

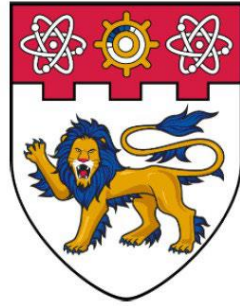
Deep learning frameworks for lane detection in rainy images

Tan, Herman Kai Liang

2021

Tan, H. K. L. (2021). Deep learning frameworks for lane detection in rainy images. Final Year Project (FYP), Nanyang Technological University, Singapore.
<https://hdl.handle.net/10356/149815>

<https://hdl.handle.net/10356/149815>



**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

Deep Learning Frameworks for Lane Detection in Rainy Images

Submitted by

Tan Kai Liang, Herman

U1720939J

(Project Number: A3227-201)

Supervisor: Assoc. Prof. Soong Boon Hee

Co-supervisor: Dr Wang Jiangang (I2R, A*STAR)

Examiner: Assoc. Prof. Siek Liter

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

A Final Year Project report presented to

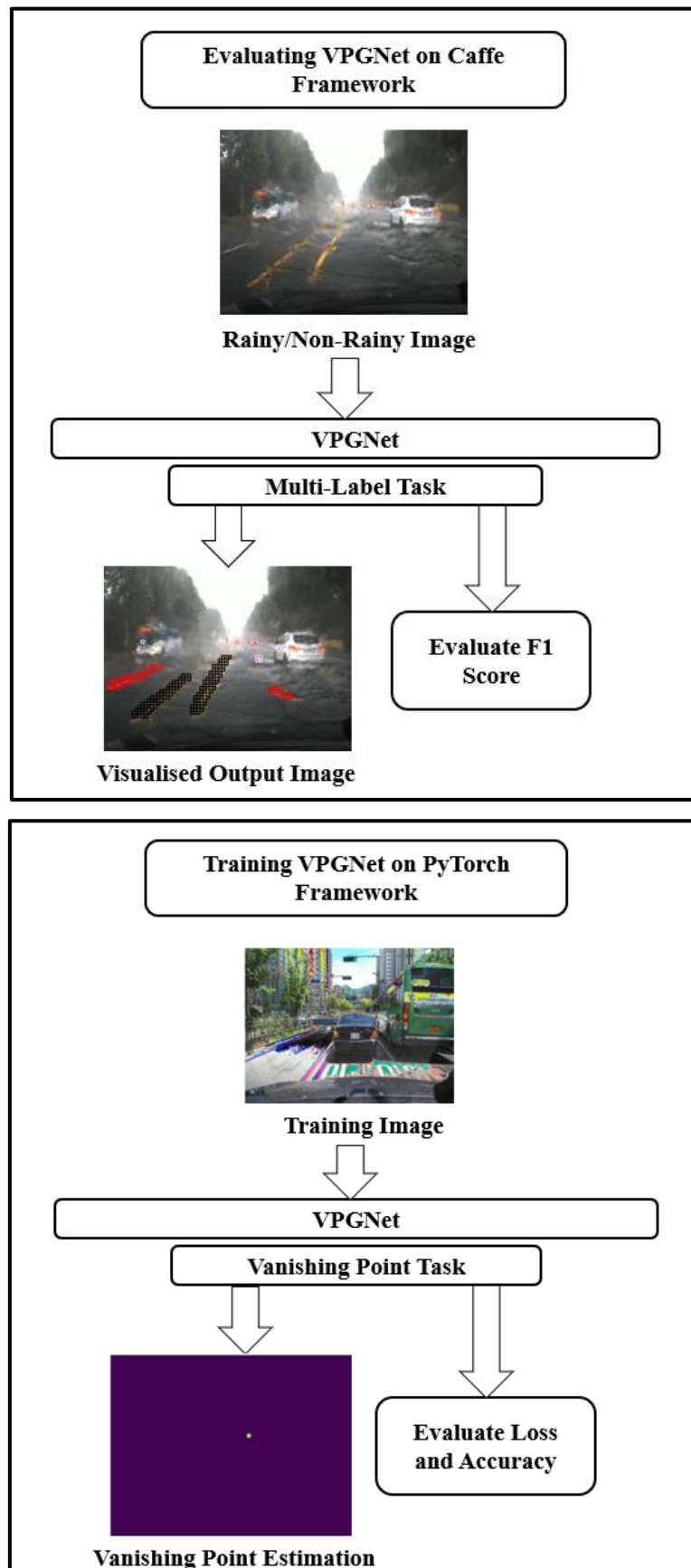
Nanyang Technological University

In partial fulfillment of the requirements for the

Degree of Bachelor of Engineering

April 2021

Deep Learning Frameworks for Lane Detection in Rainy Images



Abstract

With advancements in technology, artificial intelligence (AI) has seeped into our daily lives through innovations such as digital assistants like Amazon's Alexa, and autonomous vehicles which do not require a human driver. Although autonomous vehicles have seen improvements over the last few years, their lane detection accuracy can still be affected if they experience low visibility. In this project, the deep learning approach will be explored to improve lane detection accuracy in conditions with poor visibility, such as rain. Computer vision techniques, such as masking and regression, would be implemented and evaluated alongside a deep learning architecture to solve this problem. Vanishing point estimation would be used as an auxiliary task to further improve the lane detection process.

In this project, detectors operating on different deep learning frameworks would be tested on lane images in rainy and non-rainy conditions. Results from the different deep learning frameworks would be reviewed for their lane detection efficacy using evaluation metrics.

An existing lane detection algorithm, VPGNet, would be used as the foundation for this project. It would be trained and tested in its original deep learning framework, Caffe, for its efficacy in lane detection in rainy and non-rainy environments. VPGNet in Caffe was trained using a non-rainy dataset and a dataset that contained both non-rainy and rainy data. After the completion of training, lane detection was evaluated using the F1 score metrics.

After VPGNet was evaluated on the Caffe framework, an improved version of the model was proposed for the PyTorch framework. PyTorch was chosen for its intuitive design as well as ease of setting up. The PyTorch model would attempt to recreate the vanishing point task that was not included in VPGNet's Caffe code. Other loss functions would also be explored under the PyTorch framework to assess its effect on training and testing performance.

Results from both Caffe and PyTorch frameworks would be analysed. VPGNet in the Caffe framework showed remarkable results in non-rainy environments as well as in the different environments that arose from rainy weather, such as limited visibility and reflections on the

road surface. The vanishing point task in the PyTorch framework would still require improvements to improve its accuracy.

Acknowledgements

I would like to thank my project supervisor, Associate Professor Soong Boon Hee, for allowing me a chance to work with him and with A*STAR. I would also like to extend my deepest gratitude to Dr Teoh Eam Khwang for advising me during this duration of my Final Year Project. He had extended a great amount of encouragement and guidance to ensure we are on the right course in the project.

I also had the great pleasure of working with my co-supervisor Dr. Wang Jiangang, who had entrusted me with this project. I am extremely grateful to Mr. Prabhu for all the help he had provided me during the project. He never wavered in his support and provided invaluable assistance for the problems I had faced while working on the project.

Table of Contents

Abstract.....	i
Acknowledgements	iii
List of Figures.....	vi
List of Tables	viii
Chapter 1: Introduction	1
1.1 Background.....	1
1.2 Motivation.....	2
1.3 Objective	2
1.4 Scope.....	2
1.5 Report Organisation	3
Chapter 2: A Literature Review	4
2.1 Introduction.....	4
2.2 Traditional Methods for Object Detection	4
2.2.1 Scale Invariant Feature Transform (SIFT).....	5
2.2.2 Speeded Up Robust Features (SURF).....	7
2.3 Types of Neural Networks	8
2.3.1 Convolutional Neural Networks (CNN)	8
2.3.2 Region Based Convolutional Neural Network (R-CNN).....	9
2.3.3 You Only Look Once (YOLO)	10
2.3.4 Residual Neural Network (ResNet)	11
2.3.5 ENet	12
2.4 Existing Lane Detectors.....	14
2.4.1 Spline Fitting Approach.....	14
2.4.2 LaneNet.....	16
2.4.3 VPGNet.....	18
2.5 Concluding Remarks.....	19
Chapter 3: Training and Testing VPGNet in Caffe Framework.....	21
3.1 Introduction.....	21
3.2 Caffe Overview.....	21
3.3 VPGNet's Caffe Structure	22
3.4 Vanishing Points (VP)	24
3.5 Data Preparation.....	26

3.6 Training & Testing Environment.....	28
3.7 Training Results and Discussion.....	30
3.8 Deploying VPGNet for Lane Detection.....	32
3.9 Concluding Remarks.....	34
Chapter 4: Proposed Model in PyTorch Framework.....	35
4.1 Introduction.....	35
4.2 PyTorch Overview	35
4.3 Loss Functions	36
4.3.1 L1 Loss.....	36
4.3.2 L2 Loss.....	36
4.3.3 Binary Cross Entropy with Logits Loss	37
4.4 Stochastic Gradient Descent (SGD) with Momentum	37
4.5 Dilated Convolutions	38
4.6 Proposed Model	39
4.7 Concluding Remarks.....	40
Chapter 5: Results from Caffe and PyTorch Frameworks	41
5.1 Introduction.....	41
5.2 F1 Score Calculation.....	41
5.3 Results from Caffe Model.....	42
5.3.1 Cordova Dataset.....	43
5.3.2 VPGNet Non-Rainy Data.....	45
5.3.3 VPGNet Rainy Data.....	47
5.4 Results from Pytorch Model	52
5.5 Discussion of Results from Caffe and PyTorch Frameworks	55
5.6 Concluding Remarks.....	56
Chapter 6: Conclusion and Recommendations for Future Work	57
6.1 Conclusion	57
6.2 Recommendations for Future Work.....	58
6.2.1 Issues from F1 score calculation	58
6.2.2 Improving Caffe & PyTorch models.....	59
Bibliography	60
Appendix.....	65
Appendix A: Training Process of VPGNet on Caffe Framework.....	65
Appendix B: Deployment Code for VPGNet on Caffe Framework	66
Appendix C: Training Code for VPGNet on PyTorch Framework	70

List of Figures

Figure 2- 1: Extrema detection by comparing one pixel with 26 other neighbours [1]	5
Figure 2- 2: Gradient magnitude and direction at each point (left) is used to find the keypoint descriptor (right) [1]	6
Figure 2- 3: Gaussian second order derivative in y-direction [2]	7
Figure 2- 4: Gaussian second order derivative in xy-direction [2]	7
Figure 2- 5: Illustration of a sample CNN [3]	8
Figure 2- 6: R-CNN overview [5]	9
Figure 2- 7: YOLO's object detection overview [6]	10
Figure 2- 8: Residual Building Block [7]	11
Figure 2- 9: (a) Initial Block of ENet, (b) Bottleneck module used in ENet [8]	12
Figure 2- 10: ENet's network architecture [8]	13
Figure 2- 11: Spline fitting and localization [9]	14
Figure 2- 12: Spline fitting and localization [9]	15
Figure 2- 13: Final output [9]	15
Figure 2- 14: Lane edge proposal architecture [10]	16
Figure 2- 15: Lane line localisation architecture [10]	17
Figure 2- 16: VPGNet operating under (a) normal conditions, (b) with road markings, (c) night scenes, (d) rainy conditions [11]	19
Figure 3- 1: A sample Caffe net with 3 layers that define the top and bottom blobs [12]	22
Figure 3- 2: VPGNet's architectural details [11]	22
Figure 3- 3: Input size for VPGNet	23
Figure 3- 4: First convolutional layer in VPGNet	23
Figure 3- 5: Illustration of vanishing point [17]	24
Figure 3- 6: Output from VP task [11]	25
Figure 3- 7: Sample images from (a) Cordova, (b) VPGNet non-rainy, (c) VPGNet rainy dataset	26
Figure 3- 8: Sample annotations from Cordova dataset	26
Figure 3- 9: Sample annotations from VPGNet dataset	27
Figure 3- 10: 'make_lmdb.sh' file	28
Figure 3- 11: Verification of CUDA installation	28
Figure 3- 12: 'solver.prototxt' file	29
Figure 3- 13: Training Loss for Cordova trained model (Logarithmic scale)	30
Figure 3- 14: Final Loss values for Cordova trained model at iteration #100,000	30
Figure 3- 15: Training Loss for VPGNet trained model (Logarithmic scale)	31
Figure 3- 16: Final Loss values for VPGNet trained model at iteration #100,000	31
Figure 3- 17: Function to process an image for VPGNet	32
Figure 3- 18: Function that outputs detect lane locations onto input image	33
Figure 4- 1: Sigmoid function [26]	37
Figure 4- 2: Illustration of learning rate affecting how the model learns [27]	37
Figure 4- 3: (a) 1-dilation, (b) 2-dilation, and (c) 4-dilation convolutional outputs [29]	38
Figure 4- 4: Proposed PyTorch model	39

Figure 5- 1: (a) Original image, (b) Output from Model of Cordova Dataset	43
Figure 5- 2: (a) Original image, (b) Output from Model of VPGNet Non-Rainy Dataset	45
Figure 5- 3: (a) Original image, (b) Output from Model of VPGNet Rainy Dataset	47
Figure 5- 4: (a) Original image, (b) Output from Model of VPGNet Rainy Dataset	49
Figure 5- 5: (a) Original image, (b) Output from Model of VPGNet Rainy Dataset	51
Figure 5- 6: Ground Truth VP for sample image.....	52
Figure 5- 7: Output from VP task using L1 Loss.....	53
Figure 5- 8: Output from VP task using L2 Loss.....	54
Figure 5- 9: Output from VP task using BCE with Logits Loss.....	54
Figure 5- 10: Output from VP task using Soft Margin Loss.....	55

List of Tables

Table 3- 1: Lane and road marking classes annotated in VPGNet dataset	27
Table 4- 1: Proposed PyTorch model's layer information.....	39
Table 5- 1: Confusion Matrix	41
Table 5- 2: F1 Scores for Figure 5-1 (b)	44
Table 5- 3: F1 Scores for Figure 5-2 (b)	45
Table 5- 4: F1 Scores for Figure 5-3 (b)	48
Table 5- 5: F1 Scores for Figure 5-4 (b)	50
Table 5- 6: F1 Scores for Figure 5-4 (b)	51

Chapter 1

Introduction

1.1 Background

Autonomous vehicles have seen a growth in popularity in recent years. The Advanced-Driver-Assistance System (ADAS) guides these vehicles to prevent road accidents that could be caused by human error. Lane and road marker tracking is an essential component of the ADAS technology, allowing the vehicle to function in a multitude of different environmental conditions. Under rainy conditions, visibility is usually poor due to the obstruction from rain elements. This could prevent the autonomous vehicle from detecting changes in its surroundings, which could result in an accident. Thus, there is substantial interest in an algorithm that can handle lane and road marker tracking in rainy conditions. Due to the success in numerous computer vision tasks, such as image restoration, detection and enhancement, a deep learning approach can contribute valuable insight to this challenge. Object Detection is a computer vision task that analyses and locates classes of objects in an image. This can be studied and used in the project for lane detection, where road lanes and markers can be located in an image.

1.2 Motivation

Autonomous vehicles are brimming with potential. It provides convenience as a driverless transport system, as well as reduced labour if used in the logistics industry. To be able to realise its full potential, it must first gain the ability to transverse different weather conditions without direct human supervision. A complication brought about by rainy conditions is the poor visibility it brings about, which hampers the vehicle's decision making. Rain droplets also add distortion to the vehicle's camera feed and warps the lane markings' appearance, which can cause the accuracy of its lane detection algorithm to decrease. After rainy conditions, road surfaces would contain image reflections which can cause the model to incorrectly classify those reflections as being legitimate. One approach to this problem would be to utilise vanishing point estimation as an auxiliary task to guide the lane detection process. The vanishing point can be used to provide a wholistic geometric context for the vehicle, and thus the high correlation between the location of the lanes and the vanishing point would be advantageous in the lane detection problem.

1.3 Objective

The objective of this project would be to analyse the efficiency and efficacy of lane detectors using different deep learning frameworks under rainy conditions. Computer vision techniques, such as regression and vanishing point estimation, will be evaluated and used to improve upon the deep learning approach to the problem. Results from each framework would be examined and other techniques would be used to further improve the model.

1.4 Scope

In this project, lane detection in a rainy environment will be the main focus. A lane detector model would be used as the foundation for this project and be implemented in Caffe and

subsequently in PyTorch. The results obtained from the individual frameworks would be used to examine the efficacy of each deep learning framework.

1.5 Report Organisation

Chapter 2: A Literature Review

This chapter reviews examines the different object detection techniques used. Different variants of convolutional neural networks and lane detection algorithms will also be reviewed. A suitable lane detection algorithm would be selected as the foundation for the project, and the knowledge gained from reviewing different object detection techniques may be used to improve upon the selected algorithm.

Chapter 3: Training and Testing VPGNet in Caffe

The process for training and deploying the selected model in its original framework, Caffe, would be described in detail.

Chapter 4: Proposed Model in PyTorch

The proposed lane detection model would be outlined in this chapter. Other components and parameters available in PyTorch that could replace the existing ones in VPGNet would also be examined.

Chapter 5: Results from Caffe and PyTorch

Results from the respective deep learning frameworks would be gathered and analysed in this chapter.

Chapter 6: Conclusion & Recommendations for Future Work

Conclusions drawn from the project would be discussed, as well as other improvements that could be applied to the Caffe and PyTorch models.

Chapter 2

A Literature Review

2.1 Introduction

To prepare for the project, existing lane detection methods were studied and analysed. Since lane detection is considered part of object detection, it will also be studied as part of the project. These existing methods could provide more insight into the problem, and their strengths could be used to construct an improved lane detection algorithm. In this chapter, the techniques for object detection would be discussed and existing lane detection algorithms would be inspected.

2.2 Traditional Methods for Object Detection

Objection detection is a common computer vision problem. Prior to convolutional neural networks being the popular solution for object detection, other techniques had been proposed

and commonly used to discern the location of objects in an image. In this section, the previously known methods for object detection would be discussed.

2.2.1 Scale Invariant Feature Transform (SIFT)

SIFT was first introduced in 2004 as a technique for distinct feature extraction and identifying objects in real-time [1]. The SIFT algorithm involved four steps: Scale-space Extrema Detection, Keypoint localisation, Orientation assignment, and Keypoint descriptor.

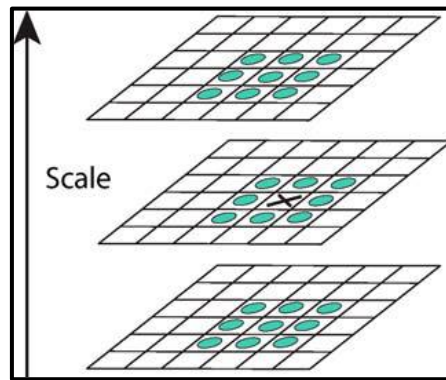


Figure 2- 1: Extrema detection by comparing one pixel with 26 other neighbours [1]

Scale-space extrema detection is used to identify the scale and locations in an image that could be assigned again when the object is viewed from different views. Thus, a continuous function of scale called scale space was used to find stable features over every possible scale. The scale space was obtained from the convolution of a variable-scale Gaussian with the input image. Stable locations in the scale space would be detected using the difference-of-Gaussian function, which involved calculating the difference between two nearby scales. To detect the extrema, one pixel in the image is compared with its 8 surrounding pixels as well as 9 neighbours in the scale below and above as shown in Figure 2-1. These extrema represent points of interest and are referred to as keypoints.

