maintain real-time inference on the Nvidia Xavier embedded system. The ResNet backbone provides the learned features we use for performing lane detection. ResNet34 outputs a fully connected layer as a 1000x1 length classification vector.

Our own model is quite simple due to time constraints of the OEM pilot demonstration. We began by taking the 1000x1 length vector output by ResNet34 then performing layer ReLu activation and layer normalization on it. Next, the feature vector is further downsized to a 500x1 length fully connected layer, then performing ReLu activation and layer normalization again. Afterwards the fully connected layer is downsized once more through a 500x1 to 200x1 length layer which is also passed to ReLu activation and then layer-normalized. Finally, a forth fully connected layer is created in 3 steps:

a. The size 200 layer is linearly transformed to a 3x14 sized fully connected layer. This 3x14 vector is a combination of the number of side candidates (3) and the number of side coordinates (14).
b. Linearly transforming the layer size 200 to 14.
c. Linearly transforming the layer size 200 to 3.

This fourth fully connected layer acts as a multi-headed output, giving 59 floats in total. Part A of the process is used for 3 lanes each with 14 x-coordinate positions at specified anchor points as mentioned earlier. This output is used by the Lane Loss Function. Part B outputs 14 floating point values which are later used in Horizon Loss to determine if a given anchor point is above or below the horizon. Finally part C gives 3 floating values which represent the probability that the lane is made up of two of the three candidate lanes. This conversely is used in Lane Selection Loss. When the model is not training, the outputs are fed directly into the post-processing phase, which visually assembles the lane.

3.3 Loss functions

Our approach to lane detection can be conceptualized by splitting the problem of detecting the ego lane into 4 components. We first determine the 2D lane coordinate points for assembling up to three candidate lane edges. In total we find 42 x-coordinate lane points (3 lanes of 14 horizontal positions). Next the model learns which points belong to which lanes. Following this, the model learns the height of the horizon. Finally these three outputs are combined together in post-processing to remove any lane predictions which appear above the horizon.

This method of learning lane detection can be broken down by making 4 separate loss functions for each component of detection. We sought to make the model learn the following:

- *Lane Loss (L1)* learns the 2D coordinates which make up a lane line. It uses *Mean Squared Error* (MSE) of 2D lane coordinates.
- *Lane Selection Loss (L2)* learns which lanes to select from and relies on *Cross Entropy Loss*.
- *Horizon Loss (L3)* learns the location of the horizon in the image. This loss relies on a variant of Cross Entropy called *Binary Cross-Entropy*.

Finally we use a separate loss function for determining lane order when there are three lane candidates. This *Lane Order Loss* function (L4) helps the model determine which two lanes should be used for the ego-lane and additionally helps the model understand how to switch lanes. See figures 3.2, 3.3 and 3.4 for reference. In the following section, a more detailed look into why the given loss function is useful and what data it uses.

Lane Loss (equation 18) learns 42 x-coordinate points or 3 lanes consisting of 14 different x-coordinate values at predefined heights using MSE. Our labeling method is a simplified version of TuSimple which reduces the amount of lane boundary positions as shown in figure 3.3.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{18}$$

Where, Y_i is the labeled value of some x-coordinate, $\hat{Y}_i$ is the value predicted by the model given the labeled image and n is the number of data points (in this case 42).

Effectively MSE measures the average squared distance between predicted values and the ground truth label. The distance being squared has the effect of dramatically increasing the loss value when the prediction is very incorrect. In turn this has the effect of increasing the derivative in the direction away from the loss during optimization, making the model learn faster. The MSE is an effective way of penalizing highly incorrect predictions, allowing the model to learn from mistakes quicker.

MSE is useful for learning a task such as finding the x-coordinates of a lane because we are interested in minimizing the error between where the lane has been marked and where the model predicts it will be. Fundamentally this is a MSE problem because minimizing the distance between where the lane actually is and where the model predicts it, is useful

for getting the model to learn the positions of where the lane lies. One idea for an improvement is to find some function such that points are aware of the relative position of their neighbors. This could lead to an understanding of some approximation that from one point to another, the *x* positions should not change so drastically.

Second, we cover the Lane Selection Loss. This loss function outputs 3 floating-point values which give the probability of the possibilities for which lane lines will be used to assemble the ego-lane. These values are used later in the post-processing phase to display the ego-lane. For Lane Selection Loss, we choose to use Cross-Entropy Loss, a common approach for interpreting multiple output probabilities of neural networks. Cross-Entropy loss has many other names including “Logistic Loss” and “Multinomial Logistic Loss”. It is the negative natural log (log base e) of the softmax function see equation 7. This equation can be represented in the form:

$$CE = -\sum_{i}^{C} t_i \log\left(s_i\right) \tag{19}$$

This form is most often used as it resembles the entropy equation in physics. For descriptive purposes, a slightly expanded form of the equation is used for two reasons. First is that the log present in this equation is actually log base ***e*** (natural log), not log base 10. Second is that the term s_i is the result of the softmax equation on the ground truth label y_i.

$$CE = -\sum_{i}^{C} t_i \ln\left(\mathrm{softmax}(\mathrm{y_i})\right) \tag{20}$$

where i is the index of the current output value, C is the total number of output values, t_i is the ground truth label of the output. Y_i is the feature vector input in softmax see equation 7. Softmax has the effect of taking multiple input values and mapping them to sum to 1 which can be seen as a probability distribution of the output. Wrapping the negative natural log around the softmax function has the effect of dramatically increasing the loss of bad predictions, while nearly linearly decreasing loss for good predictions. Ultimately this prevents overstepping in backpropagation while using our optimization method when the loss is low and quickly moving away from the previous result when the loss is high. When loss is high from a prediction, yielding a high loss function and therefore high derivative

value can be useful for ensuring our approximation moves far away from an area of local minima. When loss is low, we often want our approximation to change slower, as to not overstep an approximation which minimizes loss. In a way, it has a similar effect as *mean-squared error* except applied in different cases with different input data.

The next component to address is Horizon Loss which employs Binary Cross Entropy. This type of Cross Entropy is useful for making several yes or no decisions at once [63, 64, 65]. In this case we use 14 floating point values which predict the probability for the horizon by determining if a given horizontal anchor is above or below the horizon. Again, refer to figure 3.3. Binary Cross Entropy functions similarly to normal Cross Entropy, except that it utilizes two classes:

$$BCE = -\sum_{i}^{C=2} t_i \log(s_i) = -t_1 \log(s_1) - (1 - t_1) \log(1 - s_1) \tag{21}$$

Where, t_1 and s_1 are the ground truth classification label and score for Class 1 - above the horizon (C1) and $t_2 = 1 - t1$ and $s_2 = 1 - s_2$ are the ground truth label and score for Class 2 - below the horizon (C2).

An important drawback to this loss function is that the actual horizon often falls between the boundary where classification of above or below the horizon changes. Any points classified as part of the lane are later discarded in the post processing phase if they fall above the horizon. Due to the perspective effect there can be a long section of the road between the anchor just below the horizon and just above it. Having a hard threshold at the former anchor reduces the amount of lookahead to the road we can use for controlling the vehicle.

The final component of lane detection is the Lane Order Loss (L_4) which again uses MSE to determine if the lanes are in the wrong order, otherwise the value yields 0 in cases where lane order is correct. The total Loss Function is a combination of the previous four loss functions, modeled:

$$l_1 + w_2 l_2 + w_3 l_3 + w_4 l_4 \tag{22}$$

For the experiment we chose the *Adam optimization* algorithm [66] rather than *Stochastic Gradient Descent* as time was a major constraint and it was leading to faster convergence to local minimums.

