LANE AND TRAFFIC SIGN DETECTION IN SELF-DRIVING CARS USING DEEP LEARNING

A PROJECT REPORT

submitted by

ANEESA BANU.K
DHIVYA.S
MUKESH KARTHIKEYAN.S

in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

EASWARI ENGINEERING COLLEGE

(Autonomous Institution)

affiliated to

ANNA UNIVERSITY::CHENNAI – 600025

JUNE 2022

EASWARI ENGINEERING COLLEGE, CHENNAI

(AUTONOMOUS INSTITUTION)

AFFILIATED TO ANNA UNIVERSITY, CHENNAI – 600025

BONAFIDE CERTIFICATE

Certified that this project report **"LANE AND TRAFFIC SIGN DETECTION IN SELF-DRIVING CARS USING DEEP LEARNING"** is the bonafide work of "**ANEESA BANU.K (310618104011), DHIVYA.S (310618104027), MUKESH KARTHIKEYAN.S (310618104061)"** who carried out the project under my supervision.

SIGNATURE	**SIGNATURE**
Dr.G.S.ANANDHA MALA	**Mrs. B.PADMAVATHI**
HEAD OF THE DEPARTMENT	**SUPERVISOR**
	Assistant Professor
Department of Computer Science and Engineering	Department of Computer Science and Engineering
Easwari Engineering College	Easwari Engineering College
Ramapuram, Chennai - 600089	Ramapuram, Chennai - 600089

CERTIFICATE OF EVALUATION

College Name : Easwari Engineering College

Branch & Semester : Computer Science and Engineering & VIII

<table>
<tr><th>S.NO</th><th>NAME OF THE STUDENT</th><th>TITLE OF THE PROJECT</th><th>NAME OF THE SUPERVISOR WITH DESIGNATION</th></tr>
<tr><td>1</td><td>ANEESA BANU.K (310618104011)</td><td rowspan="3">LANE AND TRAFFIC SIGN DETECTION IN SELF-DRIVING CARS USING DEEP LEARNING</td><td rowspan="3">Mrs.B.PADMAVATHI
ASSISTANT PROFESSOR</td></tr>
<tr><td>2</td><td>DHIVYA.S (310618104027)</td></tr>
<tr><td>3</td><td>MUKESH KARTHIKEYAN.S (310618104061)</td></tr>
</table>

The report of the project work submitted by the above students in the partial fulfillment for the award of the degree of Bachelor of Engineering in Computer Science and Engineering, Anna University were evaluated and confirmed to be a report of the work done by the above students.

The viva voce examination of the project work was held on ___________________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We hereby place our deep sense of gratitude to our beloved founder, Chairman of the institution, **Dr.T.R.Pachamuthu,B.Sc.,M.I.E.,** for providing us with the requisite infrastructure throughout the course.

We would also like to express our gratitude towards our Chairman **Dr.R.Shivakumar,M.D., Ph.D.,** for giving the necessary facilities.

We convey our sincere thanks to **Dr.R.S.Kumar,M.Tech.,Ph.D.,** Principal Easwari Engineering College, for his encouragement and support. We extend our hearty thanks to **Dr.V.Elango,M.E.,Ph.D.,**Vice Principal (academics) and **Dr.S.Nagarajan,M.E,Ph.D.,** Vice Principal (admin), Easwari Engineering College for their constant encouragement.

We take the privilege to extend our hearty thanks to **Dr.G.S.Anandha Mala,M.S.,M.E.,Ph.D.,** Head of the Department, Computer Science and Engineering, Easwari Engineering College for her suggestions, support and encouragement towards the completion of the project with perfection.

We would also like to express our gratitude to our Project Coordinator, **Mrs.D.Kavitha,M.Tech.,(Ph.D.),** Assistant Professor, Department of Computer Science and Engineering, Easwari Engineering College for her constant support and encouragement.