In Keypoint localisation, the keypoints obtained in the previous step were filtered to obtain more accurate points. Taylor series expansion was used to acquire more accurate extrema locations, and extrema below a certain threshold would be rejected.

In Orientation assignment, the region surrounding the keypoints was used to calculate the gradient magnitude and direction of the region. An orientation histogram of 36 bins was formed using the gradient magnitude and a Gaussian-weighted circular window. The highest peak and peaks that were above 80% of the highest would be used to calculate the orientation. This step allowed for scale invariance as well as matching stability.

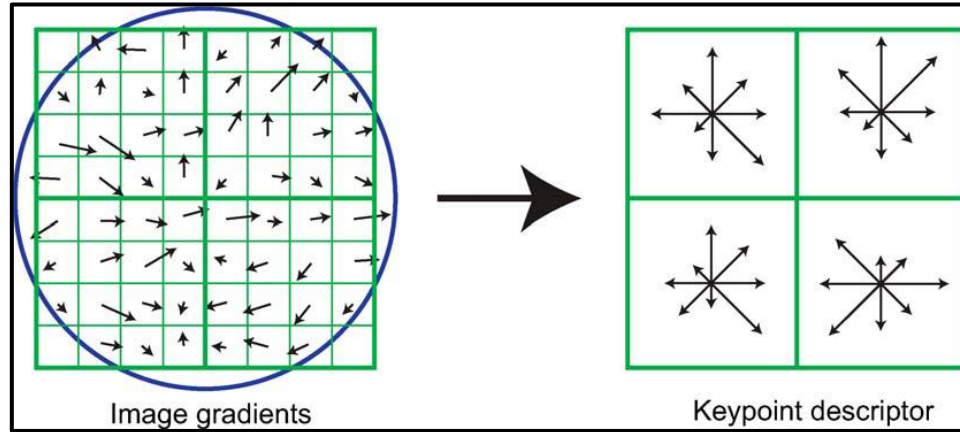


Figure 2- 2: Gradient magnitude and direction at each point (left) is used to find the keypoint descriptor (right) [1]

After Orientation assignment, each keypoint in the image would have a scale and orientation. In Keypoint descriptor, a descriptor was computed for the local image region surrounding the keypoint. The descriptor would be distinctive and invariant to changes in image views. A 16x16 window surrounding the keypoint would be formed, and this window would be further divided into sub-blocks of 4x4 size. Each sub-block would contain the corresponding sum of the gradient magnitudes in the region, as shown in Figure 2-2.

To perform object detection, keypoints were matched independently to the keypoint database obtained from training images. Clusters that contained at least 3 features that complied with an object and its pose would have a high probability of being a correctly identified object.

2.2.2 Speeded Up Robust Features (SURF)

$$\mathcal{H}(X, \sigma) = \begin{bmatrix} L_{xx}(x, \sigma) & L_{xy}(x, \sigma) \\ L_{xy}(x, \sigma) & L_{yy}(x, \sigma) \end{bmatrix} \quad (2-1)$$

SURF was another object detection algorithm that used box filters for real-time detection [2]. SURF was proposed as an improvement of SIFT by improving its speed and flexibility. A Hessian matrix was used for its detector. For a point $X = (x, y)$ in an image, the Hessian matrix in X at scale σ would be defined as $H(X, \sigma)$, as shown in Equation 2-1. $L_{xx}(X, \sigma)$ was the convolution of a Gaussian second order derivative at point X in an image.

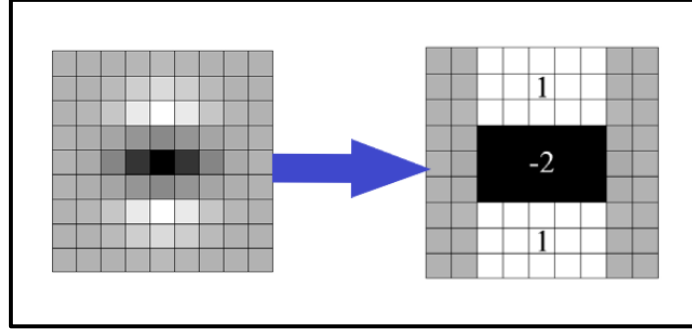


Figure 2- 3: Gaussian second order derivative in y-direction [2]

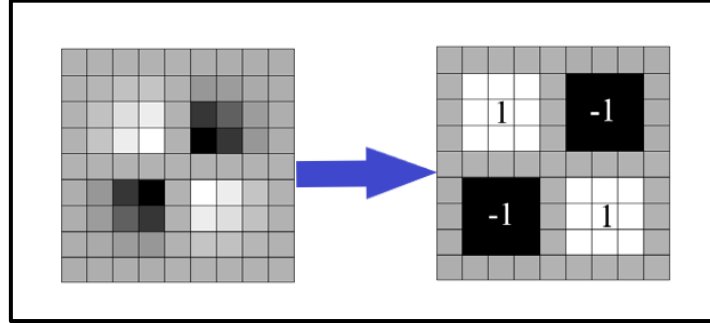


Figure 2- 4: Gaussian second order derivative in xy-direction [2]

The determinant of the Hessian matrix would be used to detect objects. In Figures 2-3 and 2-4, a 9x9 box filter was used to obtain approximations of the Gaussian second order derivatives at $\sigma = 1.2$. These approximations represent the highest spatial resolution and are denoted by D_{xx} , D_{yy} , and D_{xy} .

$$\det(\mathcal{H}_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2 \quad (2-2)$$

D_{xx} , D_{yy} , and D_{xy} would be used to calculate the determinant of the Hessian as shown in Equation 2.3. Areas on an image where this determinant is maximum would be chosen as points of interest. This information would be passed onto SURF's descriptor, which would generate the orientation of the points of interest and extract its features. These features would be used to identify objects in an image.

SURF was able to reduce the complexity of SIFT and increased the speed and accuracy of object detection.

2.3 Types of Neural Networks

Neural networks were inspired from the neural systems found in human brains. These neural networks have been used in many object detection and classification problems. In this section, the different types of neural networks would be explored.

2.3.1 Convolutional Neural Networks (CNN)

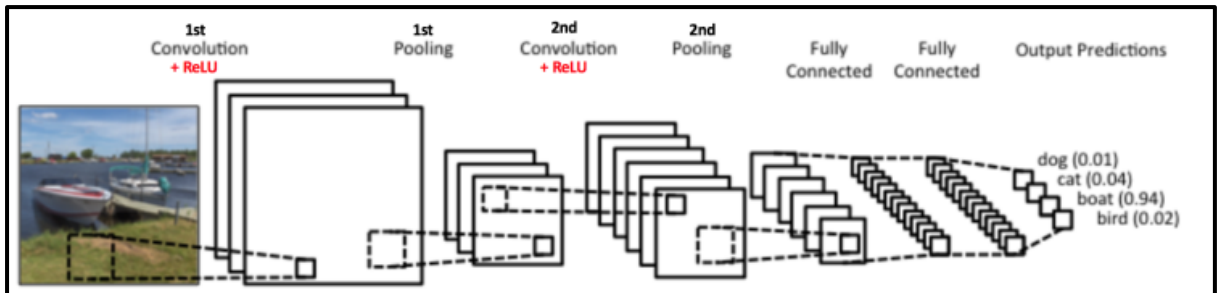


Figure 2- 5: Illustration of a sample CNN [3]

A CNN is an artificial neural network (ANN) that is widely used for object detection. CNNs have four main layers: Convolution, Rectified Learning Unit (ReLU), Pooling and Fully Connected, which is shown in Figure 2-5 [3]. The convolutional layer contained a collection of filters which were applied across the height and width of an input image [4]. The filters would have a response on a feature such as an edge. Thereafter, a 2-dimensional tensor was generated, which showed the response of the filters on every position of the input image. Next, the tensors were passed to the ReLU layer to introduce non-linearity into the network by replacing all negative pixel values in the tensor with zero. The tensors were then passed to the Pooling layer, which gradually reduces the spatial size of the tensor. This reduced the amount of computation and parameters in the network, which in turn controlled overfitting.

Depending on network specifications, there may be multiple convolutions, ReLU and pooling layers. Upon reaching the Fully Connected layer, the output of the previous layers would contain high-level feature representation of the input image. The Fully Connected layer would utilise these features to categorise the input image into the various categories that were trained into the network.

2.3.2 Region Based Convolutional Neural Network (R-CNN)

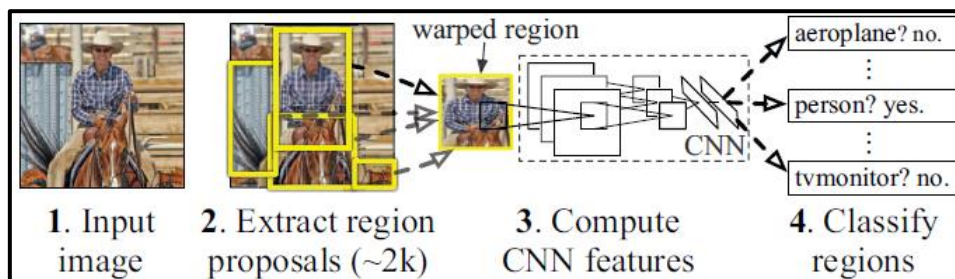


Figure 2- 6: R-CNN overview [5]

R-CNN was proposed to improve object detection efficiency and accuracy compared to previous known methods [5]. R-CNN makes use of supervised learning, where the model was trained using labelled data and the expected output is known. This was in contrast with unsupervised learning, where the model was only given unlabelled data and the correct output is not known to the model.

As shown in Figure 2-6, R-CNN consisted of 3 modules: region proposal, feature extraction, and classification. R-CNN utilised selective search to generate around 2000 region proposals in an input image, which were regions of interest and are categorically independent. These region proposals were used as input data for a CNN to extract features. The features would be input into a Support Vector Machine (SVM) to classify the objects present within the region proposal.

R-CNN was able to improve upon CNNs for object detection. However, it required long training time as it needed to classify 2000 region proposals. The trained model also required 49 seconds to detect an object using a Graphics Processing Unit (GPU), which meant that it was unable to be deployed for real-time detection.

2.3.3 You Only Look Once (YOLO)

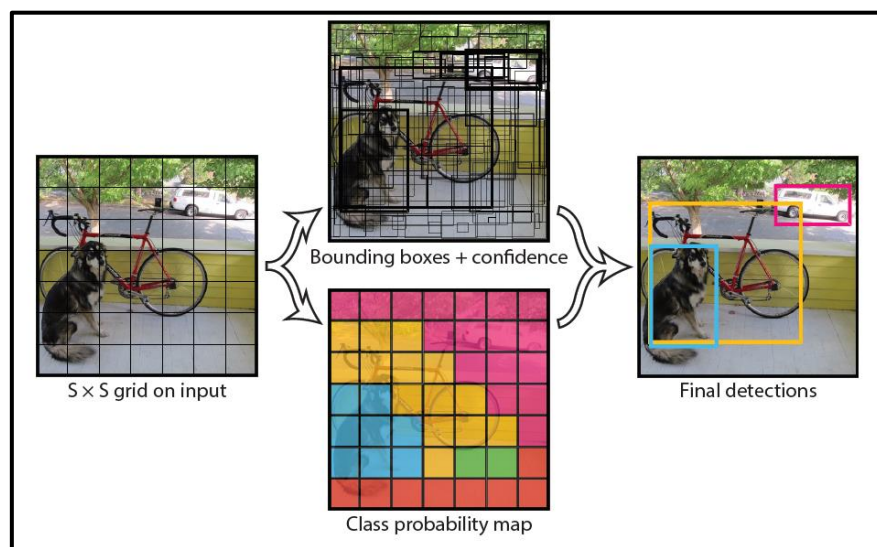


Figure 2- 7: YOLO's object detection overview [6]

YOLO [6] proposed another method for object detection. Instead of analysing the whole input image for the object, YOLO concentrated only on areas in the image with higher probabilities of having an object. As shown in Figure 2-7, YOLO first divided the input image into a 'S x S' region. If one of these grids contained the center of the object, that grid would be accountable for detecting that object. In each grid cell, bounding boxes, confidence of the boxes, and class probabilities were generated. If there are no objects in the grid, the class probability was zero.

If the class probability for a bounding box reaches above a threshold value, it would be used to detect the object.

YOLO achieved this using only a single convolutional network, which differed from R-CNN where the regions are detected before being input into the network. Although YOLO was able to achieve high detection speeds compared to other detection algorithms, it is inadequate in detecting small objects in an image.

2.3.4 Residual Neural Network (ResNet)

ResNet was a neural network proposed to solve the problem of vanishing gradient and degradation seen in many CNNs [7]. When a CNN contained many layers, the problem of vanishing gradient would arise, where the gradients obtained from the loss function would become extremely small during backpropagation. This meant that weights would have little to no change in value, and the model essentially halts its learning. Degradation arises when a neural network depth is increased, and accuracy becomes saturated and rapidly declines. This is because adding more layers to a neural network also leads to higher training error.

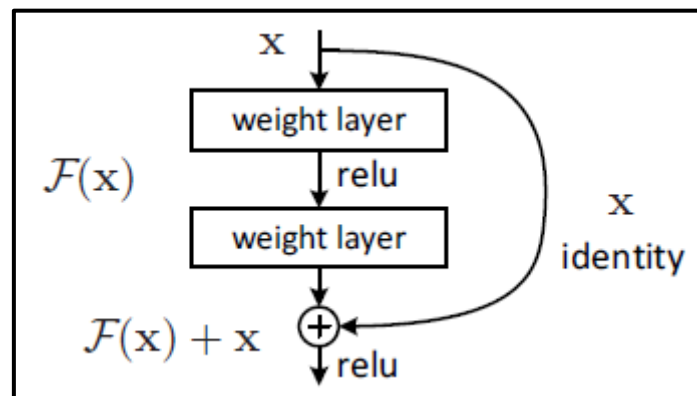


Figure 2- 8: Residual Building Block [7]

ResNet introduced the residual building block as shown in Figure 2-8, also known as bottleneck modules. ResNet's architecture consisted of multiple building blocks stacked upon each other that sequentially feeds information from one block to another. These blocks contain layers of different functions, and includes the ability to skip these layers, known as skip connections. Skipping layers allowed the activations from the previous layer to be reused until the adjacent

layer manages to learn its weights, thereby preventing the vanishing gradient problem. Skipping certain layers that do not affect network performance also allowed for shorter training times, and also reduced network depth, avoiding the problem of degradation.

2.3.5 ENet

ENet (efficient neural network) [8] was a network developed with fast inference and strong accuracy. It utilised the bottleneck modules seen in ResNet and other optimisation methods to improve performance.

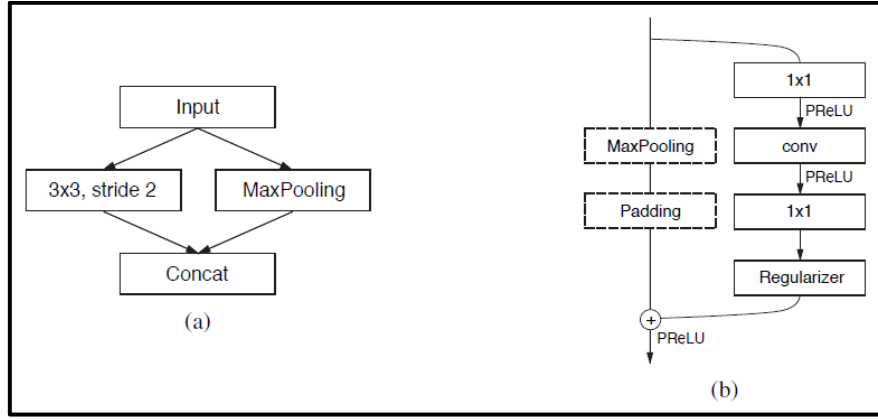


Figure 2- 9: (a) Initial Block of ENet, (b) Bottleneck module used in ENet [8]

ENet's initial stage consisted of the block shown in Figure 2-9 (a), where MaxPooling was done alongside a convolution of stride 2. Bottleneck modules, such as the example showed in Figure 2-9 (b), would follow the initial block. Each bottleneck module contained a main branch with separate branch of convolutional filters, which will be added into the main branch using element-wise addition. The convolutional branch first used a 1x1 projection to reduce dimensionality, followed by the main convolutional layer and a 1x1 expansion. Parametric ReLU (PReLU) and Batch Normalisation was performed between all convolutions. PReLU was used to improve training and inference time.

Name	Type	Output size
initial		$16 \times 256 \times 256$
bottleneck1.0	downsampling	$64 \times 128 \times 128$
4× bottleneck1.x		$64 \times 128 \times 128$
bottleneck2.0	downsampling	$128 \times 64 \times 64$
bottleneck2.1		$128 \times 64 \times 64$
bottleneck2.2	dilated 2	$128 \times 64 \times 64$
bottleneck2.3	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.4	dilated 4	$128 \times 64 \times 64$
bottleneck2.5		$128 \times 64 \times 64$
bottleneck2.6	dilated 8	$128 \times 64 \times 64$
bottleneck2.7	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.8	dilated 16	$128 \times 64 \times 64$
<i>Repeat section 2, without bottleneck2.0</i>		
bottleneck4.0	upsampling	$64 \times 128 \times 128$
bottleneck4.1		$64 \times 128 \times 128$
bottleneck4.2		$64 \times 128 \times 128$
bottleneck5.0	upsampling	$16 \times 256 \times 256$
bottleneck5.1		$16 \times 256 \times 256$
fullconv		$C \times 512 \times 512$

Figure 2- 10: ENet’s network architecture [8]

The main convolutional layer in each bottleneck module used regular, dilated, full convolutions, or asymmetric convolutions. Figure 2-10 shows ENet’s full network architecture, which mainly consisted of bottleneck modules of different main convolutional layers. Stages 1, 2 & 3 were the encoder, while stages 4 & 5 were the decoder.

ENet utilised several design choices for high speed and accuracy. Downsampling was performed during the first two blocks to decrease input size, and only a small set of feature maps were used. This was because much of the visual information was spatially unnecessary, and this early downsampling allowed for a more efficient representation.

Dilated convolutions were also used in downsampling, which provided a wider receptive field, allowing the network to gain more context. Dilated convolutions were also used in the bottleneck modules that were operating on small resolutions, which provided an increase in accuracy.

As a result, ENet was able to perform achieve fast and lower cost computations and had increased efficiency on high end GPUs.

2.4 Existing Lane Detectors

Lane detection is a fundamental function for autonomous vehicles. Therefore, many lane detection algorithms have been proposed to improve on lane detection accuracy and precision. In this section, existing lane detectors would be examined.

2.4.1 Spline Fitting Approach



Figure 2- 11: Spline fitting and localization [9]

One method proposed for lane detection was to take the top view of a road image and use line detection and random sample consensus (RANSAC) to detect the lanes [9]. In this method, inverse perspective mapping (IPM) was used to generate a top view of an input road image as shown in Figure 2-11. This allowed the perspective effect in the image to be removed, and the lanes would now appear vertical and parallel. The image was then filtered using a two-dimensional Gaussian kernel so that the image will only retain the lane markers.