3.4 Data Labeling

A large dataset of labeled images depicting diverse driving scenarios will have a huge impact on the performance of a trained model. Overfitting to specific scenarios which are overly abundant in homogenous datasets, such as straight highway driving on sunny days, often leads to poor performance in the real world. The lane detector must be robust to changes in weather or lighting as the driver's safety is at risk when a model encounters an unfamiliar scenario.

Overfitting is a problem in machine learning that occurs when a model fails to generalize features learned from a dataset and instead learns features specific to a subset of the data. This leads to poor performance in real world scenarios, as overfit models often fail to learn features present in a diverse number of scenarios. Overfitting is often addressed by splitting a dataset into training, testing and validation sets. The goal of each subset is to capture an equal distribution of features which will be found evenly across the entire dataset. The distribution of features present in each subset is very important and has a great effect on performance. Other methods of resolving overfitting were addressed in section 2.2.1. This even distribution can be ensured by careful pruning of what is included in the dataset. As an example, if the training set only includes straight driving scenarios on a sunny day, but the test set only includes highly curved roads on stormy days, the model will probably perform poorly.

Covering a range of diverse scenarios ensures models learn general features. The downside is that it is impossible to create a dataset diverse enough to cover all possible situations where the model will learn to handle the situation perfectly. The drawback is that the model can only generalize situations it encounters often, meaning it will never fully understand how to handle all situations. The goal is to ensure that the model performs consistently enough that it will work with a high degree of accuracy in nearly all situations.

Many datasets are available for public research for non-commercial purposes only. Because of this constraint, we opted to create our own dataset for training the model and to use the TuSimple dataset [67] for training and evaluation, as it is available for commercial use. TuSimple is widely used as a basemark for testing lane detection models and consists of 6,408 road images of US highways with a resolution of 1280x720. These images are organized by sequences of 20 frames. Out of this dataset, about 55% of all images are used for training with, 40% used for testing and 5% for validation. The images include scenarios of different weather and lighting conditions. TuSimple is a public dataset which was released during

their Lane Detection challenge in 2017 [67] for the *Conference on Computer Vision and Pattern Recognition* (CVPR).

Other common lane detection benchmarks such as CULane, that contains over 55 hours of driving in diverse and challenging scenarios, were not available because they were restricted to non-commercial use [68]. This means that it becomes more difficult and costly to make a reliable model as not enough data is present to generalize lane detection with just a single, relatively small dataset. With this in mind, our team opted to create our own dataset to train the model, containing over 20 hours of driving in diverse scenarios across multiple European countries. Several of these scenarios include inner-city driving, full of traffic lights, complex roads and pedestrians.

3.4.1 Labeling

Developing and testing the model was done using the labeling method employed by the TuSimple Dataset [67]. This labeling method is as follows: Construct 48 evenly spaced horizontal lines which span from the bottom of the image to a predefined height. Mark the lane line locations x position for any lanes present. Any x position of the current horizontal line that is not placed on a lane is marked as -2 to differentiate it from actual pixel values. Follow this procedure for every 20th frame of each image sequence. These lane markings are stored in a json file format which includes the pixel heights of horizontal lines.

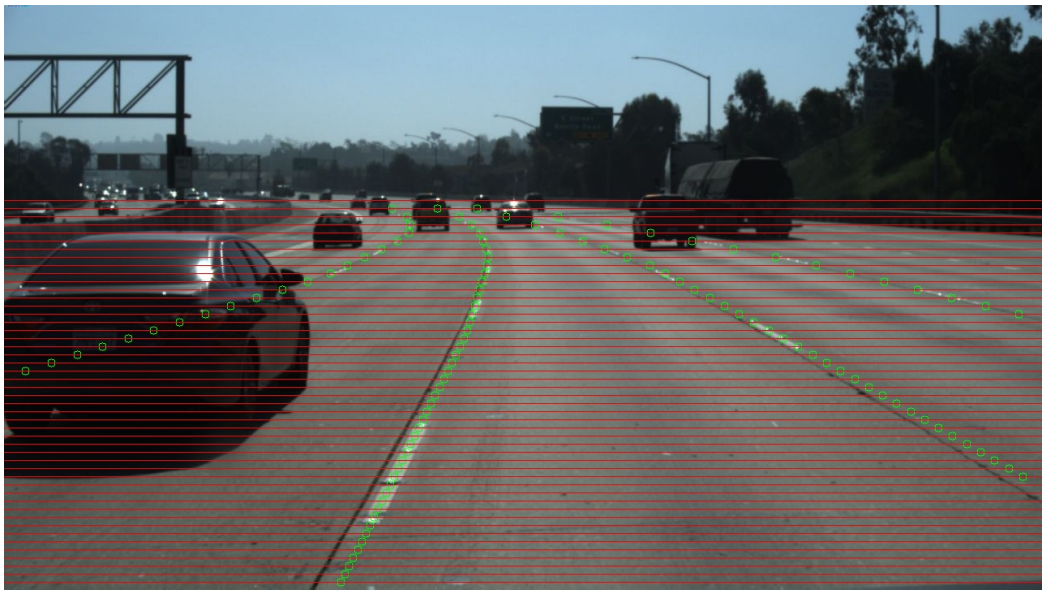


Figure 3.2: TuSimple labeled image [67].

Figure 3.3: Manually labeled image in modified TuSimple label format.

In regards to labeling our own data, our team chose to label every 13th frame while annotating significantly less horizontal lines (14) and x positions in image space (3 per row anchor). Our target is simple, only needing to detect the ego-lane (the current lane in which the vehicle is driving), allowing us to only label 2 lanes. Additionally, to increase lane annotation speed and lower costs, we built an in-house data annotation team and provided them custom annotation tools which mimicked the TuSimple method.

3.4.2 Auto-labeling

While our annotation team was manually labeling frames, we additionally experimented with an auto-labeling approach. Inspired by Tesla's strategy of using the actions of the driver to label what the model should consider ground truth in a scenario, we used Robust Lane Detection to generate image labels to train the model with. After experimenting by combining the prediction with curve fitting and then clustering, our pipeline was able to rapidly generate labeled images on previously unseen data.

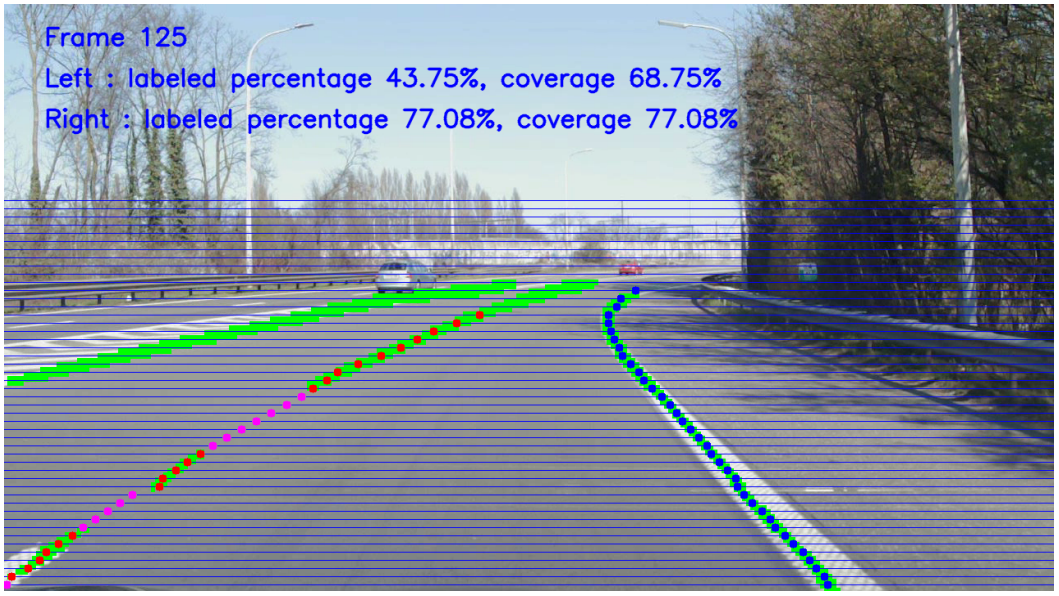


Figure 3.4: An auto-labeled image with Robust Lane Detection, clustering and line fitting.