We would also like to express our gratitude to our guide **Mrs.B.Padmavathi,M.E.,(Ph.D.),** Assistant Professor, Department of Computer Science and Engineering, Easwari Engineering College for her constant support and encouragement.

Finally, we wholeheartedly thank all the faculty members of the Department of Computer Science and Engineering for warm cooperation and encouragement.

ABSTRACT

With artificial intelligence technology progressing at a tremendous speed, intelligent driving has gotten a lot of recognition in recent years. Lane detection is one of the primary functions in self-driving cars. Traditionally, lane detection was done using image processing algorithms and computer vision techniques, which included extraction of areas which are possible lane areas, edge enhancement etc. Deep learning models with new improvements are being introduced till date. A self-driving car must also be able to identify traffic signs. In the proposed work a VGG-16 convolutional neural network is used for road segmentation. The model is trained on the KITTI Road/Lane Detection Evaluation 2013 dataset. The model performed well with an accuracy of 98.58 %. For traffic sign detection, the German Traffic Sign Recognition Benchmark dataset is used. A convolutional neural network is used with ADAM optimizer, which gives an accuracy of 95%.

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	**i**
	LIST OF TABLES	**v**
	LIST OF FIGURES	**vi**
1	**INTRODUCTION**	**1**
	1.1. General	1
	1.2 Problem Description	1
	1.3 Objective	2
	1.4 Scope of Project	2
	1.5 Organization of the Project	2
2	**LITERATURE SURVEY**	**3**
	2.1. General	3
	2.2. Existing Systems	3
	2.3. Issues in Existing Systems	7
	2.4. Proposed System	7
	2.5. Summary	8

3	**SYSTEM DESIGN**	**9**
	3.1 General	9
	3.2. System Architecture	9
	3.3. Functional Architecture	11
	3.4. Modular Design	12
	3.4.1 Lane Detection	13
	3.4.1.1 Data Pre-Processing	13
	3.4.1.2 VGG-16 Convolutional Neural Network	13
	3.4.2 Traffic Sign Detection	15
	3.4.2.1 Data Pre-Processing	15
	3.4.2.2 Convolutional Neural Network	15
	3.5. System Requirements	16
	3.6. Summary	17
4	**SYSTEM IMPLEMENTATION**	**18**
	4.1. General	18
	4.2. Overview of the Platform	18

	4.3. Module Implementation	23
	4.4. Summary	32
5	**SYSTEM TESTING AND PERFORMANCE ANALYSIS**	**33**
	5.1. General	33
	5.2. Test Cases	34
	5.3. Performance Measures	34
	5.4. Performance Analysis	35
	5.4.1 Lane Detection in Self-Driving Cars	35
	5.4.2 Traffic Sign Detection in Self-Driving Cars	36
	5.5. Summary	39
6	**CONCLUSION AND FUTURE WORKS**	**40**
	APPENDICES	**41**
	REFERENCES	**62**

LIST OF TABLES

TABLE NO.	TITLE	PAGE NO.
5.4.2.1	Performance Evaluation of CNN Model for Traffic Sign Detection	37

LIST OF FIGURES

FIGURE NO.	TITLE	PAGE NO.
3.2.1	System Architecture for Lane Detection	10
3.2.2	System Architecture for Traffic Sign Detection	10
3.3.1	Functional Architecture for Lane Detection	11
3.3.2	Functional Architecture for Traffic Sign Recognition	12
3.4.1	VGG 16 CNN Architecture for Lane Detection	14
3.4.2	CNN Architecture for Traffic Sign Detection	16
4.3.1.1	Importing the Required Libraries	23
4.3.1.2	Loading the Directories	23
4.3.1.3	Generating Training Examples and Initializing Constants	24
4.3.1.4	Generating Dataset Variables	24
4.3.1.5	Plotting the Model Summary	25
4.3.1.6	Defining the VGG Network	26
4.3.1.7	Training the Model for 200 Epochs	26
4.3.1.8	Making Predictions on the Test Data Set	27
4.3.2.1	Importing the Required Libraries	27
4.3.2.2	Loading the Directories	28
4.3.2.3	Generating the Dataset Plot	28