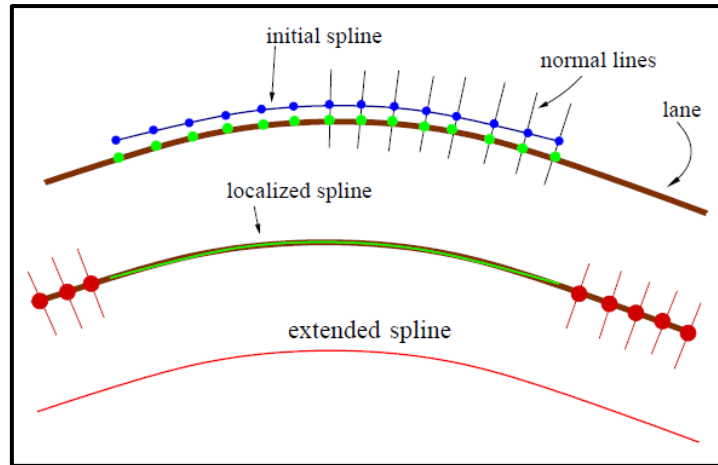


Figure 2- 12: Spline fitting and localization [9]



Figure 2- 13: Final output [9]

The number of lines were counted and grouped, and RANSAC line fitting was used to obtain a better fit for those lines. RANSAC spline fitting was then used on the lines to generate the best fit splines, which were the final detected lanes as shown in Figure 2-12. These splines were then projected onto the input image during post-processing as shown in Figure 2-13. However, this lane detection method assumed that lanes are close to parallel, which in reality is mostly not the case.

2.4.2 LaneNet

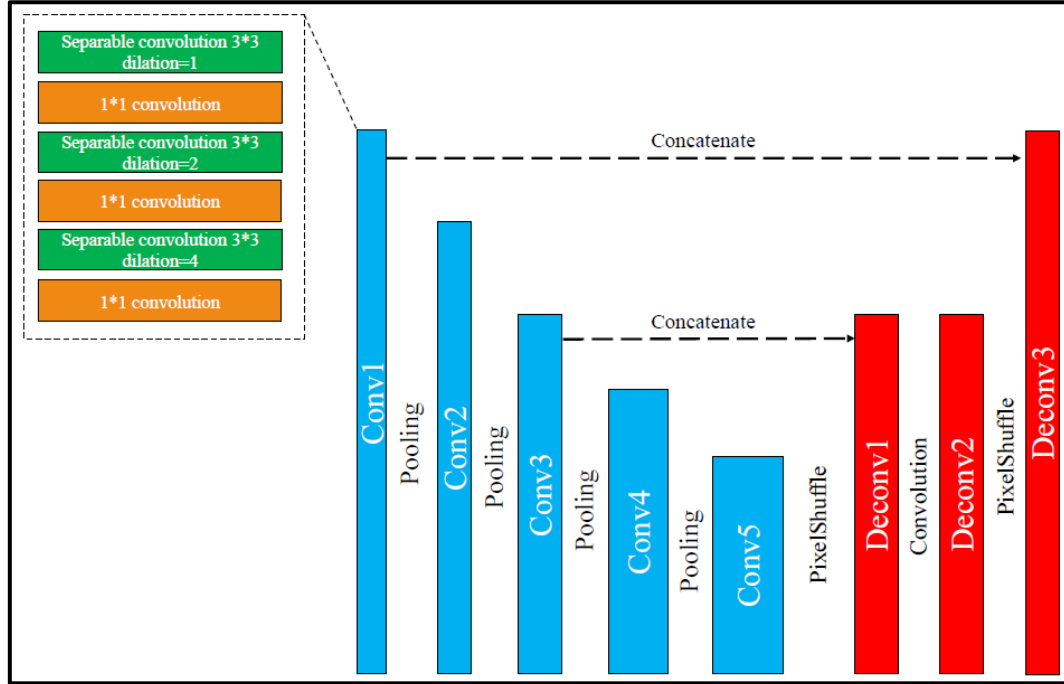


Figure 2- 14: Lane edge proposal architecture [10]

LaneNet was a deep neural network that accomplishes lane detection in two stages: lane edge proposal and lane line localisation [10]. During the lane edge proposal stage, binary classification would be done on every pixel of the input image to obtain lane edge proposals. This stage utilised an encoder-decoder architecture. The encoder would use an IPM image of a vehicle's front view as input to extract its features. IPM images were used for a reason similar with the previous detection method: to remove the perspective effect in the input image. Convolutional layers in the encoder made use of dilated convolutions to increase the receptive field. Thereafter, the decoder would recover the feature map's resolution and generate the lane edge proposal map. Figure 2-14 shows this stage's architecture, with blue blocks symbolising encoder layers and red blocks being decoder layers.

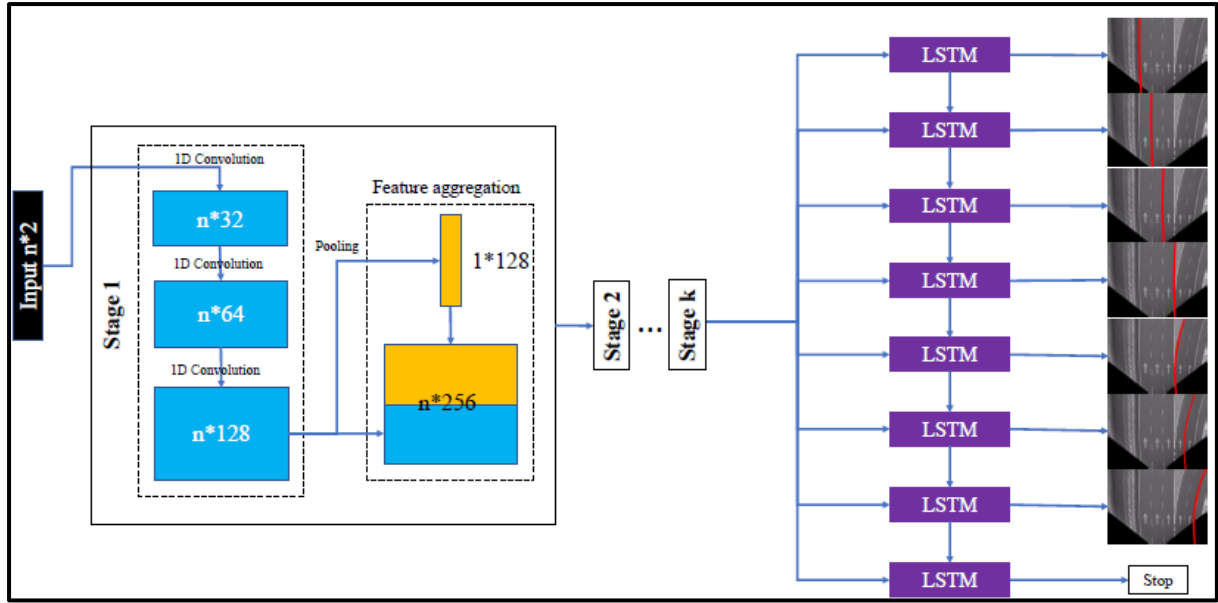


Figure 2- 15: Lane line localisation architecture [10]

In the lane line localisation stage, the network would detect the location and shape of lanes based on the lane edge proposal obtained from the previous stage. This stage also utilised an encoder-decoder architecture, as shown in Figure 2-15. In the encoder stages, one-dimensional convolutions and pooling operations encode the lane edge proposals into a low dimensional representation. The decoder is based on a Long Short-Term Memory (LSTM) network that gradually decoded the low dimension representation into individual lanes in the input image. The LSTM would predict the parameters for each lane and its confidence score. If the confidence score falls below a certain threshold, it signified that the network did not detect any more lanes in the input image.

By applying a two-stage approach, LaneNet was able to reduce computational cost due to its pipeline structure. Unlike the previous spline fitting approach, LaneNet did not need to assume that the lanes are parallel or close to parallel, allowing it to be more flexible with lane detection. However, LaneNet was not made with lane detection in different weather environments in mind, which an autonomous vehicle is sure to encounter.

2.4.3 VPGNet

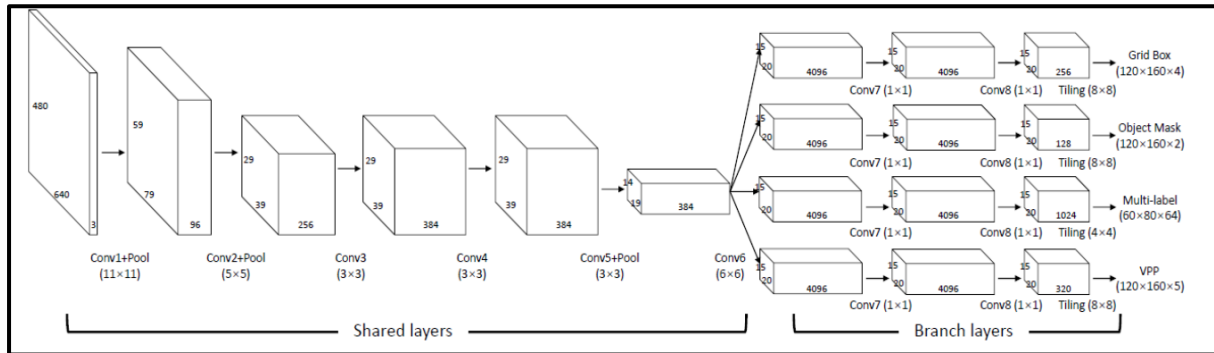


Figure 2-16: VPGNet’s architecture [11]

VPGNet was a multi-task network that was able to simultaneously employ lane and road marker detection and recognition [11]. The network was guided by a vanishing point (VP), which the authors defined as the nearest point on the horizon where the road lanes merge and disappear predictively around the farthest point of the visible lane. VP was used to give a global geometric context of the input image and infer the location of the lanes and road markings.

For lane and object detection, VPGNet ran the classification layer in parallel with a convolutional architecture. As shown in Figure 2-16, VPGNet utilised six shared convolutional layers before splitting into the 4 different tasks: grid box regression, object detection to obtain the local context, multi-label classification, and VP prediction (VPP) to obtain the global context. These tasks allowed the network to find and identify lane and road markings and predict the VP simultaneously in a single forward pass while also achieving high accuracy.

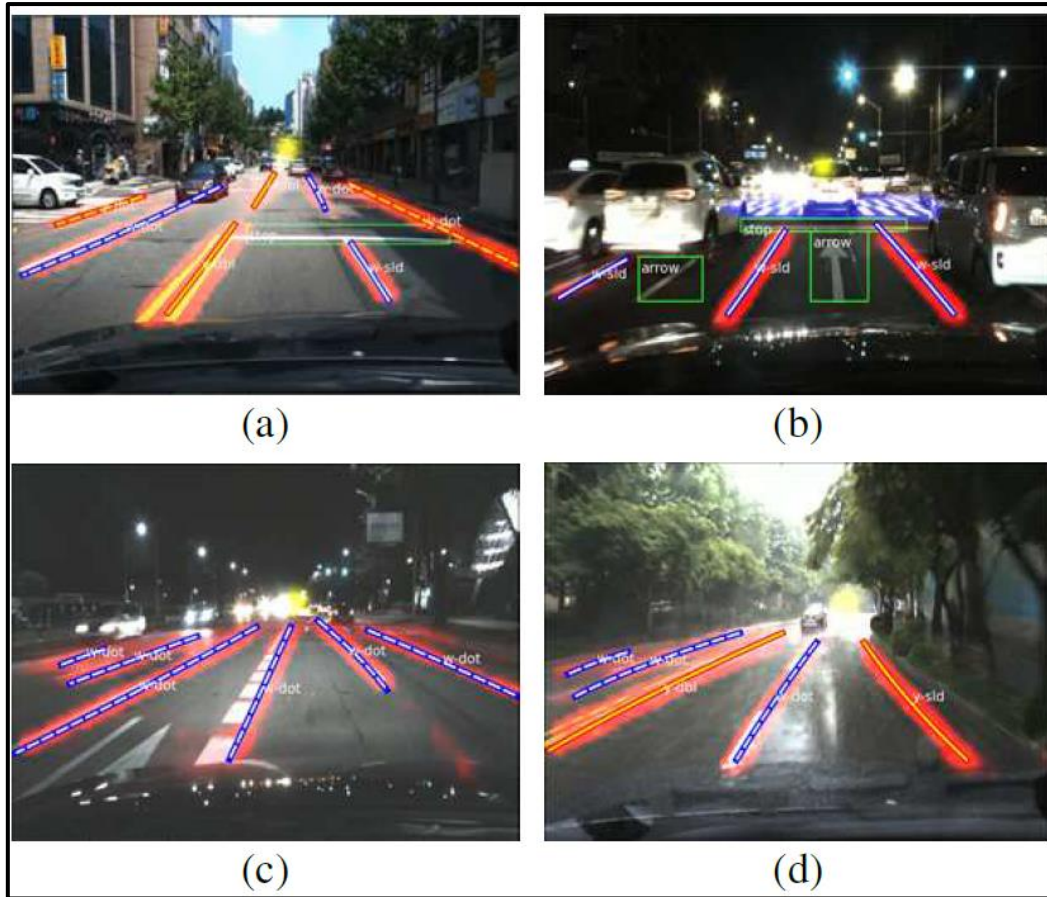


Figure 2- 16: VPGNet operating under (a) normal conditions, (b) with road markings, (c) night scenes, (d) rainy conditions [11]

By using VP as the global context, the network was able to do lane detection under many different environmental conditions, such as low illumination and rainy conditions as shown in Figure 2-17.

2.5 Concluding Remarks

In this chapter, the various object detection methods, CNN variants, and existing lane detection algorithms were reviewed and discussed.

Among the lane detectors, VPGNet was able to conduct lane detection in various weather conditions including rainy conditions. Therefore, VPGNet would be used as the baseline

network for this project. Its use of vanishing points to gather global and local context in an image may be effective in detecting lanes in rainy environments, where information is limited.

VPGNet can be further improved by utilising the bottleneck modules seen in ResNet and ENet to improve its learning process. Using dilated convolutions seen in ENet could also further increase the receptive field, allowing for more information to be gathered in VPGNet.

Chapter 3

Training and Testing VPGNet in Caffe Framework

3.1 Introduction

To judge VPGNet's effectiveness in lane detection, the network had to be trained and deployed on lane images. VPGNet's authors had released the network's code and dataset on their Github repository, which was built on the Caffe framework. In this section, the process of preparing VPGNet for training and testing would be explained.

3.2 Caffe Overview

Caffe is an open-source deep learning framework that aims for speed and modularity [12]. It used a C++ / CUDA architecture for deep learning and was able to seamlessly switch between central processing units (CPU) and graphics processing units (GPU) modes. Caffe networks

defined a model in a bottom-to-top approach, from input data to loss. A typical Caffe model consisted of blobs, layers, and nets. A Caffe blob was a wrapper that provides a unified memory interface over the data that was processed and passed along by Caffe. Layers were used to transform bottom blobs to top blobs. Nets consisted of many connected layers to define a function. Figure 3-1 shows how blobs, layers, and nets were combined to form a Caffe function.

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

Figure 3- 1: A sample Caffe net with 3 layers that define the top and bottom blobs [12]

3.3 VPGNet’s Caffe Structure

Layer	Conv 1	Conv 2	Conv 3	Conv 4	Conv 5	Conv 6	Conv 7	Conv 8
Kernel size, stride, pad	11, 4, 0	5, 1, 2	3, 1, 1	3, 1, 1	3, 1, 1	6, 1, 3	1, 1, 0	1, 1, 0
Pooling size, stride	3, 2	3, 2			3, 2			
Addition	LRN	LRN				Dropout	Dropout, branched	Branched
Receptive field	11	51	99	131	163	355	355	355

Figure 3- 2: VPGNet’s architectural details [11]

As illustrated in Figure 3-2, VPGNet’s first six shared convolutional layers consisted of different kernel sizes, strides and padding. This resulted in a larger receptive field by the end of the sixth convolutional layer before branching out into the different task branches. A network’s receptive field is size of the area in the input image that creates the output feature

[13]. A larger receptive field corresponds to more information being gathered by the network. Kernel size refers to the height and width of the filter, stride refers to the number of pixels the filter moves at each time, and pad refers to the number of pixels adding to the edges of the input image for that layer.

```
input: "data"
input_shape {
  dim: 1
  dim: 3
  dim: 480
  dim: 640
}
```

Figure 3- 3: Input size for VPGNet

VPGNet was able to take in Red-Green-Blue (RGB) images of 640 pixels height and 480 pixels width. As shown in Figure 3-3, the input data is represented as a “data” blob in Caffe and is the initial blob for the model.

```
layer {
  name: "L0"
  type: "Convolution"
  bottom: "data"
  top: "L0"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}

layer {
  name: "relu1"
  type: "ReLU"
  bottom: "L0"
  top: "L0"
}

layer {
  name: "norm1"
  type: "LRN"
  bottom: "L0"
  top: "norm1"
  lrn_param {
    k: 2
    local_size: 5
    alpha: 0.0005
    beta: 0.75
  }
}

layer {
  name: "pool1"
  type: "Pooling"
  bottom: "norm1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
```

Figure 3- 4: First convolutional layer in VPGNet

The input image was then passed to the first convolutional layer ‘L0’. Figure 3-4 shows the different Caffe layers and specifications used for the first convolutional layer. The kernel size, stride was listed under ‘convolution_param’. ReLU was done after convolution for each convolutional layer in VPGNet, with pooling and Local Response Normalisation (LRN) done depending on the layer.

ReLU is a function that introduces non-linearity into the network by replacing all negative values in a feature map with zero [14]. LRN normalises the values to a range between 0 and 1 across the channels [15]. In pooling, the feature maps are down sampled to obtain the most important information from them [16]. In this case of VPGNet, max pooling was performed in a 3x3 kernel. This means that the highest value in the 3x3 kernel would be selected in the pooling operation.

For optimisation, VPGNet utilised Stochastic Gradient Descent (SGD) with a batch size of 20 and momentum of 0.9. SGD will be further discussed in detail in Section 4.4.

3.4 Vanishing Points (VP)

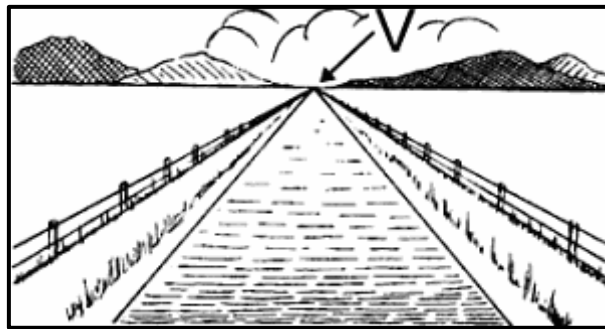


Figure 3- 5: Illustration of vanishing point [17]

According to Merriam-Webster, a vanishing point is defined as the point where receding parallel lines seemingly meet on a linear perspective [17] as shown in Figure 3-5. In the context of lane detection, lanes and road markings tend to converge at a single point even if the road is straight or curved.