The downside of this method was that results were inconsistent and had to be manually verified. Fixing auto-labeled annotations with slightly incorrect points proved to be a non-trivial task, which often resulted in many images taking longer to correct than simply manually labeling the image or being outright rejected. Given these issues, auto-labeling was an interesting experiment that could be improved by using a more accurate model than Robust Lane Detection. Swapping RLD for a better model would result in higher quality labels that are generated more reliably, enabling lighter-weight models to yield similar results. Despite the difficulties of applying auto labeling in a short timeframe for the present project, the technique seems promising and worthy of more research effort in the future.

3.5 Post-Processing

After the model outputs the prediction, we use a Kalman filter, also known as *Linear Quadratic Estimation* to smooth lane point predictions. The Kalman filter is an optimal estimation algorithm invented by Rudolf E. Kalman in 1960 [69]. This filter is often used for target tracking, as it provides a statistical method to predict the next position of an object given previous position data from multiple sources. It is extremely useful when combining approximate info provided by multiple sensors (such as IMU, GPS, etc) which inform our model about its position and velocity relative to the world. Each sensor or model provides some noisy value of where the

vehicle is located. It works by estimating unknown variables based on uncertain measurements given by measurement devices. Since we intend to combine data from sensors and approximations provided by our model for controlling the vehicle, it is critical to have an accurate idea of where the vehicle is relative to the road. Applying the filter, gives the vehicle a more informed idea of its position and velocity by finding where the approximations provided by each sensor overlap. This helps eliminate noise from each sensor.

Although we opted to use the Kalman filter alongside our lane detection model, we found it often made lane departure more difficult to detect as the filter would heavily influence the model to stay in the lane provided past data. After the predicted points are filtered, we use a 2nd degree polynomial curve to smooth lane points and display areas within the two lanes curves in green. We chose 2nd degree polynomials because higher order polynomials have overly complex curves which fail to approximate the lane line near the horizon.

4. Experiments and Results

4.1 Datasets

Since the model was produced as a pilot for a major OEM, many datasets are only available for academic research. For this reason, we chose to compare our model to others using the TuSimple dataset [67], since it is open to commercial use. TuSimple was released by the autonomous truck driving company of the same name. This dataset contains 6,408 images of roads on US highways and was made available during the CVPR2017 conference. These image sets are made of many one-second-long clips of 20 frames each of resolution 1280x720. Overall the dataset mostly contains image sequences with good or medium weather conditions, during the day, containing 2 or more lanes in several traffic conditions. Upon release, TuSimple proposed two challenges: A Lane Detection Challenge and a Velocity Estimation Challenge. As part of the challenge, they provide labels for every 20th frame for each sequence, which contain x coordinates for each lane (up to 5) at 48 different vertical positions. The vertical positions are marked by horizontal line anchors which are consistent for every image. Finally, a -2 value is set for positions on the anchors which do not overlap a lane or positions where no lane marking exists [67].

In an effort to expand the diversity of the TuSimple dataset, the Robust Lane Detection team provides additional labels for every 13th image in each sequence. Moreover, they include 1,148 image sequences of rural roads in China [48].

CULane is another dataset widely used for benchmarking lane detection models, but unfortunately is only available for education or non-commercial research [68]. Our team chose to omit testing and evaluating on this dataset to avoid any commercial conflicts.

Additionally, we recorded 20 hours of continuous driving through several European countries, including France, Belgium, the Netherlands and Finland to test and train our model to be fit for European roads. This private dataset has a wide variety of driving conditions, including inner city driving with pedestrians and highway driving with different weather and illumination conditions. Our dataset contains labels of every 13th image, with labels constructed in a similar manner as TuSimple. Rather than having 48 horizontal lane anchors, we use 14 to decrease labeling cost. We use less points to increase model inference speed by only needing to predict a maximum of 3 lanes x 14 points.

4.2 Evaluation Criteria

A variety of metrics are used by researchers to evaluate the effectiveness of machine learning models. Most of these criteria use a combination of True Positive (**TP**), True Negative (**TN**), False Positive (**FP**) and False Negative (**FN**) counts to come up with a single numerical value to determine how well the model is performing. Among these criteria are *Precision*, *Recall*, *F1* and *Accuracy*, with the latter being a value for human readability. Understanding these four values can be depicted in a *Confusion Matrix* [70, 71, 72]:

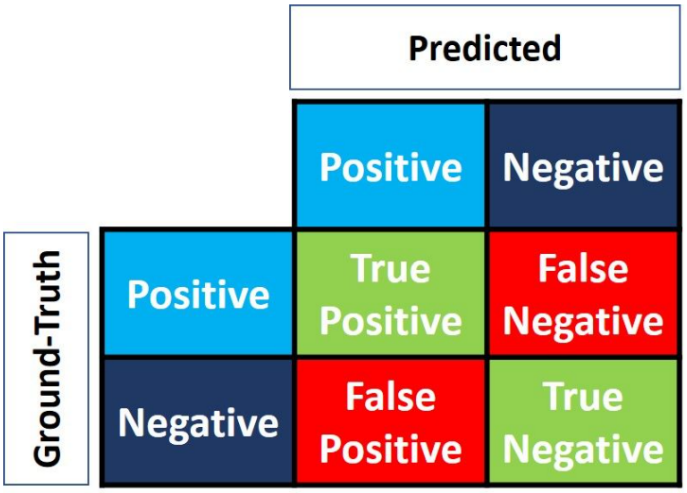


Figure 4.1: A confusion matrix [72].

Precision, also called *positive predictive value* (equation 23) gives an idea of determining how reliable a given model can classify something as positive.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{23}$$

Relying on precision as a metric allows us to determine whether or not we can trust the prediction to be accurate. *Recall*, also known as *sensitivity,* requires all positive labels and positive predictions. This is because recall measures the model's ability to detect positive examples.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{24}$$

A perfect model would produce no false negatives, resulting in a recall of 1.0. In practice it is generally not possible to have both high precision and recall as they often require tradeoffs between what is considered false positive or false negative. Often, having too high precision and recall can inform us that either the model is too complex (i.e. computationally costly) or overfit to the trained data [71]. A model which has 0 false negatives or positives is worth scrutinizing. It is important to utilize both precision and recall in evaluating the effectiveness of our model because having only a single metric can be deceiving on a case by case basis, meaning they are not always reliable.