4.3.2.4	Initializing the Parameters	29
4.3.2.5	Plotting the Model Summary	29
4.3.2.6	Defining the Lenet Architecture	30
4.3.2.7	Training the Model for 200 Epochs	31
4.3.2.8	Making Predictions on the Test Data Set	32
5.4.1.1	Sample Prediction after 200 Epochs	35
5.4.1.2	Sample Predictions for Lane Segmentation	36
5.4.2.1	Performance Plots for Cnn for Traffic Sign Detection	38
5.4.2.2	Test Images and their Predicted Classes	38
5.4.2.3	Confusion Matrix for the Cnn Model for Traffic Sign Detection	39

CHAPTER 1

INTRODUCTION

1.1 GENERAL

In 2017 alone, over 40,000 people died in the United States due to car accidents. Across the globe, the number increases to more than a million people. Most of the accidents could have been avoided if the drivers had paid attention to their surroundings. A number of automobile brands and autonomous vehicle companies are investing billions in self-driving technology.

Recently, the amount of research in the field of self-driving cars has grown significantly with autonomous vehicles having clocked in more than 10 million miles, providing a substantial amount of data for use in training and testing.

1.2 PROBLEM DESCRIPTION

1. Lane detection is the problem of locating lane boundaries without prior knowledge of the road geometry. Most lane detection methods are edge-based.
2. After an edge detection step, the edge-based methods organize the detected edges into meaningful structure (lane markings) or fit a lane model to the detected edges. Most of the edge-based methods, in turn, use straight lines to model the lane boundaries.
3. One challenge for accurate lane detection is to deal with noise appearing in the input image, such as object shadows, brake marks, breaking lane lines.
4. In some models the dataset was not balanced and hence some classes had more images pertaining to them than the rest.

1.3 OBJECTIVES

The major objective of this project is to provide the following

- To assist keeping a car in a particular lane in self-driving cars.
- To detect lane lines in images using a fully connected CNN(Convolution Neural Network).
- To understand the properties of road and traffic signs and their implications for image processing for the recognition task.
- To identify the most appropriate approach for feature extraction from road signs.
- To develop an appropriate road sign classification algorithm
- To develop robust algorithms that can be used in a wide range of conditions.

1.4 SCOPE OF THE PROJECT

The scope of the project is to create a model for detecting road lanes and recognizing traffic signs in self-driving cars using deep learning algorithms.

1.5 ORGANIZATION OF THE PROJECT

The report consists of 6 chapters, the contents of which are described below: Chapter 1 is the introduction that explains the basic information of the system. Chapter 2 is the literature survey that elaborates on the research works on the existing systems. Chapter 3 describes the system design. Chapter 4 gives details regarding the system implementation. Chapter 5 describes the performance analysis of the proposed system. Chapter 6 provides the conclusion, which summarizes the efforts undertaken in the proposed system and states findings and shortcomings in the proposed system.

CHAPTER 2

LITERATURE SURVEY

2.1 GENERAL

The documentation of a comprehensive review of the publishers and unpublished work from secondary sources data in the areas of specific interest to the researcher is referred to as a literature survey. It is a survey of all the techniques that have been used to summarize and analyze user reviews thus far. There have been major publications focusing on identifying diseases using deep learning techniques, but the number of research papers focused on large-scale application is limited. Any of the scientific papers and studies mentioned in this section were used to improve our proposed method. The fundamental properties and shortcomings of existing devices are discussed.

2.2 EXISTING SYSTEM

The contributions of various scholars are studied for survey and analyzing the merits and demerits in order to enhance the consequences for making the system work better.