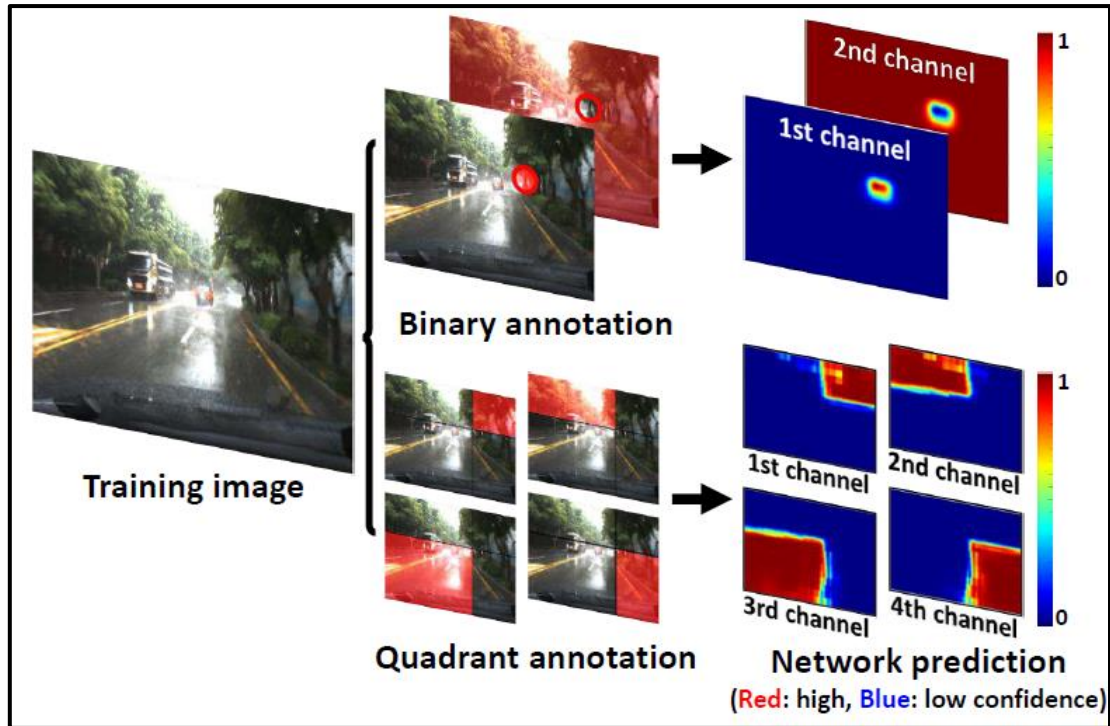


Figure 3- 6: Output from VP task [11]

For the VPP task in VPGNet, a quadrant mask was used to divide the input image into four areas. The intersection of the four areas would represent the VP, and the VP could be deduced from this intersection. The VP task contained five channels of output: four quadrant channels and one absence channel. Each quadrant channel would represent one quadrant area on the input image. The absence channel depicted all pixels that were not the VP, and if the VP is difficult to identify, every pixel would be classified as the absence channel.

The VP task utilised cross entropy loss since it was able to stabilise the gradients that were obtained from each detection task. Therefore, a binary classification method was used to classify the area on the image that contained the VP and the background. As shown in Figure 3-6, the training image would undergo binary classification and quadrant annotation to deduce the location of the VP.

3.5 Data Preparation

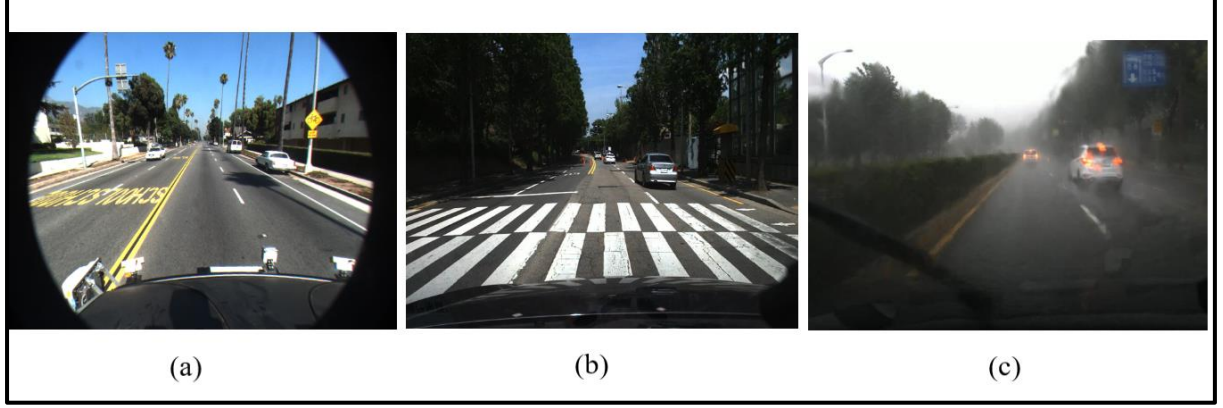


Figure 3- 7: Sample images from (a) Cordova, (b) VPGNet non-rainy, (c) VPGNet rainy dataset

For this project, two lane datasets would be used to evaluate VPGNet. The Cordova dataset [18], which was also used in the spline fitting approach, would be used to evaluate the non-rainy lane detection capabilities of VPGNet. VPGNet’s authors used their own dataset for the network, which included both rainy and non-rainy conditions. This would be used to determine VPGNet’s lane and road marking detection capabilities in both rainy and non-rainy environments. Figure 3-7 shows the sample images from each dataset.

/cordova2/f00001.png	113	33	257	40	264	1	41	249	48	256	1	41	257	48	264	1	41	305	
48	312	3	49	249	56	256	1	49	305	56	312	3	57	249	64	256	1	57	305
64	312	3	65	241	72	248	1	65	249	72	256	1	65	297	72	304	3	73	289
80	296	3	73	297	80	304	3	81	241	88	248	1	81	289	88	296	3	89	241
96	248	1	89	281	96	288	3	89	289	96	296	3	97	233	104	240	1	97	281
																			105

Figure 3- 8: Sample annotations from Cordova dataset

The Cordova dataset had 3 classes annotated, solid white lanes, broken white lanes, and yellow lanes. In the Cordova dataset, images were in PNG format with annotations stored in a ‘labels.ccvl’ file. MATLAB scripts would have to be used to extract these annotations for training purposes. One Github user was able to extract these annotations and uploaded them onto their repository [19]. The annotations would be saved in a txt format as shown in Figure 3-8.

/000031.png	355	312	192	320	200	16	320	192	328	200	16	264	200	272	208	17	272	200		
280	208	17	288	200	296	208	16	296	200	304	208	16	304	200	312	208	16	312	200	320
208	16	160	208	168	216	15	168	208	176	216	15	176	208	184	216	15	184	208	192	216
15	192	208	200	216	15	200	208	208	216	15	208	208	216	216	15	216	208	224	216	15
224	208	232	216	15	232	208	240	216	15	240	208	248	216	15	248	208	256	216	15	256
208	264	216	15	264	208	272	216	15	272	208	280	216	15	280	208	288	216	15	288	208
296	216	15	296	208	304	216	15	304	208	312	216	15	312	208	320	216	15	320	208	328
216	15	328	208	336	216	15	336	208	344	216	15	344	208	352	216	15	352	208	360	216
15	360	208	368	216	15	368	208	376	216	15	376	208	384	216	15	384	208	392	216	15

Figure 3- 9: Sample annotations from VPGNet dataset

Table 3- 1: Lane and road marking classes annotated in VPGNet dataset

Class
background
lane_solid_white
lane_broken_white
lane_double_white
lane_solid_yellow
lane_broken_yellow
lane_double_yellow
lane_broken_blue
lane_slow
stop_line
arrow_left
arrow_right
arrow_go_straight
arrow_u_turn
speed_bump
crossWalk
safety_zone
other_road_markings

The VPGNet dataset had a total of 17 lane and road marking classes annotated. Table 3-1 shows the different classes annotated for VPGNet dataset. For the VPGNet dataset, the data was

in .mat format. The annotations for the VPGNet dataset were extracted using a separate script. The .mat file saved the image in a [height, width, 5] format, with 5 corresponding to 5 channels. The third channel contains the annotations for the different classes for the multi-label task, and thus the pixelwise information from that channel is saved in as a txt file. Figure 3-9 shows a sample of the annotations extracted from the VPGNet dataset.

```

# Declare $PATH_TO_DATASET_DIR and $PATH_TO_DATASET_LIST

../../build/tools/convert_driving_data $PATH_TO_DATASET_DIR $PATH_TO_DATASET_LIST LMDB_train
../../build/tools/compute_driving_mean LMDB_train ./driving_mean_train.binaryproto lmbd

../../build/tools/convert_driving_data $PATH_TO_DATASET_DIR $PATH_TO_DATASET_LIST LMDB_test

```

Figure 3- 10: ‘make_lmdb.sh’ file

To train the data, the datasets had to be split between training and testing sets their paths declared in the ‘make_lmdb.sh’ file as shown in Figure 3-10. Running the file would create Lightning Memory-Mapped Database (LMDB) files which would be used for the training.

3.6 Training & Testing Environment

+-----+ NVIDIA-SMI 455.23.05 Driver Version: 455.23.05 CUDA Version: 11.1 +-----+									
GPU		Name	Persistence-M		Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap			Memory-Usage	GPU-Util	Compute M.	MIG M.
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+									
0	GeForce	RTX 206...	On		00000000:09:00.0	On			N/A
0%	46C	P0	40W / 184W		217MiB / 7973MiB		19%	Default	N/A
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+									
+-----+ Processes: +-----+									
GPU	GI	CI	PID	Type	Process name			GPU Memory	
	ID	ID						Usage	
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+									
0	N/A	N/A	1157	G	/usr/lib/xorg/Xorg			105MiB	
0	N/A	N/A	2038	G	/usr/bin/gnome-shell			110MiB	
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+									

Figure 3- 11: Verification of CUDA installation

To begin the evaluation of VPGNet, which was built using the Caffe framework, VPGNet’s Caffe codes needed to be built and installed. For this project, my personal desktop computer would be configured to train VPGNet with the following technical specifications:

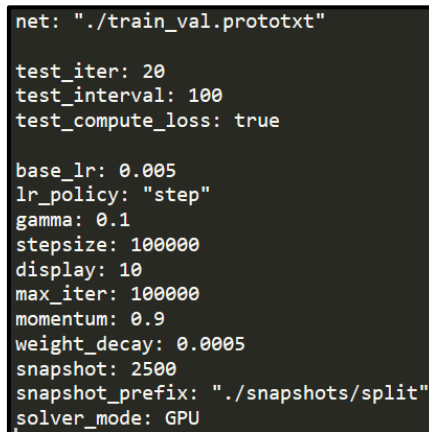
- 2176 CUDA core Nvidia RTX 2060 SUPER GPU

- 8-core AMD Ryzen 7 3700X 64-bit Processor
- 16GB DDR4 Memory

As shown in Figure 3-11, CUDA 11.1 was used with driver version 455.23.05. Ubuntu 20.04 was used as the operating system for this project. After Ubuntu and CUDA had been installed, VPGNet's Caffe codes would be built using CMake. This configuration was used to train VPGNet on the Cordova dataset.

Another configuration was set up to train VPGNet on the VPGNet dataset. Training for this dataset was done on Google Cloud Platform with the following technical specifications:

- 2560 CUDA core Nvidia Tesla P4 GPU
- 4-core n1-standard-4 CPU
- 15GB Memory



```
net: "./train_val.prototxt"

test_iter: 20
test_interval: 100
test_compute_loss: true

base_lr: 0.005
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 10
max_iter: 100000
momentum: 0.9
weight_decay: 0.0005
snapshot: 2500
snapshot_prefix: "./snapshots/split"
solver_mode: GPU
```

Figure 3- 12: ‘solver.prototxt’ file

For training, ‘solver.prototxt’ in VPGNet’s repository contained the training parameters for the model to be used for training. As shown in Figure 3-12, it contained parameters such as starting learning rate and maximum number of iterations. In this project, VPGNet was trained up to 100,000 iterations for both datasets.

3.7 Training Results and Discussion

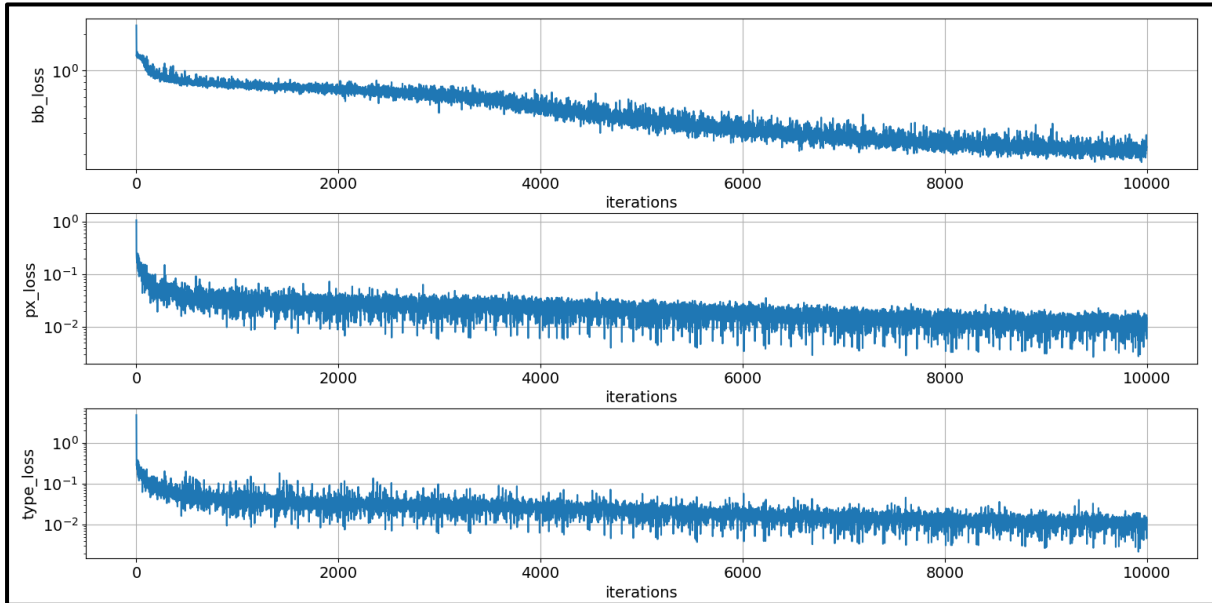


Figure 3- 13: Training Loss for Cordova trained model (Logarithmic scale)

```

Iteration 100000, loss = 0.742792
Iteration 100000, Testing net (#0)
Test loss: 1.75046
  Test net output #0: bb-loss = 0.554751 (* 3 = 1.66425 loss)
  Test net output #1: pixel-acc = 0.988739
  Test net output #2: pixel-loss = 0.0363826 (* 1 = 0.0363826 loss)
  Test net output #3: type-acc = 0.987321
  Test net output #4: type-loss = 0.0498195 (* 1 = 0.0498195 loss)
Optimization Done.

```

Figure 3- 14: Final Loss values for Cordova trained model at iteration #100,000

The Cordova dataset contained about 1,200 images in total. 80% of the dataset was used for training, while the other 20% was used for testing. The model was trained up to 100,000 iterations. As shown in Figure 3-13, the loss values for the 3 different tasks of VPGNet decreased as the number of iterations increased. Figure 3-14 shows the final loss values acquired at iteration #100,000, which was at an overall 0.742. In the original paper, VPGNet would first train its VPP task and subsequently train all the tasks using the kernels that were initialised from the first phase. However, since the VP module was not included in VPGNet's Github repository, the VP task is not trained in the Caffe model and the remaining tasks were trained together.

‘bb-loss’ corresponded to the grid box regression task, which was used to identify road marking classes. ‘bb-loss’ had a higher rate of change after iteration 3000 and had a steady decrease after iteration 6000. ‘pixel-loss’ referred to the object detection task, which produced the object mask. ‘type-loss’ refers to the multi-label classification task, which was used to detect lane classes and road marking classes. Both ‘pixel-loss’ and ‘type-loss’ had a general decreasing trend early in the training after iteration 500.

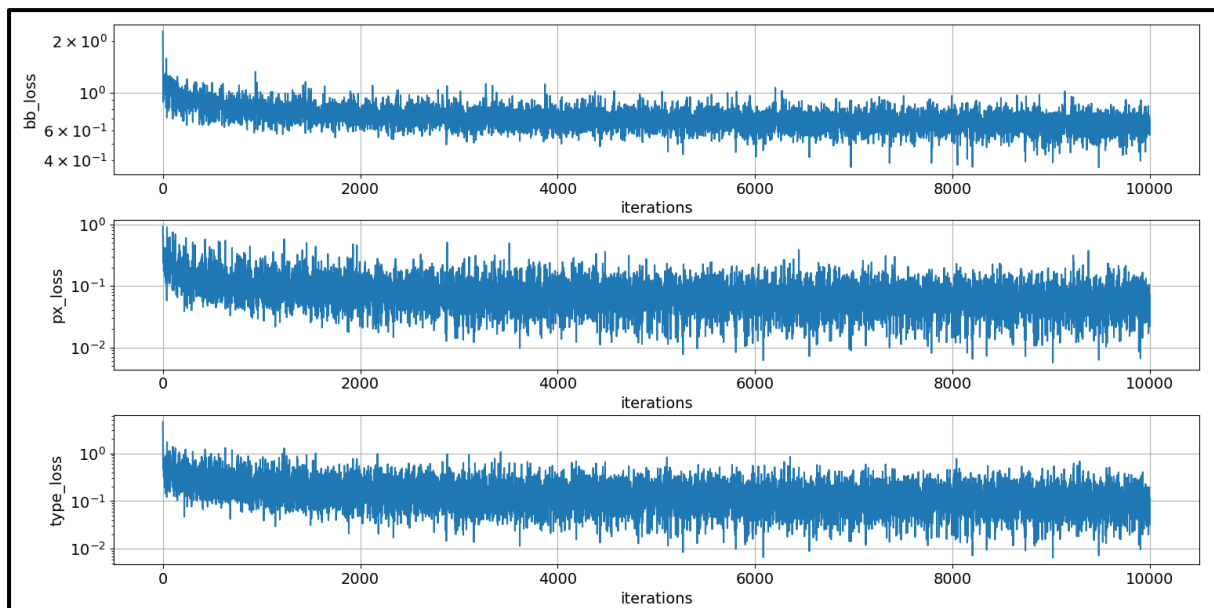


Figure 3- 15: Training Loss for VPGNet trained model (Logarithmic scale)

```
Iteration 100000, loss = 1.8183
Iteration 100000, Testing net (#0)
Test loss: 2.01138
  Test net output #0: bb-loss = 0.619193 (* 3 = 1.85758 loss)
  Test net output #1: pixel-acc = 0.975237
  Test net output #2: pixel-loss = 0.0574847 (* 1 = 0.0574847 loss)
  Test net output #3: type-acc = 0.966916
  Test net output #4: type-loss = 0.0963119 (* 1 = 0.0963119 loss)
Optimization Done.
```

Figure 3- 16: Final Loss values for VPGNet trained model at iteration #100,000

Figure 3-15 shows the loss values for the model trained using VPGNet dataset. The VPGNet dataset contained about 20,000 images; 80% of the data was used for training, 10% was used for testing, and the final 10% was used for validation. Throughout training, the loss values for the 3 different tasks had fluctuated more compared to the Cordova dataset, but also had a general decreasing trend. This may have been due to the VPGNet dataset being much larger

than the Cordova dataset. Figure 3-16 shows the final loss values acquired at iteration #100,000, which was at an overall 1.8183.