Another metric for evaluating predictive performance is F1-score (equation 25). It is useful for evaluating the effectiveness of multi-class classifiers because it helps us find a balance between precision or recall. It works by combining precision and recall, which are often competing metrics [73]. Increasing precision can decrease recall and vice versa. As F1 is dependent on precision and recall, it is not perfect in that it does not account for True Negative (TN) detections, which can be an issue in certain situations. F1 ranges from 0 to 1, with values closer to 1 being better.

$$\mathbf{F1} = 2 \cdot \frac{\mathbf{Precision} \cdot \mathbf{Recall}}{\mathbf{Precision} + \mathbf{Recall}} \tag{25}$$

Other metrics such as the Matthews Correlation Coefficient (MCC) can be employed when a more thorough evaluation is needed in situations where there are unbalanced FNs vs FPs [60]. Accuracy, aka *Error Rate* (equation 26a) is the last commonly used metric for evaluating prediction capabilities [75].

$$\mathbf{Accuracy} = \frac{\mathbf{TP} + \mathbf{TN}}{\mathbf{TP} + \mathbf{TN} + \mathbf{FP} + \mathbf{FN}} \tag{26a}$$

Unlike the previous 3 metrics, accuracy takes into account True Negative (TN) predictions. Accuracy evaluates the models performance across all classes and is particularly useful when classes are of equal importance [71].
Having a balanced (i.e. unbiased) dataset is also important for evaluating the effectiveness of our model. Christopher K. Williams [60] notes that precision of a classifier depends on the ratio of positive and negative cases present in the test dataset [74]. As an example, imagine we have a biased dataset containing 99 pictures of dogs and 1 picture of cats for a dog-cat binary classification model. If our model classifies all 100 images as dogs, it

would be said to have an accuracy of 0.99 (99%), Precision of 99 / (99 + 1) = 0.99 and Recall of 99 / (99 + 0) = 1.0, even when it failed to correctly label the cat class 100% of the time. A more balanced dataset which contains a realistic distribution of inputs would result in a more trustworthy value of precision and recall. Having datasets full of cherry picked data would lead to misleading metrics on real predictions. This is one reason why datasets are often shuffled, then split into training, testing and validation sets. Ultimately, knowing if detecting false positives or false negatives is more critical for the effectiveness of the model is situation dependent.

4.2.1 TuSimple Evaluation

Since we rely on TuSimple as the main dataset for evaluation compared with other models, we choose to use their evaluation formula specified in the lane detection challenge. For the TuSimple lane detection challenge, they account Accuracy (equation 26b) as their main metric.

$$\mathbf{Accuracy} = \frac{\sum_{\mathbf{clip}} \mathbf{C_{clip}}}{\sum_{\mathbf{clip}} \mathbf{S_{clip}}} \tag{26b}$$

Where, C_{clip} is the number of correct points in the last frame of the clip and S_{clip} is the number of requested points in the last frame of the clip.

Although, they remain vague when it comes to defining what qualifies as a correct point: “If the difference between the width of ground-truth and prediction is less than a threshold, the predicted point is a correct one” [67]. TuSimple evaluation criteria includes rate of false positives (27a), false negatives (27b):

$$\mathbf{FP} = \frac{\mathbf{F_{pred}}}{\mathbf{N_{pred}}}, \quad \mathbf{FN} = \frac{\mathbf{M_{pred}}}{\mathbf{N_{gt}}} \tag{27a), (27b}$$

Where, F_{pred} is the number of wrong predicted lanes, N_{pred} is the number of all predicted lanes, M_{pred} is the number of missed ground-truth lanes in the predictions and N_{gt} is the number of all ground-truth lanes.

This evaluation criteria combines both True Positive and True Negative into "correct predictions". Because of this, Precision, Recall and F1-score are not considered.

When examining the criteria provided by the TuSimple challenge, a few questions come to mind: Since a lane consists of 48 points, what constitutes a correctly or incorrectly predicted lane? Is it some percentage of the total lane considered matching? How far away is the prediction compared to the ground truth label? Lanes in the image tend to become smaller the further away they are, meaning a prediction 20 pixels away from the ground truth label at the bottom of the image is much closer than 20 pixels away at the horizon. We will need to define these ourselves and note that, although evaluation criteria is provided, it is not necessarily rigorous enough to distinguish a good model.

To begin addressing the questions above, we define a confusion matrix as follows: True Positive (TP) defined as when the prediction falls within the thresholded distance away from the ground truth point. True Negative (TN) when both the model and ground truth label are marked as -2, meaning the lane is not present in the image. False Positive (FP) defined as when the model predicts a pixel value, but the label is -2. False Negative (FN) when the model predicts a -2 value, but the label is some real pixel value. In our case, having a bias towards the model's performance of false positive detections is more important because our lane detection model is intended to guide lane correction. Predicting a lane is not present when it is, is less dangerous than predicting a lane is present when it is not because the vehicle would try to steer back into a lane it knows exists or stop the vehicle. If the model predicted a lane is present when reality there is nothing there, this could be dangerous because the vehicle could drive off road.

For inspiration in defining the threshold, we take a look at UFLD's method. They provide a clever way of creating a threshold that depends on the angle of the lane and a pixel distance. Once the model has predicted the x coordinates for each lane, a simple linear regression model uses least-squares to fit a straight line to all points on the lane. Next, they take the main coefficient (i.e. the slope of the line) and find the angle by taking the inverse tangent of the slope. Afterwards they divide the pixel threshold (in this case 20 pixels) by the cosine of the angle. Their reasoning for choosing 20 pixels is not specified, but can be viewed as:

$$Threshold = \frac{20}{\cos(arctan(m))} \tag{28}$$

Where, m = the coefficient of x in a slope-intercept form line $y = mx + b$.

This results in the higher the angle of the lane, the less distance is allowed to fit within the threshold. This is because shallow angles correspond to lanes further from the camera and therefore containing less pixels overall in the image, meaning the further lanes have less margin for error. Note, this is still not perfect, because points closer to the horizon are still treated the same as points closer to the bottom of the image.

UFLD evaluates their model by using the maximum accuracy over all predicted lanes. This method should be scrutinized because it does not inform the audience of the accuracy of each lane. A better approach would be to provide the list of accuracies for each lane and use a normal distribution to weight the accuracies, with lanes closer to the center of the image receiving higher weighting. This would better inform of cases where some lanes are detected very clearly, but other lanes are not detected. UFLD's current method would not be capable of distinguishing the effectiveness of a model which only predicts 2 lanes from one that predicts 5 lanes. Ignoring the model's ability to predict multiple lanes makes for an ineffective evaluation criterion. Ignoring the model's performance on all other lanes fails to communicate how the model will perform when switching lanes.

The basis of calculating the evaluation metrics (precision, recall, etc) relies on what predictions we consider to be True Positive. In the case of TuSimple, the confusion matrix is defined by the accuracy of the model. As previously noted, using only one threshold value would result in higher accuracy metrics which do not actually reflect an accurate model. One approach may be to tie the threshold size to the inverse distance from the camera combined with UFLD's method of penalizing lanes further in the x direction. Effectively this is accounting for the perspective effect with the threshold. Another approach could be to average the results of many different pixel distances. Since we lack a labeled depth estimation, we rely on both TuSimple's original metric and UFLD's modified threshold.