Lane Detection:

Kanagaraj, et al., 2021 [7] demonstrated a deep learning strategy for self-driving autonomous cars lane recognition using a Convolution Neural Network (CNN) and a Spatial Transformer Network (STN) with a calibration matrix and distortion coefficients. During the testing phase, the neural network's depth assisted the vehicles in making decisions based on the training data. In addition to lane detection, a model for detecting traffic signs was built. A traffic signs detection model was also developed in addition to lane detection. The Adam Optimizer, which operates on top of the LeNet-5 architecture, is used in the suggested method. German Traffic Sign dataset is used to train the model. The LeNet-5 design was determined to be 97%.

Zhang et al., 2021 [17] proposes an effective lane line detection method called Ripple-GAN. They initially suggest RiLLD-Net, a simpler and more fundamental Ripple-GAN network structure, which can learn features quickly. They developed Ripple-GAN by combining RiLLD-Net with the notion of WGAN to deal with difficult circumstances including complicated, incomplete, or occluded lane lines. The generator in RippleGAN is a multi-target segmentation network, and Gaussian noise is introduced to the network's input, giving Ripple-GAN the capacity to handle detection tasks under difficult road conditions. On the TuSimple dataset, the suggested Ripple-GAN achieved satisfactory results, and its F1 score is greater than that of previous approaches. When the road surface is entirely or partially covered, such as a street surface with dim lights, detecting lane lines becomes more difficult which is the direction of their future work.

Marzougui et al., 2020 [12] presents a real-time lane marking and detection system based on computer vision. Smoothing and edge detection operators are used to preprocess the dataset. The region of interest is marked. The Progressive Probabilistic Hough Transform (PPHT) and the Kalman filter are used to track road boundaries. Based on road borders and the vehicle's position, the algorithm determines if the car has strayed off the road. The average correct detection percentage is 93.82 percent using our approach on the Catltech dataset.

Zhang et al., 2018 [16] offers a system that uses a 3D-LiDAR sensor to automatically partition the road and recognize the lanes. The point cloud data from the sensor is used to differentiate between on-road and off-road locations. The off-road data is then used to suggest a sliding-beam approach for segmenting the route. Finally, a curb-detection approach is used to determine the location of curbs for each road section. The suggested technique is validated using data from Tongji University's VeCaN lab's self-driving automobile. The results of the off-line trial show that the curbs can be successfully derived. With an average accuracy of 84.89% and recall of 82.87%, the average F1 score is 83.73%. Furthermore, in real-time testing, the average processing time per frame is around 12 milliseconds, which is sufficient for self-driving.

The Scene Understanding Physics-Enhanced Real-time (SUPER) method is introduced by **Lu et al., 2021 [13]** as a lane identifying system. A hierarchical semantic segmentation network extracts scene information for lane inference in the proposed technique. They train the proposed system using heterogeneous data from Vistas, Cityscapes, and Apollo, and then test it on four different datasets: Tusimple, Caltech, URBAN KITTI-ROAD, and Mcity-3000. The proposed method beats existing lane detection models trained on the same dataset, and it also performs well on datasets that have never been trained on. In comparison to the Mobileye, preliminary test results show promising real-time lane detection capabilities.

Traffic Sign Detection:

Zhang et al., 2020 [6] proposes a cascaded R-CNN to obtain the multiscale features for traffic sign detection. Except for the initial layer, each layer of the cascaded network fuses the output bounding box of the preceding layer to perform joint training. Then dot-product and softmax are employed to get weighted multiscale features, which are then fine-tuned to highlight traffic sign characteristics and detection accuracy. Finally, to reduce interference, they increase the number of difficult negative examples in the training dataset to obtain a balanced dataset. The data augment approach improves the German traffic sign training dataset by mimicking complex environmental changes. A number of tests are carried out to see whether the suggested strategy is effective. In the GTSDB dataset, the method's accuracy and recall rates are 98.7% and 90.5%, respectively.