3.8 Deploying VPGNet for Lane Detection

```
def load_image(self, filename):
    self.filename = filename
    self.img = caffe.io.load_image(filename)
    print self.img.shape
    transformer = caffe.io.Transformer({'data': self.net.blobs['data'].data.shape})
    transformer.set_transpose('data', (2, 0, 1)) # move image channels to outermost dimension
    transformer.set_raw_scale('data', 255) # rescale from [0, 1] to [0, 255]
    transformer.set_channel_swap('data', (2, 1, 0)) # RGB -> BGR
    self.transformed_img = transformer.preprocess('data', self.img)
    print self.transformed_img.shape
    self.net.blobs['data'].data[...] = self.transformed_img
```

Figure 3- 17: Function to process an image for VPGNet

Caffe has many requirements for the input image before it can be used for testing. However, VPGNet’s authors did not provide a method for deploying VPGNet on any input data. A user on Github [19] was able to write their own deploy script for VPGNet, and their code was referenced for the deploy script used in this project. VPGNet was able to accept images of 640 pixels width and 480 pixels height in the 3-channel RGB format in either JPG or PNG. Input images must first be processed before VPGNet was able to read it.

Figure 3-17 shows the function used to load and process the input image. After loading the image file, the ‘Transformer’ function is instructed about the expected shape for the input. The ‘set_transpose’ function is used to change the image format from Height-Width-Channel (HWC) into Channel-Height-Width (CHW) format. The input is then rescaled from binary to [0, 255], since VPGNet used that scale. Thereafter, the RGB channels are swapped to a Blue-Green-Red (BGR) sequence, since Caffe was only able to accept an image in BGR format. The ‘preprocess’ function then executes the previously defined parameters on the input image. The previously mentioned steps were necessary to process an image that is readable by Caffe.

After the input image had been preprocessed, it was forwarded to VPGNet for deployment. Although VPGNet’s authors had specified four separate tasks for the network, the VPP was not included in their Github repository [20]. VPGNet’s authors had made the binary task for

binary classification, which was mainly used together with the VPP task. For the object detection task, VPGNet's authors used it to gather a local context of the image and was used in conjunction with the VPP, which obtained the global context. Although the task was meant for object detection, no useful information could be gathered from the task's output even after many weeks of testing. Since both binary and object detection tasks required the VPP to fully realise its potential, the multi-label task was used to deploy VPGNet since it contained most of the classification information and could be used for lane detection.

```
def visualize(self, num):
    # visualize classification
    original_img = cv2.imread(self.filename)
    original_img = cv2.resize(original_img, (640, 480))
    classification = self.net.blobs['multi-label'].data
    y_offset_class = 1 # offset for classification error
    x_offset_class = 1
    grid_size = self.img.shape[0]/60

    for i in range(60):
        for j in range(80):
            max_value = 0
            maxi = 0
            # Finding max value
            for k in range(64):
                if classification[0, k, i, j] > max_value:
                    max_value = classification[0, k, i, j]
                    maxi = k
            if maxi != 0:
                pt1 = ((j + y_offset_class)*grid_size, (i+x_offset_class)*grid_size)
                pt2 = ((j + y_offset_class)*grid_size+grid_size, (i+x_offset_class)*grid_size+grid_size)
                cv2.rectangle(original_img, pt1, pt2, color_options(maxi), 2)
```

Figure 3- 18: Function that outputs detect lane locations onto input image

The information from the multi-label task was extracted by the function shown in Figure 3-18. After forwarding the input image into VPGNet, the multi-label task would produce an output of 1x64x60x80 dimensions. This represented 64 channels of 60 pixels height and 80 pixels width. Offset values were included to adjust the boxes drawn to better fit the input image. The 'Grid_size' variable was used during up sampling of the output. To detect the lanes, the function searched through the 64 channels in one pixel of data to find the maximum value. This maximum value represented the lane class that was present in that pixel and was used to draw the location of the lane on top of the input image.

The image process and visualisation methods were used deploy VPGNet on both the Cordova and VPGNet datasets to evaluate its lane detection efficacy.

3.9 Concluding Remarks

In this chapter, VPGNet was further examined, and the model was trained using the Cordova and VPGNet dataset. The two datasets were used to evaluate model under non-rainy and rainy conditions.

The data preparation, training process, and deployment of VPGNet were also discussed. Although not discussed in detail in this report, setting up the environment to run Caffe was the most time-consuming part of this project due to Caffe requiring many dependencies, which in turn also required other dependencies to be installed and troubleshooted. VPGNet's authors also used Python 2.7 as part of their project, which had already been deprecated on 1 January 2020. This meant that some of the dependencies for VPGNet were no longer available or supported. VPGNet's authors also did not include a script for deploying their network, and a great deal of time was spent trying to deploy VPGNet due to the many intricacies in processing the input data for Caffe and obtaining the relevant information from VPGNet.

Caffe strengths lie in its modularity. However, the difficulty in setting up the Caffe environment is steep especially for new users.

Chapter 4

Proposed Model in PyTorch Framework

4.1 Introduction

For this project, another deep learning framework would be used to test VPGNet. PyTorch was selected due to its ease of set up and usage. In this section, the proposed PyTorch model would be discussed. Other components and parameters available in PyTorch that could be used to replace the existing techniques in VPGNet would also be explored.

4.2 PyTorch Overview

PyTorch is a Python library used for deep learning. It is a deep learning framework designed to be integrated into Python, which makes it easier to transfer other essential Python modules, such as NumPy and scipy, into PyTorch [21].

PyTorch provides a GPU-ready tensor library which can run on a CPU or GPU environment. Tensors in PyTorch are similar to NumPy ndarrays, with only a slight syntax difference.

PyTorch also contains minimal framework overhead, and popular acceleration libraries like cuDNN are already integrated to optimise speed. Memory allocation efficiency is maximised in PyTorch, which allows for larger deep neural networks to be trained.

Compared to Caffe, PyTorch is easier to set up and its interface is more intuitive. PyTorch does not need to compile codes to run it, unlike Caffe.

4.3 Loss Functions

When training a deep learning model, the model's error needs to be constantly approximated and evaluated [22]. A loss function would be used to approximate this error so that the model's current weights can be revised to reduce error in the following iteration. In VPGNet, the authors had used cross entropy loss for their VPP task. This section will cover the various loss functions included in PyTorch's library that were considered for the PyTorch model's design.

4.3.1 L1 Loss

L1 loss, also known as Mean Absolute Error (MAE), is a loss function included in PyTorch's library [23]. It measured the average of the absolute difference between predicted and expected output. L1 loss is able to review the absolute size of the error within a set of predicted values, therefore it is able to detect any outlier values that are very different from the mean value.

4.3.2 L2 Loss

L2 loss, also known as Mean Squared Error (MSE), loss measures the average of squared differences between predicted and expected output [24]. By squaring the differences, the complexity from negative values is reduced. Furthermore, squaring will highlight large errors made by the model, which can help to identify and correct these errors.

4.3.3 Binary Cross Entropy with Logits Loss

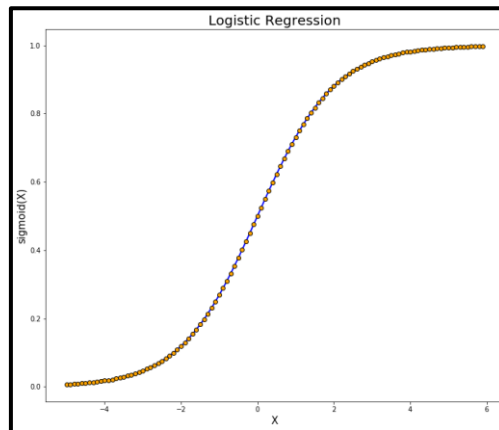


Figure 4- 1: Sigmoid function [26]

Binary Cross Entropy (BCE) with Logits Loss adds a sigmoid activation BCE Loss [25]. A sigmoid function would convert the output of a model into values between 0 and 1. BCE Loss compares the predicted probability of a class with the expected output of either 0 or 1. The loss value would be computed from the difference of the predicted and expected probability. BCE loss is able to correct extreme outliers, such as the case of the model being very confident but incorrect in its prediction.

Logistic regression utilises a sigmoid function, where it is used to fit the data as shown in Figure 4-1. Logistic regression is usually used for binary classification, and the probability of the class being present is plotted on the sigmoid function.

4.4 Stochastic Gradient Descent (SGD) with Momentum

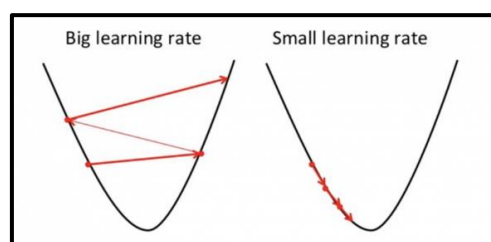


Figure 4- 2: Illustration of learning rate affecting how the model learns [27]

Stochastic gradient descent (SGD) is a common algorithm for optimising neural networks by finding the best values for the network's weights. In gradient descent, the optimal value is found by reaching the global minimum point of the loss function. Firstly, a random value is chosen, and a step is taken in the direction of the function where the gradient is decreasing until a value close to the global minimum is reached. The size of this step is known as the learning rate. As shown in Figure 4-2, a smaller learning rate would be preferred, since a large learning rate means that the next updated value would rebound across the function, making more difficult to reach the minima.

Gradient descent would have to go through the entire dataset to make a single step. However, SGD would only take a small number of training samples to make a single step. This means that for larger datasets, SGD is able to achieve lower computational cost and can converge at the minima faster than gradient descent.

However, SGD has difficulties in traversing near local minimum points. Its steps would start to slow down around local minimum points, which will impede learning. Momentum was added to accelerate SGD by increasing learning rate updates when it is in the appropriate direction and decreases it when the direction changes [28]. This is similar to momentum in physics, where a ball rolling down a hill would increase its speed the further it rolls down due to the gaining momentum. This allows SGD to have increased speed while having lesser oscillation around local minimum points.

4.5 Dilated Convolutions

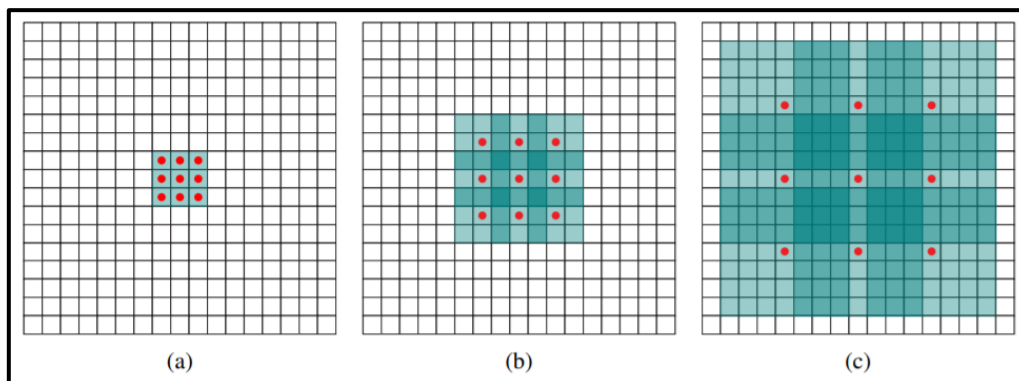


Figure 4- 3: (a) 1-dilation, (b) 2-dilation, and (c) 4-dilation convolutional outputs [29]

Dilated convolution is able to increase the receptive field by increasing the space between each cell in an image [29]. As shown in Figure 4-3, filters of varying dilation sizes can be used to increase the receptive field of an image. 1-dilation would be similar to a conventional convolution, where there is no spacing between cells. 2-dilation would create a 1-cell spacing as shown in Figure 4-3 (b), vice versa. By increasing the receptive field, more information can be gathered by the network.

4.6 Proposed Model

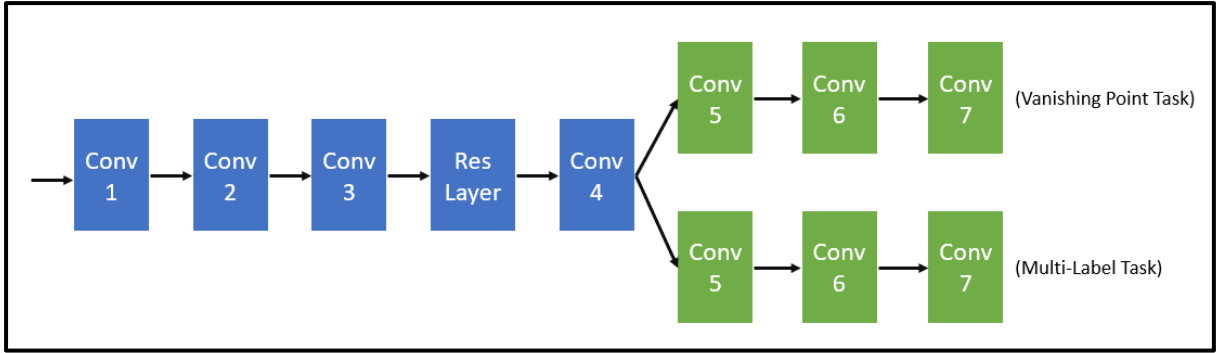


Figure 4- 4: Proposed PyTorch model

Table 4- 1: Proposed PyTorch model's layer information

Layer	Conv 1	Conv 2	Conv 3	Res	Conv 4	Conv 5	Conv 6	Conv 7
Type	Conv2d	Conv2d	Conv2d	5 x ResBlock	Conv2d	UpConv	UpConv	UpConv
Kernel	5	3	3	3	3	3	3	3
Stride	5	1	1	1	1	1	1	1
Padding	2	2	2	1	1	1	1	1
Dilation	2	2	2	1	1	1	1	1
Addition	LRN	LRN		ReLU				

Figure 4-4 shows the proposed PyTorch model for VPGNet with encoder layers shown in blue and decoder layers shown in green. Shared layers are the encoder, while individual tasks are

the decoder. The proposed model includes the VPP and the multi-label task. The bounding box and object detection task which were originally included into VPGNet may be included in the future once their purpose has been verified.

VPGNet's original network did not utilise dilated convolutions during encoding. Therefore, the PyTorch model proposed dilated convolutions to be used for the first 3 shared layers to increase the receptive field of the model as shown in Table 4-1. Dilated convolutions of scale 2 would be used. This would allow the model to obtain more context from the input image, which is especially useful for rainy images where information is limited.

Instead of using convolutional layers throughout the shared layers, the PyTorch model included residual blocks. As shown in Figure 4-4, the fourth shared layer comprised of five residual blocks with each block utilising ReLU activation.

For the VP task, the PyTorch model will be using a similar approach as VPGNet, which uses the absence channel and four quadrant channels.

SGD Optimiser was used in the PyTorch model, which was the same optimiser being used in the Caffe model.

4.7 Concluding Remarks

In this chapter, the proposed PyTorch model was described, and several components and parameters that could replace the ones used in VPGNet were discussed. The various loss functions would be implemented onto the model to gauge which would be most suited for the model. Other improvements could be further explored in the future to further improve the model's accuracy.

Chapter 5

Results from Caffe and PyTorch Frameworks

5.1 Introduction

In this chapter, the F1 score evaluation metric will be described in detail. The results from deploying the VPGNet Caffe model on the two datasets will be reviewed for non-rainy and rainy conditions. Finally, samples of output from the PyTorch model using different loss functions would be inspected.

5.2 F1 Score Calculation

Table 5- 1: Confusion Matrix

		Predicted	
		Positive	Negative
Actual	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

The F1 score can be used to determine the accuracy of the model. It provides a harmonic mean between precision and recall [30], which are both derived from the confusion matrix in Table 5-1. In general, a high F1 score corresponds to better detection.

$$Recall = \frac{TP}{TP+FN} \quad (5-1)$$

$$Precision = \frac{TP}{TP+FP} \quad (5-2)$$

Recall measures the model's capability for identifying all positive instances as shown in Equation 5-1. It takes the proportion of correctly predicted positives over the number of actual positives. Precision measures the model's capability for identifying only the relevant instances as shown in Equation 5-2. It takes the proportion of correctly predicted positives over the total number of predicted positives.

$$F1 = 2 * \frac{Precision*Recall}{Precision+Recall} \quad (5-3)$$

The F1 score would give equal weights to both precision and recall as shown in Equation 5-3, and extreme values would be punished. For example, a model can have an excellent recall of 1 but a poor precision of 0, which would result in an F1 score of 0. Therefore, F1 score can help to achieve an equilibrium between precision and recall.

To determine the accuracy of VPGNet, the F1 score calculation was done in the same way as the authors had described. The minimum distance from the center of each grid cell to the sampled lane point is measured. If this distance is within a threshold boundary R, the grid cell would be labelled as detected and the sampled point would be marked as true positive.

5.3 Results from Caffe Model

This section details the results obtained when deploying the Caffe VPGNet model on the Cordova and VPGNet datasets. The VPGNet model trained on the Cordova dataset was deployed on the Cordova data, while the VPGNet model trained on the VPGNet dataset was deployed on the VPGNet data. These results include the location of detected lanes and their

corresponding F1 score. For the VPGNet dataset, a ‘Detected?’ column was included to illustrate whether the class with an F1 score was reflected in the output image.

5.3.1 Cordova Dataset

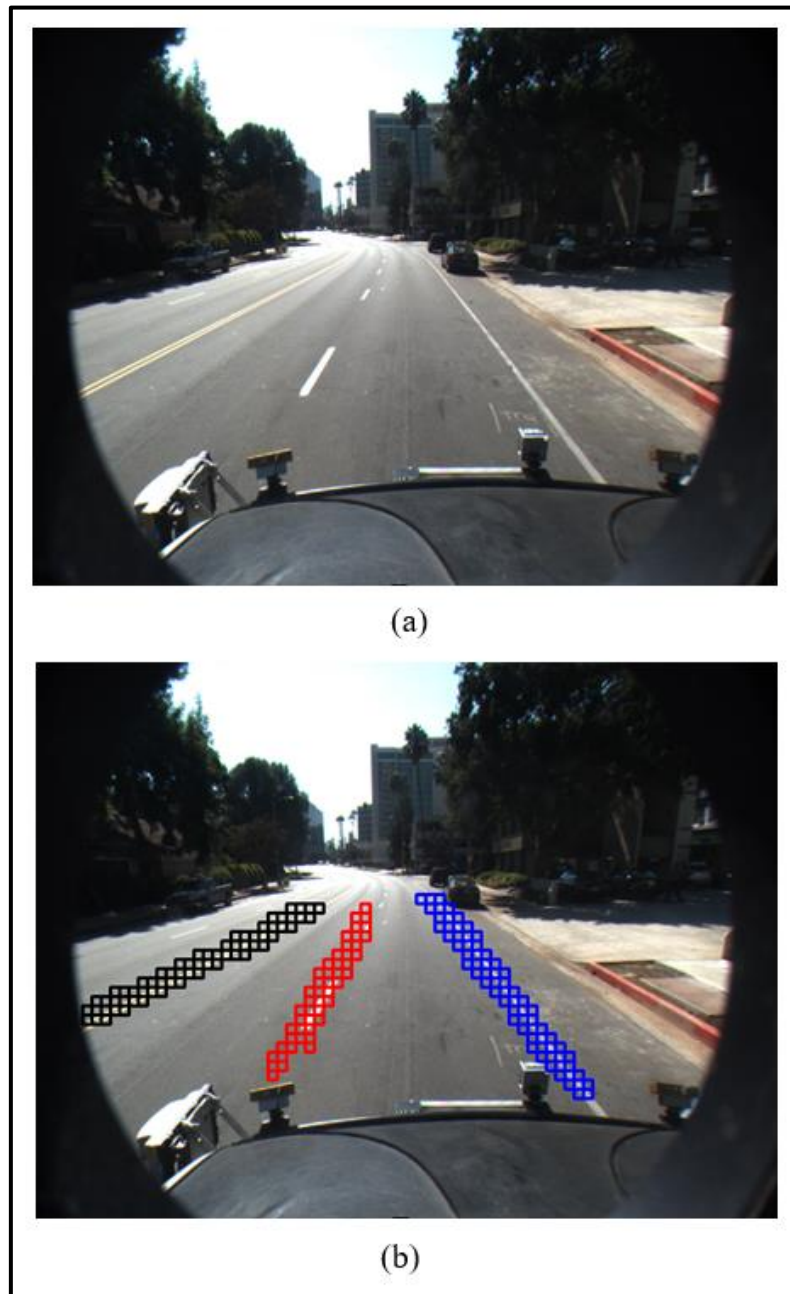


Figure 5- 1: (a) Original image, (b) Output from Model of Cordova Dataset

Table 5- 2: F1 Scores for Figure 5-1 (b)

	F1	Recall	Precision	Colour
Solid White Lanes	0.110119963	0.058268229	1.0	Blue
Broken White Lanes	0.095922845	0.050377604	1.0	Red
Yellow Lanes	0.105189142	0.055514323	1.0	Black

In Figure 5-1, the model was deployed on one of the images in the Cordova dataset to observe its lane detection accuracy with the output shown in Figure 5-1 (b). By comparing with the original image in Figure 5-1 (a), the model was able to correct identify the different lane classes in the image: solid white lanes (blue), broken white lanes (red), and yellow lanes (black).