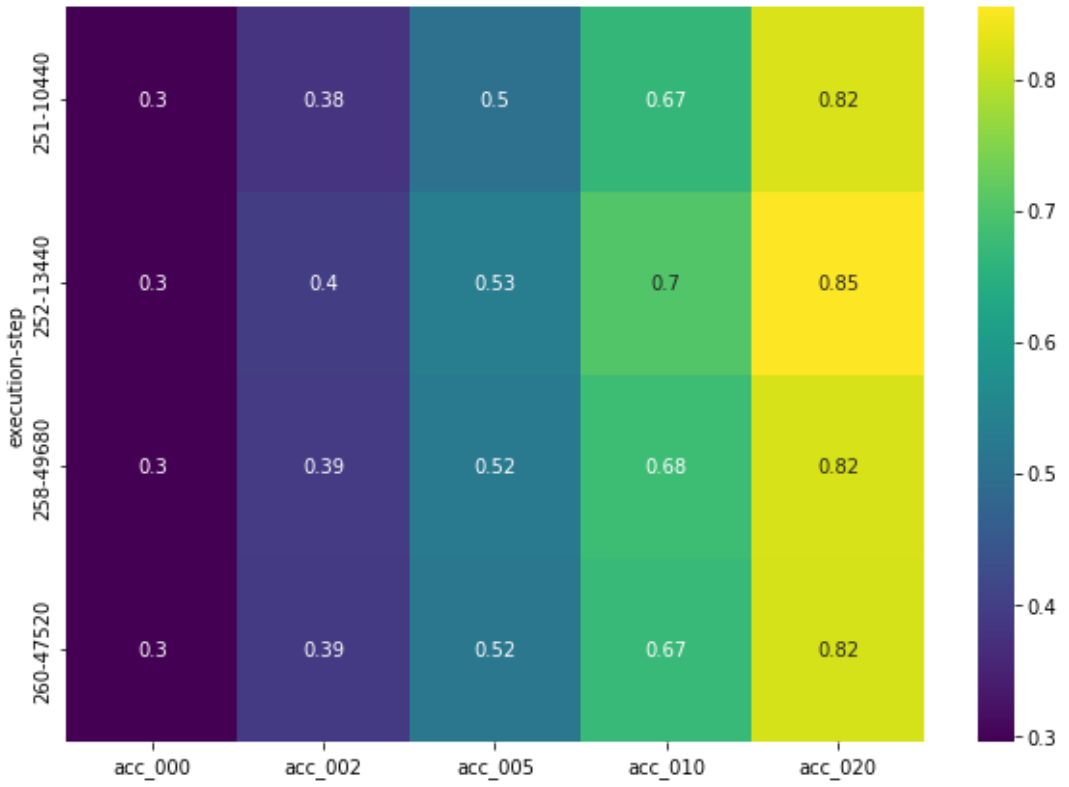


Figure 4.2: Accuracy compared to threshold size with number of iterations. The *x* direction shows different accuracy thresholds in pixels, while the colored bar on the right shows the accuracy of the model with a given threshold.

In figure 4.2 the accuracy increases as more pixel distance is allowed in the threshold. Accuracy at 0 pixel distance only works for evaluating the prediction of the *NaN* or *-2* values (i.e. true or false negative detections), since there is no distance in between missed.

4.3 Implementation Details

4.3.1 Input Image Size

Our cameras provide an image size of 1280x720 while our model expects an image resolution of 640x360. After features are detected with the ResNet backbone, the input image is scaled down to 256x256 on the GPU to speed up computation.

4.3.2 Hardware

Training the model was performed on an NVIDIA GeForce RTX 3080 Laptop GPU. CPU hardware used includes an 11th Gen Intel(R) Core(™) i7-11850H @ 2.50GHz with 64 GBs of RAM. The model was then deployed on an Nvidia Jetson AGX Xavier which has a 512-core Volta GPU with 64 Tensor cores and an 8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3 and 32B 256-Bit LPDDR4x memory [60].

4.3.3 Hyperparameters

The model was trained on a single GPU at image resolution of 640x360 and then 256x256 after feature detection. Batch size was 192, performed using 6 workers. Learning rate, selection loss weight and horizon loss weight were all set to 0.001. Lane order loss weight was set to 1. Maximum steps allowed was 20,000 with our best results achieved just after step 10000 or about 119 Epochs. A custom script saved the maximum accuracy value and step count and would terminate if no improvements were made while training the model on the cloud to save money. Figure 4.2 demonstrates that accuracy lowers slightly and plateaus from 15000 to 50000 steps.

4.3.4 Backbones

Our model was built upon PyTorch's implementation of the ResNet34 architecture, a commonly used deep residual neural network (RNN) which combines a series of convolutional layers for feature extraction in images [40]. ResNet34 [41] takes some input image and outputs a 1000 length feature vector describing the image. See the architecture diagram in section 3.2 for how we utilize this feature vector further in our network. We settled on ResNet34 as a backbone after trying other length ResNets (i.e. ResNet18, ResNet50, etc) as we found ResNet34 to result in a balance between inference speed and model accuracy when deployed on our embedded hardware.

4.3.5 Data Augmentation

Data augmentation is a method of enhancing the size of a dataset by applying techniques such as cropping an image or changing illumination conditions (brightness, contrast, noise, etc) to make a model both more robust to variations and better at generalizing the task [76]. Networks are only as good as the data fed in, meaning networks which are fed more data

generally yield better results. The amount of data needed tends to relate to the complexity of the problem and therefore the complexity of the model. There is not a way to know how much data is needed exactly for the task, but heuristics and approximations can help. A rule of thumb is to use 10 times more data samples than parameters [77]. Our team attempted many different types of data augmentation including spatial augmentation and augmenting color, but unfortunately found these augmentations to result in longer training times without much better results.

4.4 Results

4.4.1 Comparison with the State of the Art

In this section we cover the results using the TuSimple dataset comparing our model against 4 others including LaneNet, Robust Lane Detection, Ultra-Fast Lane Detection tested on an NVIDIA GeForce RTX 3080 Laptop GPU. All models were trained from scratch using the TuSimple training set. Training parameters were taken from the UFLD research paper which set batch size to 32, learning rate to 4e-4 with a cosine decay learning strategy with Adam optimizer and total number of epochs to 100. FPS was calculated by taking the average over 100 frames 10 times, then averaging the results. Frames were taken sequentially from the TuSimple testset. GPU startup time often leads to the first set of 100 frames having a significantly lower average FPS, which slightly pulls down the mean. Median FPS on inference is typically slightly higher. Randomly pulling frames from the dataset leads to slightly lower inference times and was avoided because it does not replicate real world scenarios, therefore all frames were loaded sequentially.

Table 1: Comparison of our method to several others on TuSimple Dataset.

Method	Accuracy	Runtime (fps)	Runtime (s)
LaneNet	96.40%	15	0.0666
RLD: unet lstm	97.91%	56	0.0178
RLD: unet	96.53%	57	0.0175
UFLD	95.82%	286	0.0035
Ours (ResNet 18)	93.10%	510	0.0019
Ours (ResNet 34)	96.94%	343	0.0029

We provide inference speed of our model deployed on our target embedded hardware: Nvidia Jetson AGX Xavier. As shown, even with pre and post processing, our model still runs above real time on our target hardware. See table 2 below:

Table 2: Vahidnet performance on Nvidia Jetson AGX Xavier.