The model trained on the Cordova dataset was able to achieve a perfect precision with its lane detection as shown in Table 5-2. However, its recall score was low, which caused its F1 score to be poor. A low recall small signifies that there were a high number of false negatives. From observation in Figure 5-1, the recall scores should be higher, since it showed a high number of true positives and a minimal number of false negatives. F1 score calculation for the Cordova dataset was similar to the VPGNet dataset, where the ground truth region was expanded and the distance between the predicted pixel and center of the ground truth pixel is calculated. If it falls within a threshold distance, the pixel would be marked as true positive. There may have been an issue in processing the Cordova dataset's ground truth. However, this issue was unable to be rectified at the end of the project.

5.3.2 VPGNet Non-Rainy Data

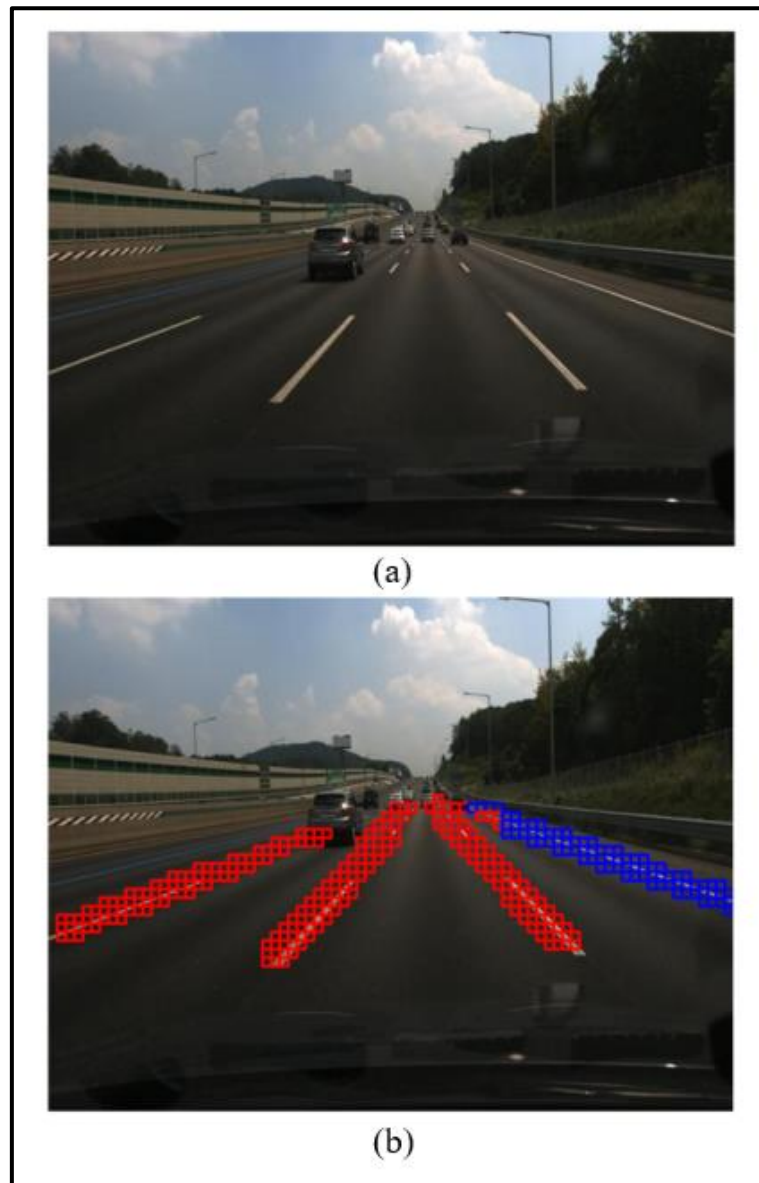


Figure 5- 2: (a) Original image, (b) Output from Model of VPGNet Non-Rainy Dataset

Table 5- 3: F1 Scores for Figure 5-2 (b)

Class Detected	F1	Recall	Precision	Detected?
lane_solid_white	0.86990783	0.93285046	0.814922225	Y (Blue)
lane_broken_white	0.85755229	0.85689870	0.858206884	Y (Red)

Figure 5-2 (b) shows the output generated from the model when deployed on a non-rainy image from the VPGNet dataset. When compared to the original image in Figure 5-2 (a), the model was able to correctly detect the lane classes and their corresponding locations. However, the model had incorrectly identified a small portion of the solid white lane to be a broken white lane.

From Table 5-3, the model was able to achieve a high recall and precision score, which resulted in a high F1 score. This meant that there were a low number of false negatives and false positives, and thus VPGNet was able to achieve a high degree of lane detection accuracy for the non-rainy data.

5.3.3 VPGNet Rainy Data

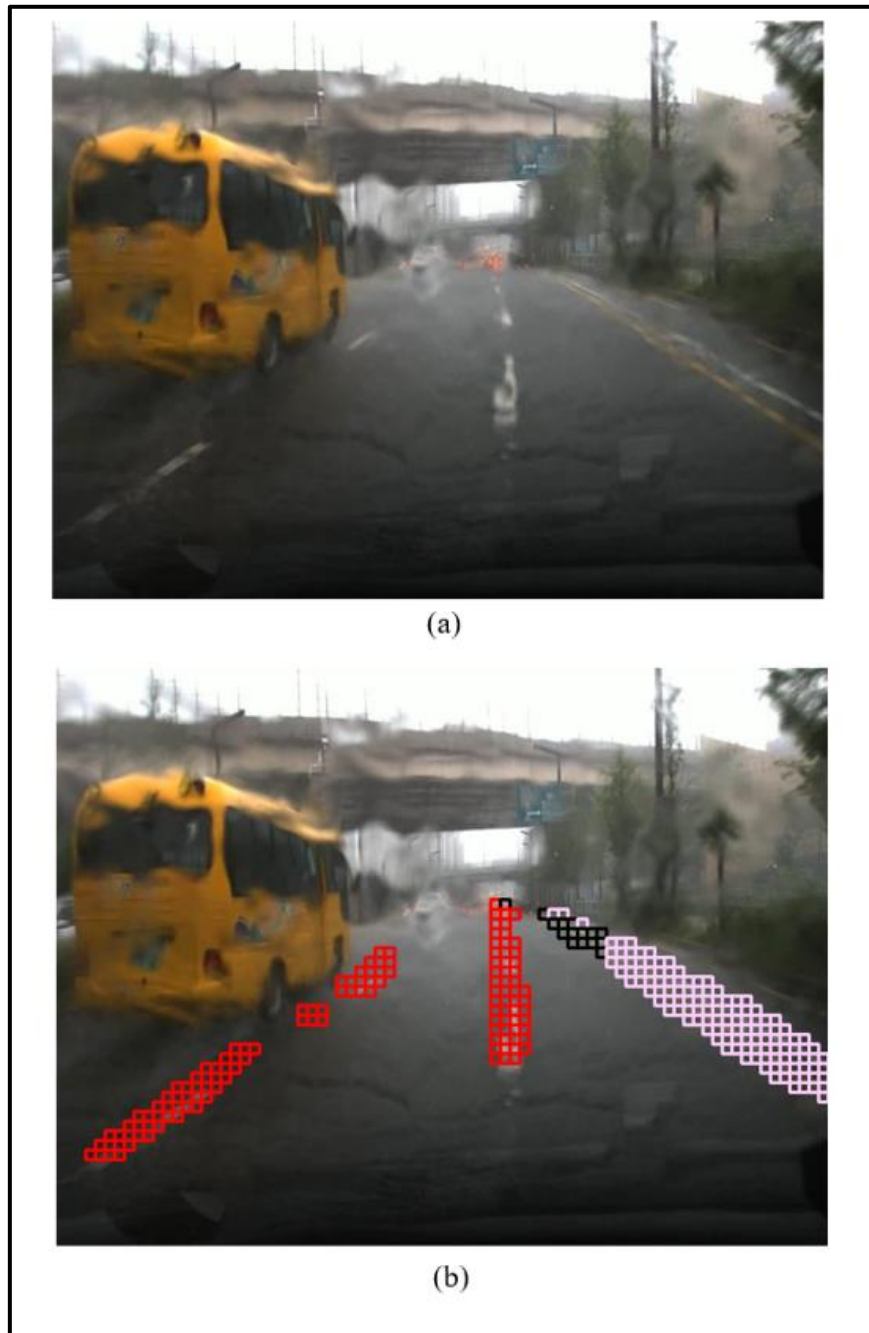


Figure 5- 3: (a) Original image, (b) Output from Model of VPGNet Rainy Dataset

Table 5- 4: F1 Scores for Figure 5-3 (b)

Class Detected	F1	Recall	Precision	Detected?
lane_broken_white	0.804483837	0.709588411	0.928678905	Y (Red)
lane_solid_yellow	0.408156091	0.262610269	0.915615142	Y (Black)
other_road_markings	0.622836245	0.584770334	0.666203060	Y (Pink)

The rainy image in Figure 5-3 (a) depicts a situation where the lane locations were distorted due to rain droplets on the vehicle's camera. Figure 5-3 (b) shows the output generated from the model when deployed on a non-rainy image from the VPGNet dataset. When compared to the original image in Figure 5-3 (a), the model was able to detect the broken white lanes and the other road markings. For this image, the combination of solid white and yellow lanes next to each other were annotated as other road markings. Although it seemed erroneous, the solid yellow lane was actually annotated in this image. The model had also incorrectly classified part of the road as a broken white lane, possibly due to the lighting conditions.

From Table 5-4, the model was able to achieve a high F1 score for the broken white lanes. However, possible due to the distortion and the way the image was annotated, the model may have been unable to differentiate between the solid yellow land and other road marking thus reducing its recall score. Overall, the model had performed well in this situation where the lanes were distorted.

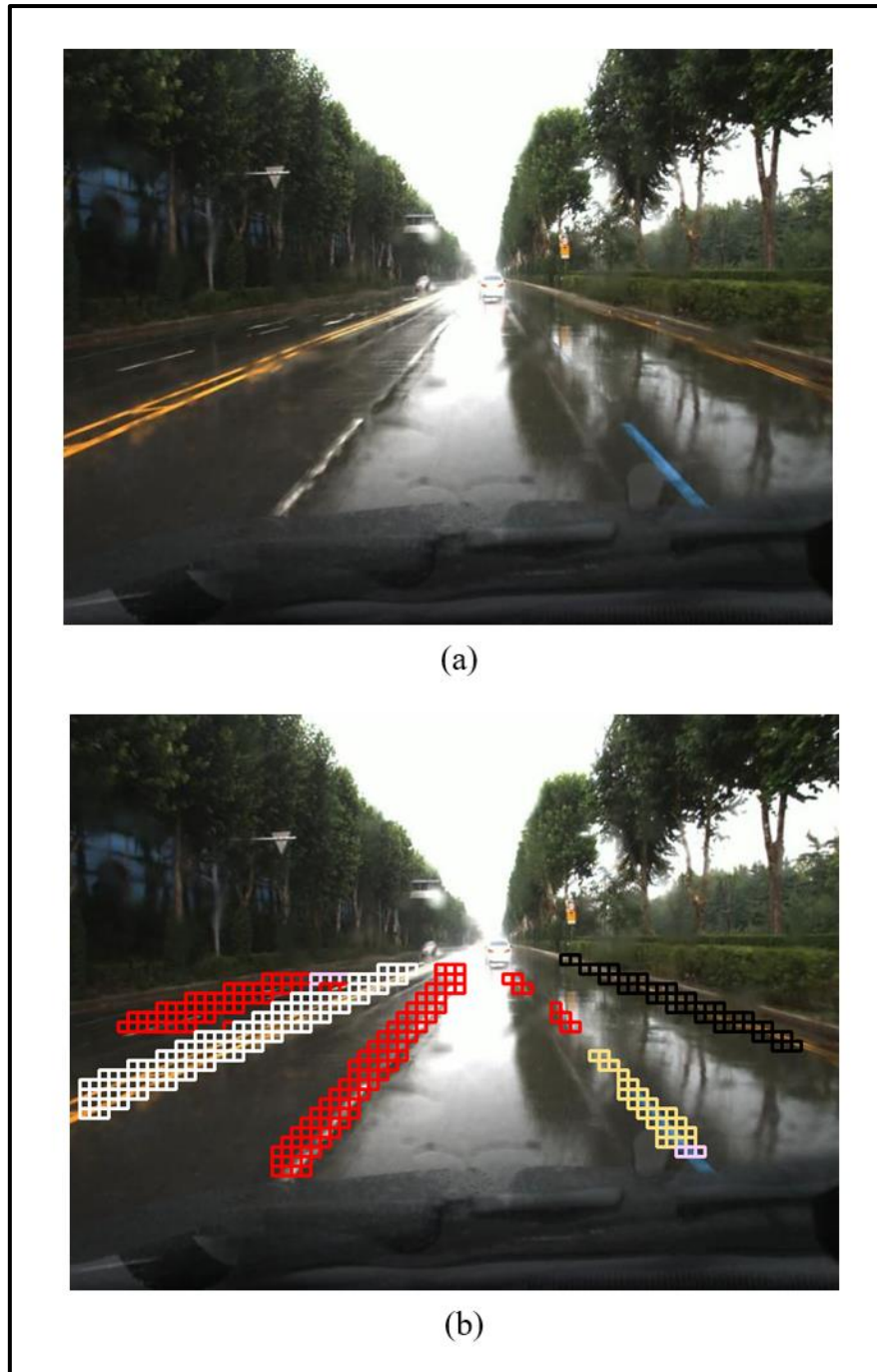


Figure 5- 4: (a) Original image, (b) Output from Model of VPGNet Rainy Dataset

Table 5- 5: F1 Scores for Figure 5-4 (b)

Class Detected	F1	Recall	Precision	Detected?
lane_broken_white	0.660694378	0.831196153	0.548235761	Y (Red)
lane_solid_yellow	0.746695863	0.614414414	0.951564680	Y (Black)
lane_double_yellow	0.586497890	0.707979626	0.500600240	Y (White)
lane_broken_blue	0.644624705	0.475606061	1.0	N (Light Blue)
safety_zone	0.339010764	0.204101723	1.0	N (Dark Green)
other_road_markings	0.267095162	0.176207303	0.551622419	Y (Pink)

Figure 5-4 (a) depicts a phenomenon that occurs during or after rainy conditions, where the wet road would cause roads to have reflections and appear bright. When deployed, the model was able to detect the broken white lanes, solid yellow lanes, and double yellow lanes without issue, as shown in Figure 5-4 (b). However, it had misclassified the broken blue lane as another road marking. This problem may have been caused by the reflection on the road surface, which caused the broken blue lane to appear as another type of road marking.

However, when compared to Table 5-5, the model had achieved a perfect precision for broken blue lanes, yet it did not appear on the output image. Furthermore, it had calculated a score for the safety zone road marking, even though there were no safety zones present in the original image. Therefore, there may be some discrepancies between the F1 score calculation and the way the output image is generated.

For broken blue lane and safety zone classes, the perfect precision score meant that the model was did not falsely identify those classes in the image. However, the low recall scores indicated that the model was not able to correctly identify several instances of those classes. This may have been due to the road surface reflection obstructing those classes from being detected.

Conversely, the double yellow lane class had a high recall score but lower precision score. This meant that there were a low number of false negatives and a considerable number of false positives. This may have also been caused by the road surface reflections, which may have caused several parts of the double yellow lane to appear as white lanes as shown by the erroneous classification.

Overall, the model had performed mostly well on a road surface with reflections.

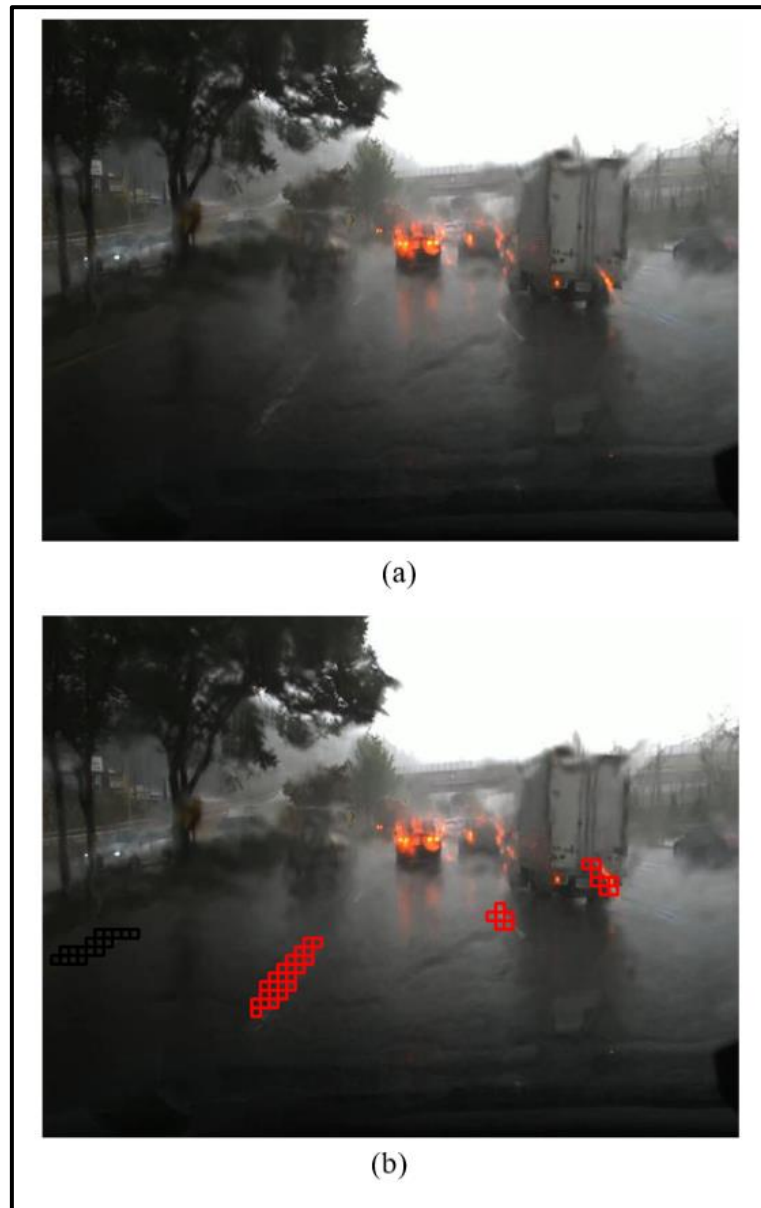


Figure 5- 5: (a) Original image, (b) Output from Model of VPGNet Rainy Dataset

Table 5- 6: F1 Scores for Figure 5-4 (b)

Class Detected	F1	Recall	Precision	Detected?
lane_broken_white	0.403303510	0.283969762	0.695631529	Y (Red)
lane_solid_yellow	0.662431319	0.676130389	0.649276338	Y (Black)

Figure 5-5 (a) depicts another scenario that arises from rainy conditions, where visibility is hampered, and lanes are barely visible. When deployed, the model was able to detect some of the broken white lanes and solid yellow lanes as shown in Figure 5-5 (b). However, it had incorrectly classified a taillight as a broken white lane. This may have been caused by a combination of rain droplets and the vehicles moving at different speeds, which causes the taillight to appear like a line.