Operation	Median FPS
Lane detection inference	48.37
LD inference + pre and post-processing	36.42

4.4.2 Lane Detection Prediction Results

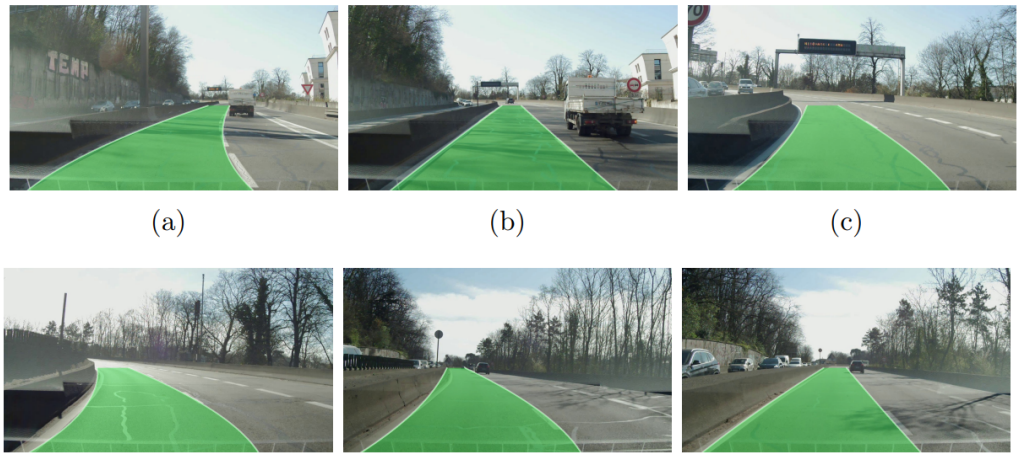


Figure 4.3 (a-f): We demonstrate the results of our model with post-processing applied.

4.4.3 Performance using Multiple Networks

Table 3: Comparison of model performance on Nvidia Jetson AGX Xavier

Operation	Media FPS	Processor Unit
LD inference	47.23	GPU
LD inference + pre and post-processing	35.13	GPU + CPU

Other NN	33.67	CPU

Even when both networks are run simultaneously, we are able to achieve above real time inference. Additionally we test inference when running alongside another neural network. The other neural network is proprietary, but produces similar FPS when tested alone.

5. Conclusion

5.1 Conclusion

This research was performed to examine the effectiveness of lane detection methods when placed on the Nvidia Jetson AGX Xavier SOC. Several lane detection methods focus solely on detection accuracy at the expense of speed, leading to models which are too slow to be useful in the real world. Significant computing power and therefore more expensive and larger equipment is required which requires major modification to the vehicle, prohibiting affordable commercial applications of autonomous driving.

We present a real time method which provides competitive accuracy and runs on an SOC capable of fitting in a vehicle's dashboard. We iterate on the row-wise classification method pioneered by UFLD [49] which enables stable and accurate lane detection at high speeds using individual images. We further modify this method to run at real time on the Nvidia Jetson AGX Xavier SOC by reducing the model's complexity and output parameters. Additionally, the model meets level 2 / 2++ ADAS operational design domain requirements.

Accurately and truthfully evaluating a lane detection model is not a trivial task. Without external information about the vehicle, evaluating the model often relies on methods which do not model the real world, such as choosing a pixel threshold for predictions to fall within. Other lane detection methods often rely on subjective heuristics for evaluating performance which do not accurately reflect how the model might perform in reality. By breaking down our model into 3 separate tasks of lane detection and order, lane selection and horizon detection, we are able to eliminate false predictions and consistently define the ego lane in which the vehicle is driving. This method also allows the ability for the model to stably switch lanes with consistent predictions. One limitation is that the model is intended to be used for the purpose of informing the driver when the vehicle is departing the lane while driving on the highway. The single camera method does not have a full understanding of the surrounding vehicle and will only be used as a level 2 ADAS safety feature. Fully autonomous control will require a more robust perception stack with more cameras to increase the vehicle's awareness.

Although the model has been evaluated visually and methodologically, we have yet to see how it performs when used to control a vehicle. Understanding if the model will perform safely when combined with a control system is difficult to determine without additional validation. Realistically evaluating the effectiveness of the model will require real world

tests which are beyond the scope of this research. Nonetheless, our model combined with the Kalman filter has demonstrated reliable predictions which are stable and accurate.

The method mainly relies on the ResNet34 backbone for feature detection and may be improved using more powerful architectures. Furthermore, the accuracy of the model could benefit from training on more data in more situations than highway driving or with a more rigorous use of data augmentation methods. Other similar methods such as SwiftLane [55] use false positive suppression techniques to increase accuracy and speed. Finally, we seek to continue downscaling the model such that it can run at real time on less powerful SOCs. We will continue to update our model with recent advances to ensure a safe and reliable lane detection model for semi-autonomous driving.

5.2 Discussion

In tackling this project the two biggest problems encountered had little to do with the actual science involved, but rather the accessibility to the ML models provided by other researchers.

With our current code hosting platforms such as GitHub, it has been easier than ever to access the research results of others. Being able to run and verify the results of a model developed by a team across the world is a modern wonder which has been a great triumph for research. Although, a few barriers stand in the way of this free and open accessibility to research. In trying to evaluate Robust Lane Detection, only some of the pre-trained models provided by the team were accessible through Google drive. The efforts of the Robust Lane Detection team were excellent in that they provided free access to an extended version of the TuSimple dataset. Access to Google is (at the time of writing) restricted in China and thus the team collaborated internationally in order to provide these models. I think it is very important for us to be aware of the platforms in which we publish our research, noting that it may very well be inaccessible to our fellow scientists elsewhere in the world.

The second issue involves dependencies in deep learning. With the current infrastructure, we have a pipeline of PyTorch, OpenCV, NumPy, Docker and Nvidia's CUDA which enables a development to edge pipeline which enables quick iteration when creating real world applications with deep learning. It comes at the cost of managing many dependencies which often do not work well together. There were many moments in this research where some combination of dependencies had become obsolete and no longer available on the platforms which host previous versions of the

dependencies listed above. Effectively this pushes research which relies on these old dependencies out of reach, making them quickly obsolete.

In the case of testing LaneNet and even my team's current model, the difference between software versions made these models un-deployable without first virtualizing an environment. This dependency problem greatly increases the time and complexity of running a model that is from just a few years before. Attempting to test SCNN, using up to date hardware to test the other models no longer supported the outdated versions of dependencies required to run their model, which resulted in the model being dropped from the experiments. Thinking about this pipeline brings the topic of how to maintain deep learning models and prevent them from becoming obsolete within a matter of years.

Bibliography

[1] Road traffic injuries [Internet]. World Health Organization. World Health Organization; [cited 2022Jun28]. Available from: https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries#:~:text=Approximately%201.3%20million%20people%20die,pedestrians%2C%20cyclists%2C%20and%20motorcyclists.

[2] 2016 fatal motor vehicle crashes: Overview - transportation [Internet]. U.S Department of Transportation National Highway Traffic Safety Administration; 2017 [cited 2022Jul3]. Available from: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812456

[3] Reimer J. Adas Tech can save lives by helping you and your car drive safer at night [Internet]. here.com. 2022 [cited 2022Jul3]. Available from: https://www.here.com/company/blog/adas-technology-night-time#:~:text=ADAS%3A%20an%20intelligent%20nightlight%20for,injuries%20and%2029%25%20of%20deaths.

[4] Benson, A., Tefft, B.C., Svancara, A.M. & Horrey, W.J. (2018). Potential Reduction in Crashes, Injuries and Deaths from Large-Scale Deployment of Advanced Driver Assistance Systems (Research Brief). Washington, D.C.: AAA Foundation for Traffic Safety. [cited 2022Jul5]. Available from: https://aaafoundation.org/potential-reduction-in-crashes-injuries-and-deaths-from-large-scale-deployment-of-advanced-driver-assistance-systems/

[5] Brussels, European Commission / European Road Safety Observatory (ERSO), 2016, 38 p., ref. [cited 2022Jul5]. Available from: https://swov.nl/en/publicatie/advanced-driver-assistance-systems-0

[6] Nissan D. A brief history of lane departure warnings [Internet]. Medium. Medium; 2016 [cited 2022Feb17]. Available from: https://medium.com/@ducannissan/a-brief-history-of-lane-departure-warnings-f6316fce8427

[7] O.-R. A. D. (ORAD) Committee, 'Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles', SAE International 2021.[cited 2022Jul7]. Available from: https://saemobilus.sae.org/content/j3016_202104

[8] Francis T. What is Lane-Keeping Assist? [Internet]. Auto Express. AutoExpress; 2021 [cited 2022Jul6]. Available from:

https://www.autoexpress.co.uk/tips-advice/356028/what-lane-keeping-assist

[9] F. Chollet, Deep Learning with Python, 2017, isbn: 9781617294433. (page 5)

[10] Gonzalez RC, Woods RE. 3.4 Fundamentals of Spatial Filtering. In: Digital Image Processing. 4th ed. New York, NY: Pearson; 2018. p. 153–9. (Global Edition).

[11] Fukushima K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics. 1980Apr1;36(4):193–202.

[12] P. V. C. Hough, "Method and Means for Recognizing Complex Patterns", US Patent 3,069,654, Ser. No. 17,7156 Claims, 1962.

[13] Canny J. A computational approach to edge detection. Readings in Computer Vision. 1987;:184–203.

[14] Shiwakoti N. Lane detection and tracking [Internet]. Encyclopedia. Scholarly Community Encyclopedia; [cited 2022Feb24]. Available from: https://encyclopedia.pub/entry/15330

[15] Duda, R. O. and P. E. Hart, "Use of the Hough Transformation to Detect Lines and Curves in Pictures," Comm. ACM, Vol. 15, pp. 11–15 (January, 1972).

[16] B. Yu and A. K. Jain, "Lane boundary detection using a multiresolution Hough transform," Proceedings of International Conference on Image Processing, 1997, pp. 748-751 vol.2, doi: 10.1109/ICIP.1997.638604.

[17] Ping-shu Ge, Lie Guo, Guo-kai Xu, Rong-hui Zhang, Tao Zhang, "A Real-Time Lane Detection Algorithm Based on Intelligent CCD Parameters Regulation", Discrete Dynamics in Nature and Society, vol. 2012, Article ID 273164, 16 pages, 2012. https://doi.org/10.1155/2012/273164

[18] Peng Sun and Hui Chen "Lane detection and tracking based on improved Hough transform and least-squares method", Proc. SPIE 9301, International Symposium on Optoelectronic Technology and Application 2014: Image Processing and Pattern Recognition, 93011U (24 November 2014); https://doi.org/10.1117/12.2072393

[19] Seitz S, Sinha P. Image Filtering [Internet]. CSE455: Computer Vision, University of Washington; Lecture presented Winter (2003); [cited 2022May11]. Available from: https://courses.cs.washington.edu/courses/cse455/03wi/lectures/filter.pdf

[20] J. P. Gonzalez and U. Ozguner, "Lane detection using histogram-based segmentation and decision trees," ITSC2000. 2000

IEEE Intelligent Transportation Systems. Proceedings (Cat. No.00TH8493), 2000, pp. 346-351, doi: 10.1109/ITSC.2000.881084.

[21] Kuo-Yu Chiu and Sheng-Fuu Lin, "Lane detection using color-based segmentation," IEEE Proceedings. Intelligent Vehicles Symposium, 2005., 2005, pp. 706-711, doi: 10.1109/IVS.2005.1505186.

[22] J. Lee and J. Cho, "Effective Lane Detection and Tracking Method Using Statistical Modeling of Color and Lane Edge-Orientation," 2009 Fourth International Conference on Computer Sciences and Convergence Information Technology, 2009, pp. 1586-1591, doi: 10.1109/ICCIT.2009.81.

[23] Kippenbrock R. Ross Kippenbrock - Finding Lane lines for self driving cars [Internet]. YouTube. PyDataTV; 2017 [cited 2022Mar1]. Available from: https://www.youtube.com/watch?v=VyLihutdsPk

[24] Keenan R, Kul S, Bucholtz B, Chadha H. Udacity/carnd-advanced-lane-lines [Internet]. GitHub. Udacity; [cited 2022Mar1]. Available from: https://github.com/udacity/CarND-Advanced-Lane-Lines

[25] Y. Lee and H. Kim, "Real-time lane detection and departure warning system on embedded platform," 2016 IEEE 6th International Conference on Consumer Electronics - Berlin (ICCE-Berlin), 2016, pp. 1-4, doi: 10.1109/ICCE-Berlin.2016.7684702.

[26] Zhou D-X. Universality of deep convolutional Neural Networks. Applied and Computational Harmonic Analysis. 2020;48(2):787–94.

[27] Rosenblatt, F. The perceptron - A perceiving and recognizing automaton. Cornell Aeronautical Laboratory; 1957. Report No.:85-460-1

[28] Brownlee J. Perceptron algorithm for classification in Python [Internet]. Machine Learning Mastery. 2020 [cited 2022Mar3]. Available from: https://machinelearningmastery.com/perceptron-algorithm-for-classification-in-python/#:~:text=The%20Perceptron%20algorithm%20is%20a,and%20predicts%20a%20class%20label.

[29] Minsky, M, Papert, S. Perceptrons: An Introduction to Computational Geometry. MIT Press; 1969.

[30] Robbins H, Monro S. A stochastic approximation method. The Annals of Mathematical Statistics. 1951;22(3):400–7.

[31] Baheti P. Activation functions in neural networks [12 types & use cases] [Internet]. V7. Microsoft; [cited 2022Mar9]. Available from: https://www.v7labs.com/blog/neural-networks-activation-functions

[32] Fukushima K. Cognitron: A self-organizing multilayered neural network. Biological Cybernetics. 1975;20(3-4):121–36.

[33] Hassan, Hassan & Negm, Abdelazim & Zahran, Mohamed & Saavedra, Oliver. (2015). ASSESSMENT OF ARTIFICIAL NEURAL NETWORK FOR BATHYMETRY ESTIMATION USING HIGH RESOLUTION SATELLITE IMAGERY IN SHALLOW LAKES: CASE STUDY EL BURULLUS LAKE.. International Water Technology Journal. 5. Figure 4.

[34] Zheng A, Casari A. Chapter 2: Fancy Tricks with Simple Numbers / Feature Selection. In: Feature Engineering for Machine Learning: Principles and techniques for Data scientists. Beijing: O'Reilly; 2018. p. 38–9.

[35] Pani A. Convolutional Neural Networks [Internet]. LinkedIn. Microsoft; 2018 [cited 2022Sep10]. Available from: https://www.linkedin.com/pulse/convolutional-neural-networks-anindita-pani/

[36] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional Neural Networks. Communications of the ACM. 2017;60(6):84–90.

[37] Amini A. MIT 6.S191 (2020): Convolutional Neural Networks [Internet]. YouTube. MIT; 2020 [cited 2022Mar11]. Available from: https://www.youtube.com/watch?v=iaSUYvmCekI

[38] Li F-F, Wu J, Gao R. [Internet]. CS231N convolutional neural networks for visual recognition. Stanford University; [cited 2022Mar11]. Available from: https://cs231n.github.io/convolutional-networks/

[39] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[40] Resnet34 [Internet]. resnet34 - Torchvision main documentation. [cited 2022Jul2]. Available from: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet34.html

[41] He, K, Zhang, X, Ren, S, Sun, JDeep Residual Learning for Image Recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2016 (pp. 770-778).

[42] Polyak BT. Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics. 1964;4(5):1–17.