From Table 5-6, the model had a fairly high F1 score for detecting solid yellow lanes. However, its recall score for broken white lanes was low, which resulted in a low F1 score. A low recall score could signify that there were a high number of false negatives, which meant that the model was having trouble correctly identifying the broken white lane in the image. This could be due to the broken white lanes blending into the road surface due to the road surface reflections.

Overall, the model was able to detect some lanes in this low visibility situation. However, its detection can be further improved.

5.4 Results from Pytorch Model

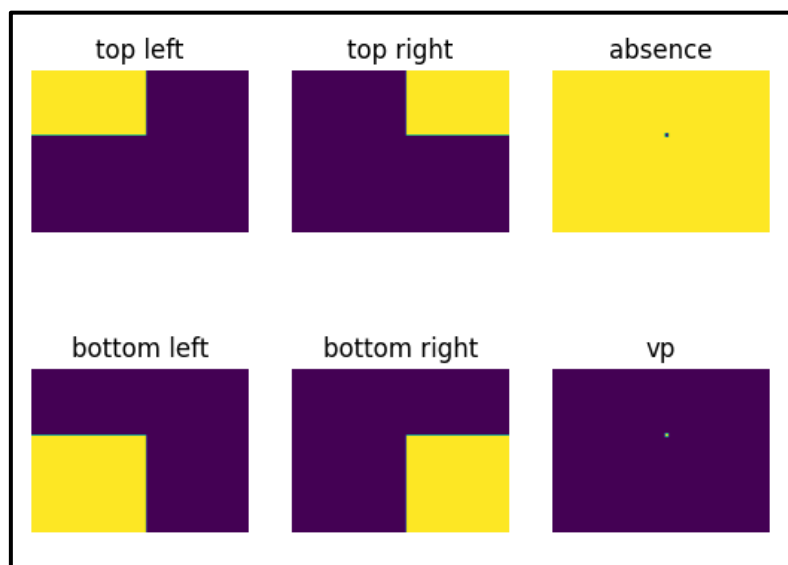


Figure 5- 6: Ground Truth VP for sample image

For the PyTorch model's VPP task, various loss functions were used to find out which was most suited for training this task. VPGNet's authors had experimented with regression losses such as L1 and L2 loss for the VP task, however they found it challenging to balance the losses from the other tasks in the network due to the difference in loss scale. For the purpose of experimentation, these regression losses would still be tested to observe the problems that VPGNet's authors had faced.

When training and testing the VPP, a learning rate of 0.000003, batch size of 2, and maximum epoch of 40 was used. A low learning rate was used because it was found to have produced the most optimal results.

Figure 5-6 shows the ground truth for the VP in one of the VPGNet datasets and the corresponding quadrants. Taking the top left corner as the origin, the VP point was located at ($x = 337$, $y = 194$).



Figure 5- 7: Output from VP task using L1 Loss

When using L1 Loss for the VPP task's training, the VP appears far from the ground truth as shown in Figure 5-7 where VP point was at ($x = 24$, $y = 10$). The model would usually predict the VP near the edge of the image. Due to the low learning rate, it had a loss value of 115.8013 at the end of 40 epochs.



Figure 5- 8: Output from VP task using L2 Loss

Figure 5-8 shows the VP predicted when L2 Loss was used to train the VPP task. Similar to the previous case, the VP predicted was far away from the ground truth and would mostly appear near the edge of the image at $(x = 471, y = 29)$. However, L2 Loss had a loss value of 172,826 at the end of 40 epochs.



Figure 5- 9: Output from VP task using BCE with Logits Loss

Since the distance-based approach loss functions did not produce satisfactory results, other loss functions based on probability were tested. Figure 5-9 illustrates the result when BCE with Logits Loss was used to train the VPP task. The VP predicted at $(x = 215, y = 366)$ was much closer to the ground truth when compared to the previous cases. Furthermore, BCE with Logits had a loss value of 69.646 by the end the training. However, this loss value seems incorrect in the context of BCE with logits, where the loss value actually represents the difference between expected and predicted probabilities.



Figure 5- 10: Output from VP task using Soft Margin Loss

Another loss function tested was Soft Margin loss, which also utilised logistic regression to find the probability classes detected. Soft Margin loss was able to predict VP locations that were close to the ground truth, as shown in Figure 5-10 where it was predicted to be at $(x = 215, y = 416)$. Furthermore, Soft Margin loss was able to achieve a very low loss value of 0.4431 at the end of training. Although Soft Margin loss was able to produce excellent results, it is not suited for training the VPP task. This is because Soft Margin loss takes in values in the range $[-1, 1]$, but due to the ReLU activation in the model, values predicted would be in the range $[0, 1]$.

5.5 Discussion of Results from Caffe and PyTorch Frameworks

The results from the Caffe model had shown to be promising with high lane detection accuracy for non-rainy data. For the rainy data, several scenarios arising from rainy conditions such as distorted lanes, reflections on the road surface, and low visibility, were used to test the model's lane detection efficacy. The model was able to correctly identify and classify lane locations to an acceptable degree of accuracy. However, an issue arose during the F1 score calculation where non-detected lanes would appear in the calculation. For the PyTorch model, much more work needs to be done on the VPP task for it to achieve an acceptable degree of accuracy. Results from using the various loss functions were shown, with BCE with Logits loss and Soft Margin producing a VP which was closest to the ground truth.

5.6 Concluding Remarks

In this chapter, the F1 score calculation method was elaborated upon and results from both Caffe and PyTorch frameworks were discussed. The Caffe model showed good efficacy in lane detection. The PyTorch model would require additional work for it to be able to detect lanes.

Chapter 6

Conclusion and Recommendations for Future Work

6.1 Conclusion

In this project, an existing lane detector algorithm, VPGNet, was implemented and evaluated for its efficacy for lane detection under both non-rainy and rainy conditions in Caffe. VPGNet was able to achieve a high degree of lane detection accuracy in non-rainy conditions. In rainy conditions, VPGNet was able to correctly identify most of the lanes. However, it had misclassified some of the lanes, possibly due to the distortions from the raindrops. A great deal of time was spent to set up and deploy VPGNet in Caffe due to Caffe's many requirements and understanding how to deploy VPGNet.

An improvement of VPGNet was also proposed under the PyTorch framework. This proposed model included modifications seen in other object detection algorithms such as dilated

convolutions and bottleneck modules, which were added in hopes of further improving the network's learning and accuracy. Due to time constraints, only the VPP task was able to be trained and various loss functions were tested to improve on the VPP task.

Current findings from the project indicates high competency for lane detection for the multi-label task in VPGNet for both non-rainy and rainy conditions. However, due to the exclusion of the VPP in VPGNet's repository, the role of the VP and other tasks in lane detection is still unclear. Results from the proposed PyTorch model have yet to reproduce accurate VP and lane detection predictions. Therefore, it is difficult to draw comparisons for both Caffe and PyTorch models based on lane detection results. This project hopes to provide better understanding on the implementation and results from the Caffe model, as well as other improvements that can be made. Once the VPP task is reproduced, it can be further evaluated and enhanced for detecting lanes in rainy conditions.

6.2 Recommendations for Future Work

The methods used in this project were able to gather some satisfactory results. However, these methods can be further improved upon and other techniques could be used to obtain greater results. The work done in this project can be used as a guide for future improvements.

6.2.1 Issues from F1 score calculation

The F1 score calculation method used in this project attempted to replicate the same technique used by VPGNet's authors. This was done to evaluate VPGNet in a Caffe and PyTorch framework.

However, as seen in Section 5.3, some of the F1 score calculations did not represent the output of the model well. Furthermore, there seemed to be some discrepancies in F1 score calculated and the classes detected by the model.

Therefore, the F1 score calculation script needs to be further examined to discover the underlying issues. Perhaps the VPGNet's author's F1 score calculation technique could be further improved upon. Other evaluation metrics, such as Mean Average Precision (mAP) could also be used to appraise the model.

6.2.2 Improving Caffe & PyTorch models

The VPGNet Caffe model was able to achieve satisfactory results when detecting lanes in rainy conditions even without the VPP task. However, detection was limited to the Cordova and VPGNet datasets. More diversified datasets could be used to train the model so that it can respond to more situations.

Unfortunately, work for the PyTorch model did not continue past developing the VPP module. The PyTorch model's architecture and other parameters can be further explored to improve on its accuracy.

Some suggestions for improving the PyTorch model include:

- (1) More Residual Blocks: Change the convolutional layers to residual blocks. Perhaps the first few layers were not learning as well, which caused inaccuracy in VPP task.
- (2) Other Loss Functions and Optimisers: Adam optimiser can be used instead of SGD to see how it can improve the training process. Other probability-based loss functions such as Kullback Leibler Divergence Loss can also be implemented.
- (3) Diversify Training Data: Other publicly available datasets with rainy lane conditions, such as BDD100K, can be used to train the model and improve its robustness.

Bibliography

- [1] Lowe, D.G., ‘*Distinctive Image Features from Scale-Invariant Keypoints*’, International Journal of Computer Vision 60, 91–110 (2004). doi: 10.1023/B:VISI.0000029664.99615.94
 - [2] H. Bay, T. Tuytelaars, L.V. Gool, “*SURF: Speeded Up Robust Features*”, Computer Vision and Image Understanding, Volume 110, Issue 3, 2008, Pages 346-359, ISSN 1077-3142.
 - [3] ujjwalkarn, ‘*An Intuitive Explanation of Convolutional Neural Networks*’, Aug 2016. Accessed on 31 Oct 2020. [Online]. Available: <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>
 - [4] Stanford University, ‘*Convolutional Neural Networks (CNNs / ConvNets)*’. Accessed on 31 Oct 2020. [Online]. Available: <https://cs231n.github.io/convolutional-networks/>
 - [5] R. Girshick, J. Donahue, T. Darrell, J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation”, 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, doi: arXiv:1311.2524v5 [cs.CV].
 - [6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection”, in University of Washington, Allen Institute for AI, Facebook AI Research, 2015
 - [7] K. He, X. Zhang, S. Ren, and J. Sun. “*Deep residual learning for image recognition.*” In Proceedings of the IEEE conference on computer vision and pattern recognition, 2016.
 - [8] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, “ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation”, arXiv, arXiv: 1606.02147, June 2016.
 - [9] M. Aly, “Real time detection of lane markers in urban streets,” 2008 IEEE Intelligent Vehicles Symposium, Eindhoven, 2008, pp. 7-12, doi: 10.1109/IVS.2008.4621152.
 - [10] Z. Wang, W. Ren, and Q. Qiu, “LaneNet: Real-Time Lane Detection Networks for Autonomous Driving,” arXiv.org, 04-Jul-2018. [Online]. Available: <https://arxiv.org/abs/1807.01726>. [Accessed: 23-Mar-2021].
-

-
- [11] S. Lee et al., “VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition,” 2017 IEEE International Conference on Computer Vision (ICCV), Venice, 2017, pp. 1965-1973, doi: 10.1109/ICCV.2017.215.
- [12] Berkeley Artificial Intelligence Research, ‘*Caffe Tutorial*’. Accessed on Oct 31 2020 [Online]. Available: <https://caffe.berkeleyvision.org/tutorial/>
- [13] A. Araujo, W. D. Norris, and J. Sim, “*Computing Receptive Fields of Convolutional Neural Networks*”, 2019. Accessed: 16 Feb 2021. [Online]. Available: <https://research.google/pubs/pub48691/>.
- [14] J. Brownlee, “*A Gentle Introduction to the Rectified Linear Unit (ReLU)*”, Aug 2020. Accessed 16 Feb 2021. [Online]. Available: <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>.
- [15] Berkeley Artificial Intelligence Research, “*Local Response Normalization (LRN)*” Accessed on 15 Mar 2021. [Online]. Available: <http://caffe.berkeleyvision.org/tutorial/layers/lrn.html>
- [16] J. Brownlee, “*A Gentle Introduction to Pooling Layers for Convolutional Neural Networks*”, Jul 2019. Accessed 16 Feb 2021. [Online]. Available: <https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/>
- [17] “*Vanishing point.*” Merriam-Webster.com Dictionary, Merriam-Webster. Accessed 17 Mar. 2021. [Online]. Available: <https://www.merriam-webster.com/dictionary/vanishing%20point>.
- [18] M. Aly, ‘*Caltech Lanes Dataset*’. Accessed on Oct 31 2020. [Online]. Available: <http://www.mohamedaly.info/datasets/caltech-lanes>
- [19] Rui Wang, “*VPGNet Usage and Lane Detection*”, Accessed on Dec 28 2020. [Github Repository]. Available: https://github.com/ArayCHN/VPGNet_for_lane
- [20] S. Lee et al., ‘*VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition*’. Accessed on Oct 31 2020. [Github Repository]. Available: <https://github.com/SeokjuLee/VPGNet>
- [21] ‘*PyTorch*’, [Github Repository]. Available: <https://github.com/pytorch/pytorch>
-

-
- [22] J. Brownlee, ‘*How to Choose Loss Functions When Training Deep Learning Neural Networks*’, Accessed on Feb 15 2021. [Online]. Available: <https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>
- [23] ‘*L1Loss*’. Accessed 22 Feb 2021. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.L1Loss.html>
- [24] ‘*MSELoss*’. Accessed 22 Feb 2021. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html>
- [25] ‘*BCEWithLogitsLoss*’. Accessed 22 Feb 2021. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html>
- [26] z_ai, ‘*Logistic Regression Explained*’, Feb 2020. Accessed 28 Feb 2021. [Online]. Available: <https://towardsdatascience.com/logistic-regression-explained-9ee73cede081>
- [27] N. Donges, ‘*Gradient Descent: An Introduction to 1 of Machine Learning’s most popular algorithms*’, June 2019. Accessed 28 Feb 2021. [Online]. Available: <https://builtin.com/data-science/gradient-descent>
- [28] S. Ruder, ‘An overview of gradient descent optimization algorithms’, 2017. Accessed 28 Feb 2021. [Online]. Available: <https://arxiv.org/abs/1609.04747>
- [29] F. Yu and V. Koltun, “Multi-Scale Context Aggregation by Dilated Convolutions,” Apr 2016. Accessed 28 Feb 2021. [Online]. Available: <https://arxiv.org/abs/1511.07122>.
- [30] W. Koehrsen, “Beyond Accuracy: Precision and Recall”, Mar 2018. Accessed on 20 Feb 2021. [Online]. Available: <https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c>
- [31] Berkeley Artificial Intelligence Research, ‘*Installation*’. Accessed on Oct 31 2020 [Online]. Available: <https://caffe.berkeleyvision.org/installation.html>
- [32] Canonical, ‘*Install Ubuntu desktop*’. Accessed on Oct 31 2020 [Online]. Available: <https://ubuntu.com/tutorials/install-ubuntu-desktop#5-prepare-to-install-ubuntu>
-

-
- [33] Ji m, ‘*How to Install The Real Ubuntu System on USB Flash Drive*’, Nov 2014. Accessed on Oct 31 2020. [Online]. Available: <http://ubuntuhandbook.org/index.php/2014/11/install-real-ubuntu-os-usb-drive/>
- [34] NVIDIA Corporation, ‘*NVIDIA CUDA Installation Guide for Linux*’, Oct 2020. Accessed on Oct 31 2020. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- [35] S. Gregory, ‘*Installing CUDA 10.1 on Ubuntu 20.04*’, June 2020. Accessed on Oct 31 2020. [Online]. Available: https://medium.com/@stephengregory_69986/installing-cuda-10-1-on-ubuntu-20-04-e562a5e724a0
- [36] Google Groups, ‘*installation error: ‘CV_LOAD_IMAGE_COLOR’ was not declared*’, Dec 2014. Accessed on Oct 20 2020. [Online]. Available: <https://groups.google.com/g/caffe-users/c/lr10Q5RCiTo?pli=1>
- [37] Stack Overflow, ‘*Resume training in Caffe from the previous training point*’, Aug 2017. Accessed on Oct 20 2020. [Online]. Available: <https://stackoverflow.com/questions/45658204/resume-training-in-caffe-from-the-previous-training-point>
- [38] Github, ‘*Could you explain more details about how to run make_lmdb.sh?*’, Dec 2017. Accessed on Oct 20 2020. [Online]. Available: <https://github.com/SeokjuLee/VPGNet/issues/6>
- [39] AskUbuntu, ‘*Graphics issues after/while installing Ubuntu 16.04/16.10 with NVIDIA graphics*’, Apr 2016. Accessed on Oct 20 2020. [Online]. Available: <https://askubuntu.com/questions/760934/graphics-issues-after-while-installing-ubuntu-16-04-16-10-with-nvidia-graphics>
- [40] D. Stutz, ‘*pyCaffe Tools, Examples and Resources*’, Dec 2016. Accessed on Oct 31 2020. [Online]. Available: <https://davidstutz.de/pycaffe-tools-examples-and-resources/>
- [41] C. Bourez, ‘*Deep learning tutorial on Caffe technology : basic commands, Python and C++ code*’, Sep 2015. Accessed on Oct 31 2020. [Online]. Available: <https://christopher5106.github.io/deep/learning/2015/09/04/Deep-learning-tutorial-on-Caffe-Technology.html>
-

- [42] Lecture Notes, ‘CS231n: Convolutional Neural Networks for Visual Recognition’, Stanford University. Accessed on Oct 31 2020. [Online]. Available: <https://cs231n.github.io/>
- [43] F. M. Carlucci, ‘Understanding caffe.io.Transformer’, Apr 2015. Accessed on Dec 15 2020. [Online]. Available: <https://groups.google.com/g/caffe-users/c/kVDppVZo3jQ>
- [44] Stack Overflow, ‘Why are RGB channels swapped in Caffe's Examples?’, Apr 2017. Accessed on Dec 15 2020. [Online]. Available: <https://stackoverflow.com/questions/43229497/why-are-rgb-channels-swapped-in-caffes-examples>