[43] Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986Oct9;323(6088):533–6.
[44] Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. 2014 Dec 22.
[45] What is layer normalization? | deep learning fundamentals [Internet]. YouTube. AssemblyAI; 2022 [cited 2022Jun14]. Available from: https://www.youtube.com/watch?v=2V3Uduw1zwQ
[46] Ba JL, Kiros JR, Hinton GE. Layer normalization [Internet]. arXiv.org. 2016 [cited 2022Jun16]. Available from: https://arxiv.org/abs/1607.06450
[47] Wang, Z, Ren, W, Qiu, Q. "LaneNet: Real-Time Lane Detection Networks for Autonomous Driving" (2018). https://arxiv.org/abs/1807.01726 (accessed Jan. 18, 2022)
[48] Q. Zou, H. Jiang, Q. Dai, Y. Yue, L. Chen, Q. Wang. "Robust lane detection from continuous driving scenes using deep neural networks". IEEE Transactions on Vehicular Technology 2020; 69(1):41–54
[49] Qin, Z, Wang, H, Li, X. Ultra Fast Structure-aware Deep Lane Detection (2020). https://arxiv.org/abs/2004.11757
[50] Xingang Pan, Jianping Shi, Ping Luo, Xiaogang Wang, Xiaoou Tang. "Spatial As Deep: Spatial CNN for Traffic Scene Understanding", AAAI2018
[51] Q. Ju, et al., "PathMark: A Novel Fast Lane Detection Algorithm for Embedded Systems," in 2012 Fourth International Symposium on Information Science and Engineering (ISISE 2012), Shanghai, 2012 pp. 68-73. doi: 10.1109/ISISE.2012.24
[52] Jeng, Ming-Jer & Hsueh, Pi-Chih & Yeh, Chun-Wei & Hsiao, Pei-Yung & Cheng, Chao-Han & Chang, Liann-Be. (2007). Real time mobile lane detection system on PXA255 embedded system.
[53] N. Mechat, N. Saadia, N. K. M'Sirdi and N. Djelal, "Lane detection and tracking by monocular vision system in road vehicle," 2012 5th International Congress on Image and Signal Processing, 2012, pp. 1276-1282, doi: 10.1109/CISP.2012.6469683.
[54] M. Nieto, L. Garcia, O. Scnderos and O. Otaegui, "Fast Multi-Lane Detection and Modeling for Embedded Platforms," *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, pp. 1032-1036, doi: 10.23919/EUSIPCO.2018.8553262.
[55] Oshada Jayasinghe, Damith Anhettigama, Sahan Hemachandra, Shenali Kariyawasam, Ranga Rodrigo, Peshala JayasekaraSwiftLane: Towards Fast and Efficient Lane Detection. In 2021 20th IEEE

International Conference on Machine Learning and Applications (ICMLA) 2021 . IEEE.

[56] Neven D. De Brabandere, B. Georgoulis, S, Proesmans, M, Van Gool, L. Towards End-to-End Lane Detection: an Instance Segmentation Approach.

[57] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in International Conference on Medical Image Computing and Computer Assisted Intervention, 2015.

[58] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W. chun Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," in Advances in neural information processing systems (NIPS), 2015, pp. 802–810.

[59] Gospodinova S, Miccoli F. Press corner [Internet]. European Commission - European Commission. European Commission; [cited 2022Jun15]. Available from: https://ec.europa.eu/commission/presscorner/detail/en/IP_22_4312

[60] Jetson AGX Xavier Developer kit [Internet]. NVIDIA Developer. Nvidia; 2022 [cited 2022Jun15]. Available from: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[61] Jetson Nano Developer Kit [Internet]. NVIDIA Developer. NVIDIA; 2022 [cited 2022Jun12]. Available from: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[62] RESNET34 Fast.ai V2 classification model [Internet]. Roboflow. [cited 2022Jun6]. Available from: https://models.roboflow.com/classification/resnet34

[63] Binary crossentropy loss function: Peltarion platform [Internet]. Peltarion. Peltarion; [cited 2022May11]. Available from: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy

[64] Gómez R. Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names [Internet]. Understanding categorical cross-entropy loss, binary cross-entropy loss, Softmax loss, logistic loss, focal loss and all those confusing names. 2018 [cited 2022Jul29]. Available from: https://gombru.github.io/2018/05/23/cross_entropy_loss/

[65] DigitalSreeni. Tips tricks 15 - understanding binary cross-entropy loss [Internet]. YouTube. YouTube; 2021 [cited 2022May13].

Available from: https://www.youtube.com/watch?v=Md4b67HvmRo

[66] Hansen C. Optimizers explained - adam, momentum and stochastic gradient descent [Internet]. Machine Learning From Scratch. Machine Learning From Scratch; 2019 [cited 2022Jul2]. Available from: https://mlfromscratch.com/optimizers-explained/#/

[67] TuSimple. Tusimple/Tusimple-benchmark: Download datasets and ground truths: Https://github.com/tusimple/tusimple-benchmark/issues/3 [Internet]. GitHub. TuSimple; [cited 2022Jan24]. Available from: https://github.com/TuSimple/tusimple-benchmark

[68] CuLane Dataset [Internet]. Culane dataset. Multimedia Laboratory, The Chinese University of Hong Kong; [cited 2022Jan28]. Available from: https://xingangpan.github.io/projects/CULane.html

[69] Becker A. Online kalman filter tutorial [Internet]. Kalman Filter Tutorial. [cited 2022Jul13]. Available from: https://www.kalmanfilter.net/default.aspx

[70] Precision and recall in Machine Learning - Javatpoint [Internet]. www.javatpoint.com. [cited 2022Jul3]. Available from: https://www.javatpoint.com/precision-and-recall-in-machine-learning#:~:text=Recall%20of%20a%20machine%20learning%20model%20is%20dependent%20on%20positive,correctly%20classifying%20all%20positive%20samples.

[71] Classification: Precision and recall | machine learning | google developers [Internet]. Google. Google; [cited 2022Jul9]. Available from: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

[72] Gad AF. Accuracy, precision, and recall in deep learning [Internet]. Paperspace Blog. Paperspace Blog; 2021 [cited 2022Jul12]. Available from: https://blog.paperspace.com/deep-learning-metrics-precision-recall-accuracy/#:~:text=The%20recall%20measures%20the%20model%27s,classified%2C%20e.g.%20for%20the%20precision.

[73] LT Z. Essential things you need to know about F1-score [Internet]. Medium. Towards Data Science; 2022 [cited 2022Jul12]. Available from: https://towardsdatascience.com/essential-things-you-need-to-know-about-f1-score-dbd973bf1a3#:~:text=1.-,Introduction,competing%20metrics%20%E2%80%94%20precision%20and%20recall.

[74] Christopher K. I. Williams. "The Effect of Class Imbalance on Precision-Recall Curves". Neural Computation 2021; 33(4):853–857.
[75] Classification: Accuracy | machine learning | google developers [Internet]. Google. Google; [cited 2022Jul29]. Available from: https://developers.google.com/machine-learning/crash-course/classification/accuracy
[76] Takimoglu A. Top data augmentation techniques: Ultimate guide for 2022 [Internet]. AIMultiple. 2022 [cited 2022Jul14]. Available from: https://research.aimultiple.com/data-augmentation-techniques/
[77] Brownlee J. How much training data is required for machine learning? [Internet]. Machine Learning Mastery. 2019 [cited 2022Jul12]. Available from: https://machinelearningmastery.com/much-training-data-required-machine-learning/