Appendix

Appendix A: Training Process of VPGNet on Caffe Framework

```

I1229 17:42:19.909015 2616 solver.cpp:242] Iteration 60300, loss = 0.97797
I1229 17:42:19.909056 2616 solver.cpp:258]   Train net output #0: bb-loss = 0.313428 (* 3 = 0.940284 loss)
I1229 17:42:19.909062 2616 solver.cpp:258]   Train net output #1: pixel-loss = 0.0170834 (* 1 = 0.0170834 loss)
I1229 17:42:19.909066 2616 solver.cpp:258]   Train net output #2: type-loss = 0.0206023 (* 1 = 0.0206023 loss)
I1229 17:42:19.909070 2616 solver.cpp:571] Iteration 60300, lr = 0.005
I1229 17:42:31.762604 2616 solver.cpp:242] Iteration 60310, loss = 1.0091
I1229 17:42:31.762650 2616 solver.cpp:258]   Train net output #0: bb-loss = 0.327106 (* 3 = 0.981317 loss)
I1229 17:42:31.762655 2616 solver.cpp:258]   Train net output #1: pixel-loss = 0.0144426 (* 1 = 0.0144426 loss)
I1229 17:42:31.762660 2616 solver.cpp:258]   Train net output #2: type-loss = 0.0133369 (* 1 = 0.0133369 loss)
I1229 17:42:31.762665 2616 solver.cpp:571] Iteration 60310, lr = 0.005
I1229 17:42:43.627400 2616 solver.cpp:242] Iteration 60320, loss = 1.00788
I1229 17:42:43.627463 2616 solver.cpp:258]   Train net output #0: bb-loss = 0.321681 (* 3 = 0.965044 loss)
I1229 17:42:43.627470 2616 solver.cpp:258]   Train net output #1: pixel-loss = 0.0222926 (* 1 = 0.0222926 loss)
I1229 17:42:43.627475 2616 solver.cpp:258]   Train net output #2: type-loss = 0.0205394 (* 1 = 0.0205394 loss)
I1229 17:42:43.627480 2616 solver.cpp:571] Iteration 60320, lr = 0.005
I1229 17:42:55.498549 2616 solver.cpp:242] Iteration 60330, loss = 1.14652
I1229 17:42:55.498596 2616 solver.cpp:258]   Train net output #0: bb-loss = 0.366548 (* 3 = 1.09965 loss)
I1229 17:42:55.498602 2616 solver.cpp:258]   Train net output #1: pixel-loss = 0.0222089 (* 1 = 0.0222089 loss)
I1229 17:42:55.498607 2616 solver.cpp:258]   Train net output #2: type-loss = 0.0246623 (* 1 = 0.0246623 loss)
I1229 17:42:55.498612 2616 solver.cpp:571] Iteration 60330, lr = 0.005
I1229 17:43:07.366305 2616 solver.cpp:242] Iteration 60340, loss = 1.05087
I1229 17:43:07.366349 2616 solver.cpp:258]   Train net output #0: bb-loss = 0.340181 (* 3 = 1.02054 loss)
I1229 17:43:07.366356 2616 solver.cpp:258]   Train net output #1: pixel-loss = 0.0149167 (* 1 = 0.0149167 loss)
I1229 17:43:07.366360 2616 solver.cpp:258]   Train net output #2: type-loss = 0.0154113 (* 1 = 0.0154113 loss)
I1229 17:43:07.366365 2616 solver.cpp:571] Iteration 60340, lr = 0.005
I1229 17:43:19.227862 2616 solver.cpp:242] Iteration 60350, loss = 1.04397
I1229 17:43:19.227932 2616 solver.cpp:258]   Train net output #0: bb-loss = 0.332457 (* 3 = 0.997372 loss)
I1229 17:43:19.227938 2616 solver.cpp:258]   Train net output #1: pixel-loss = 0.0238541 (* 1 = 0.0238541 loss)
I1229 17:43:19.227942 2616 solver.cpp:258]   Train net output #2: type-loss = 0.0227404 (* 1 = 0.0227404 loss)
I1229 17:43:19.227947 2616 solver.cpp:571] Iteration 60350, lr = 0.005

```

Figure A-1: Loss values during training

Appendix B: Deployment Code for VPGNet on Caffe Framework

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score, recall_score, precision_score
from scipy.io import loadmat
from scipy import ndimage
import sys
import os

caffe_root = '/home/ubuntu/fyp/VPGNet-master/caffe'
sys.path.insert(0, os.path.join(caffe_root, 'python'))
import caffe
import cv2

class LaneDetector:

    def __init__(self, workspace_root='.'):
        if not os.path.exists(os.path.join(os.getcwd(), workspace_root)):
            os.mkdir(workspace_root)

        self.model = './deploy.prototxt'

        ## Trained Model Location
        self.pretrained = 'snapshots/split_iter_100000.caffemodel'

        caffe.set_mode_cpu()
        self.net = caffe.Net(self.model, self.pretrained, caffe.TEST)
        print ("successfully loaded classifier")

    def load_image(self, filename):
        self.filename = filename
        self.img = caffe.io.load_image(filename)
        print self.img.shape
        transformer = caffe.io.Transformer({'data': self.net.blobs['data'].data.shape})
        transformer.set_transpose('data', (2, 0, 1)) # move image channels to outermost dimension
        transformer.set_raw_scale('data', 255) # rescale from [0, 1] to [0, 255]
        transformer.set_channel_swap('data', (2, 1, 0)) # RGB -> BGR
        self.transformed_img = transformer.preprocess('data', self.img)
        print self.transformed_img.shape
        self.net.blobs['data'].data[...] = self.transformed_img

    def forward(self):
        # Forward Propagation
        self.net.forward()

    def visualize(self, num):
        # Visualise Classification
        original_img = cv2.imread(self.filename)
        original_img = cv2.resize(original_img, (640, 480))
        classification = self.net.blobs['multi-label'].data
        # classes = []
        y_offset_class = 1 # offset for classification error
        x_offset_class = 1
        grid_size = self.img.shape[0]/60
```

```

# create color for visualizing classification
def color_options(x):
    return {
        1: (255, 0, 0),          # blue          # lane_solid_white
        2: (0, 0, 255),          # red           # lane_broken_white
        3: (255, 153, 0),         # orange         # lane_double_white
        4: (0, 0, 0),            # black          # lane_solid_yellow
        5: (153, 153, 153),       # grey           # lane_broken_yellow
        6: (255, 255, 255),       # white          # lane_double_yellow
        7: (128, 223, 255),       # light blue     # lane_broken_blue
        8: (128, 64, 0),          # brown          # lane_slow
        9: (230, 230, 0),         # yellow         # stop_line
        10: (230, 230, 0),        #               # arrow_left
        11: (230, 230, 0),        #               # arrow_right
        12: (230, 230, 0),        #               # arrow_go_straight
        13: (230, 230, 0),        #               # arrow_u_turn
        14: (51, 204, 51),        # green          # speed_bump
        15: (198, 255, 179),      # light green     # crossWalk
        16: (25, 103, 25),        # dark green      # safety_zone
        17: (255, 204, 255),      # pink           # other_road_markings
        18: (255, 0, 255)         # magenta         # NIL
    }[x]

for i in range(60):
    # classes.append([])
    for j in range(80):
        max_value = 0
        maxi = 0
        # Finding max value
        for k in range(64):
            if classification[0, k, i, j] > max_value:
                max_value = classification[0, k, i, j]
                maxi = k
        # classes[i].append(maxi)
        if maxi != 0:
            pt1 = ((j + y_offset_class)*grid_size, (i+x_offset_class)*grid_size)
            pt2 = ((j + y_offset_class)*grid_size+grid_size, (i+x_offset_class)*grid_size+grid_size)
            cv2.rectangle(original_img, pt1, pt2, color_options(maxi-1), 2)

cv2.imwrite('VPG_log/labelled/%d_labeled.png'%num, original_img)

def create_circular_mask(self, h, w, center=None, radius=None):
    # Create a circular mask from center on an (h,w) map with euclidean distance radius
    if center is None: # use the middle of the image
        center = (int(h/2), int(w/2))
    if radius is None: # use the smallest distance between the center and image walls
        radius = min(center[0], center[1], h-center[0], w-center[1])

    X, Y = np.ogrid[:h, :w]
    dist_from_center = np.sqrt((X - center[0])**2 + (Y-center[1])**2)

    mask = dist_from_center <= radius
    return mask

```

```

def vpg_lane_f1(self, map1, map2, mask_R = 12):
    # For single class metric, calculate two map1,2 whose label are binary (0 for no, and !0 for label)
    # Input:
    #     map1: (h,w,1) size label predicted map
    #     map2: (h,w,1) size label groundtruth map, every 8*8 pixels an grid with same label
    #     mask_R: euclidean distance of radius R, default 4
    # Return:
    #     single_class_f1: f1 score for map1 and map2 described in VPGNet Sec5.3

    # First extend the grid-labeled map2 to circle-labeled map extend_mask with boundary R

    map1_mask = map1 > 0
    map2_mask = map2 > 0 # Assume map1,2 only have one class
    extend_mask = np.zeros((480, 640), dtype=bool) # extended groundtruth (from 8*8 square grid to radius R circle)
    for i in range(0, 480):
        for j in range(0, 640):
            if map2_mask[i,j] == True: # if this pixel have label, this 8*8 grid should have same label
                area_mask = detector.create_circular_mask(480, 640, center = (i,j), radius = mask_R)
                extend_mask = extend_mask + area_mask # add the area_mask to blank mask

    # Compare map1 and the extended mask for f1 score
    single_class_f1 = f1_score(extend_mask.flatten(), map1_mask.flatten())
    recall_dict = recall_score(extend_mask.flatten(), map1_mask.flatten())
    precision_dict = precision_score(extend_mask.flatten(), map1_mask.flatten())
    return single_class_f1, recall_dict, precision_dict

def calcSingleImage(self, image_path, gt_path, sensivity, mask_R):
    #==== Load Ground Truth ====
    mat = loadmat(gt_path) # load mat file
    rgb_seg_vp_label = mat['rgb_seg_vp']
    gt = rgb_seg_vp_label[:, :, 3].reshape(480,640)

    mlabel = self.net.blobs['multi-label'].data # mlabel: saves 18 feature maps for different classes
    mlabel = np.delete(mlabel, 1, 1) # delete the second channel in prediction, which I don't know what is it.
    x = 10 # vertical shifting
    y = 12 # horizontal shifting
    f1_dict = {}
    recall_dict = {}
    precision_dict = {}

    def class_names(class_num):
        return {
            1: 'lane_solid_white',
            2: 'lane_broken_white',
            3: 'lane_double_white',
            4: 'lane_solid_yellow',
            5: 'lane_broken_yellow',
            6: 'lane_double_yellow',
            7: 'lane_broken_blue',
            8: 'lane_slow',
            9: 'stop_line',
            10: 'arrow_left',
            11: 'arrow_right',
            12: 'arrow_go_straight',
            13: 'arrow_u_turn',
            14: 'speed_bump',
            15: 'crossWalk',
            16: 'safety_zone',
            17: 'other_road_markings',
            18: 'unknown'
        }[class_num]

```

```

for i in range(1, 18):
    small_mask = mlabel[0, i, ...] * 255 # normalize from (0,1) to (0,255)
    resized_mask = cv2.resize(small_mask, (640, 480))
    translationM = np.float32([[1, 0, x], [0, 1, y]])
    resized_mask = cv2.warpAffine(resized_mask, translationM, (640, 480))
    imggray = resized_mask.astype(np.uint8)
    ret, thresh = cv2.threshold(imggray, sensitivity, 1, cv2.THRESH_BINARY)
    thresh = thresh.reshape(480, 640).astype(np.uint8)
    predn = thresh.astype(np.bool)
    gtn = (gt == i)
    class_type = class_names(i)
    f1_dict[class_type], recall_dict[class_type], precision_dict[class_type] = detector.vpg_lane_f1(predn, gtn, mask_R = mask_R)
return f1_dict, recall_dict, precision_dict

workspace_root = 'VPG_log/'
detector = LaneDetector(workspace_root)

## Deploy on data
vpg_img = '000300'
gt_path = '/home/ubuntu/fyp/VPGNet-master/20160809_1428_34_distort/' + vpg_img + '.mat'
image_path = '/home/ubuntu/fyp/VPGNet-master/20160809_1428_34_distort/' + vpg_img + '.png'

detector.load_image(image_path)
detector.forward()
detector.visualize(int(vpg_img))

f1_score, recall_score, precision_score = detector.calcSingleImage(image_path, gt_path, 50, 13)

print("F1 score: ", f1_score)
print("Recall Score: ", recall_score)
print("Precision Score: ", precision_score)

```


Appendix C: Training Code for VPGNet on PyTorch Framework

```
# -*- coding: utf-8 -*-
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import torchvision.transforms as tf
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

from model.VPGNetV2 import *
from model.data_loader import *
from model.metrics import *

def init_optim_losses(task, **kwargs):
    # reset all param require_grad to True
    for param in net.parameters():
        param.requires_grad = True

    # based on task, select params for training and instantiate optim and loss
    if task=='vp': # only shared and vp layers trained
        for name, param in net.named_parameters():
            if 'ml' in name or 'om' in name:
                param.requires_grad = False
        optim_fn = torch.optim.Adam(net.parameters(),
                                     lr=lr_vp,
                                     betas=(0.5,0.999))
        loss_fn = nn.BCEWithLogitsLoss()
    else:
        raise ValueError(f'Task {task} is not recognized, only accept "vp", "objectmask", or "multilabel"!')
    return optim_fn, loss_fn

def train_task_epoch(**kwargs):
    for i, batch in enumerate(trainloader):
        # Train iteration
        img, gt = batch['image'].to(device), batch[task].to(device)

        # Forward pass
        pred = net(img)[task]
        ch = pred.shape[1]
        print(f'\nraw values:\nmax: {pred.view(batch_size,ch,-1).max(dim=2)[0]}\nmin: {pred.view(batch_size,ch,-1).min(dim=2)[0]}')

        # Compute loss
        loss = loss_fn(pred, gt[:, :, :, :])

        # Zero optimizer
        optim_fn.zero_grad()

        # Backprop
        loss.backward()

        # Optimize weights
        optim_fn.step()
        if i%10==0:
            print(f'Epoch: [{ep}/{max_epoch-1}] Iter: [{i}/{len(trainloader)-1}] {task} loss: [{loss:.4f}] ')

    # Save weights every x epochs
    if (ep)%save_freq==0:
        print(f'\nSaving weights for task: {task} epoch: {ep}')
        torch.save(net.state_dict(), f'checkpoints/Epoch_{task}_{ep}.pth')
```

```

def val_task_epoch(**kwargs):
    if task=='vp':
        # Initialize accumulator for scores
        ep_f1, ep_recall, ep_precision = 0, 0, 0

    else:
        # Initialize score dict
        ep_f1_dict = {i+1: 0 for i in classes}
        ep_recall_dict = {i+1: 0 for i in classes}
        ep_precision_dict = {i+1: 0 for i in classes}

    # loop across valset
    for il, batch in enumerate(tqdm(valloader)):
        img, gt = batch['image'].to(device), batch[task].to(device)
        img.to(device)

        # Forward pass
        pred = net(img)[task]

        # Compute VP point
        if task=='vp':
            pred_4sum = pred.sum(dim=1) # sum along 4 quadrants
            pred_max = pred_4sum.view(batch_size,-1).max(dim=1)[1]
            vp_x = pred_max//im_size[1]
            vp_y = pred_max%im_size[1]
            print("vp_x = ", vp_x)
            print("vp_y = ", vp_y)

            pred = torch.zeros(pred_4sum.shape)
            pred_viz = torch.zeros(pred_4sum.shape)
            for j in range(batch_size):
                pred[j, vp_x[j], vp_y[j]] = 1
                pred_viz[j, vp_x[j]-5:vp_x[j]+5, vp_y[j]-5:vp_y[j]+5] = 1

            gt = 1 - gt[:,0,:,:] # 1-absence channel = vp channel

        # display samples
        if (il%20)==0:
            f, axes = plt.subplots(1, 2)
            j1 = np.random.randint(batch_size)
            axes[0].imshow(img[j1,:,:,:].permute(1,2,0).cpu().numpy().astype(np.uint8))
            axes[0].set_title(f'img')
            axes[0].axis('off')
            axes[1].imshow(pred_viz[j1,:,:,:].cpu().numpy())
            axes[1].set_title(f'vp')
            axes[1].axis('off')
            plt.savefig(f'samples/{task}_{ep}_{batch}_val_sample.png')

        if pred.view(batch_size,-1).sum(dim=1).sum().item() < batch_size:
            print(f'num preds: {pred.view(batch_size,-1).sum(dim=1)}')

```

```

if __name__ == '__main__':
    # Image and annotation paths
    train_path = './data/mat'
    val_path = './data/mat'
    classlist_path = './data/vpgnet_classlist.txt'

    # Train parameters
    start_epoch = 0 # start epoch index
    max_epoch = 40 # max num of epochs for training in each task
    save_freq = 20 # freq of saving weights (epoch)
    eval_freq = 1 # freq of evaluation (epoch)
    batch_size = 2
    lr_vp = 0.000003
    lr_om = 0.0005
    lr_ml = 0.0005
    im_size = (480, 640) # row x col
    tasks = ['vp', 'objectmask', 'multilabel']
    # tasks = ['vp']

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(device)

    # Instantiate dataset and dataloader
    tfm = [tf.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))]

    trainloader = DataLoader(VPG_dataloader(dataPath=train_path,
                                           classList=classlist_path,
                                           mode='train',
                                           _transforms=tfm),
                             batch_size=batch_size,
                             shuffle=True,
                             #num_workers=8,
                             drop_last=True)

    valloader = DataLoader(VPG_dataloader(dataPath=val_path,
                                           classList=classlist_path,
                                           mode='val',
                                           _transforms=tfm),
                            batch_size=batch_size,
                            shuffle=False,
                            #num_workers=8,
                            drop_last=True)

    # Extract classes
    classes, numclass = trainloader.dataset.classlist, trainloader.dataset.numclass

    # Instantiate model
    net = VPGNet_v2(numclass=numclass).to(device)

    # init model weights
    init_weights(net, tasks[0], start_epoch)

```

```
### Train loop ###
for task in tasks:
    print(f'Training "{task}" task')

    # Initialize score dict for objectmask and multilabel tasks only
    if task!='vp':
        f1_dict = {i+1: 0 for i in classes}
        recall_dict = {i+1: 0 for i in classes}
        precision_dict = {i+1: 0 for i in classes}

    # best f1 score -> trigger save best weights
    best_avg_f1 = 0

    # Initialize optim fn and loss fn
    optim_fn, loss_fn = init_optim_losses(task)

    # print(f'loss fn: {loss_fn}')
    # Train task
    for ep in range(start_epoch, max_epoch):
        # Train epoch
        train_task_epoch()

        # Validate every x epoch
        if (ep)%eval_freq==0:
            # print(f'Validating epoch {ep}')
            avg_f1 = val_task_epoch()

            # Save best weights if val results are best
            print(f'Avg validation f1 score for epoch {ep}: {avg_f1}')
            if avg_f1 > best_avg_f1:
                best_avg_f1 = avg_f1
                torch.save(net.state_dict(), f'checkpoints/best_{task}_epoch.pth')

    print(f'Completed Training {task} branch')
```