Liu et al., 2020 [11] proposed SADANet, a traffic sign detection system that combines a multiscale prediction network (MSPN) and a domain adaptive network (DAN). The Multiscale Feature Extraction Network (MSPN) focuses on extracting multiscale features. It makes full use of both low-level location and high-level semantic data. DAN is committed to making features domain invariant in the absence of sufficient labelled test data. Using the mapping relationship between the picture representation and the multiscale features, the domain distributions from several scales are successfully aligned. According to trial data, the SADANet is successful at identifying traffic signs and is also competitive when compared to state-of-the-art techniques. The precision of SADANet is 95.59%. It also has a detection rate of 82.88% for small objects and 85.39% for medium objects, respectively.

Jin et al., 2020 [5] presents the MF-SSD method for traffic sign identification. It is an enhanced (Single Shot Detector) SSD algorithm based on multi-feature fusion and improvement. The German Traffic Sign Recognition Benchmark (GTSRB) dataset is used to test the proposed MF-SSD method. The MF-SSD algorithm has advantages in identifying minor traffic signals. The precision measurements of small, medium and large image sizes obtained by the proposed model are 28.8, 67.5 and 82.6 respectively.

Liu et al. 2019 [10] developed a multiscale region-based convolutional neural network (MR-CNN) for traffic sign detection that employs a multiscale deconvolution operation. While inside the region proposal network (RPN), the fused feature map focuses on improving picture resolution and semantic information for minor traffic sign detection. Feature representation is improved using the fused feature map. TsinghuaTencent 100K, the largest dataset available, was used to test the MR-CNN model. In detecting minor traffic signals, the MR-CNN beats previous methods.

2.3 ISSUES IN EXISTING SYSTEM

Many existing systems aim to detect road lanes using edge detection methods. In some models the dataset was not balanced and hence some classes had more images pertaining to them than the rest, this led to wrong classification outputs. All the merits and demerits of some existing systems are taken into consideration to build the proposed system.

2.4 PROPOSED SYSTEM

The proposed system is to detect lane lines in images. We will build a deep learning model using fully connected CNN pre-trained model to detect lane in an image. In the proposed work a VGG-16 convolutional neural network is used for road segmentation. For traffic sign detection, the German Traffic Sign Recognition Benchmark dataset is used. A convolutional neural network is used with ADAM optimizer. The model is then trained and makes predictions on the test data set for which get an accuracy of the system.

2.5 SUMMARY

The literature survey has covered information regarding existing systems. The issues and challenges have been identified in related work. This chapter has highlighted the proposed model.

CHAPTER 3

SYSTEM DESIGN

3.1 GENERAL

This chapter deals with the design aspect of the proposed system. Systems design implies a systematic approach to the design of a system. It may take a bottom- up or top-down approach, but either way the process is systematic wherein it takes into account all related variables of the system that needs to be created from the architecture, to the required hardware and software, right down to the data and how it travels and transforms throughout its travel through the system. Systems design then overlaps with systems analysis, systems engineering and systems architecture.

3.2 SYSTEM ARCHITECTURE

System architecture, also known as a systems model, is a conceptual model that describes a system's structure, actions, and other aspects. A systematic explanation and representation of a system structured in a way that facilitates thinking about the system's structures and behaviors is called an architecture description. System architecture may be made up of system modules and subsystems that will collaborate to execute the overall system. There have been attempts to formalize languages for describing machine architecture, which are referred to as Architecture Description Languages collectively (ADLs). For lane detection, the input image is loaded to the processing module and the output is the image with the detected lane. In the CNN model the image is pre-processed, sampled and the extracted features are sent to the CNN training module. Inside the training module it first passes through the Convolutional layer gets convoluted, then through the max pooling, fully connected layers and then the sigmoid activation function. For traffic sign detection the image with the traffic sign is inputted and the class corresponding to the predicted traffic sign is obtained as output. The image passes through the convolutional layers which produce feature maps which pass through the max pooling layers. Finally, the predicted class is outputted. Fig 3.2.1 and Fig 3.2.2 depicts the architecture diagram of the proposed work.

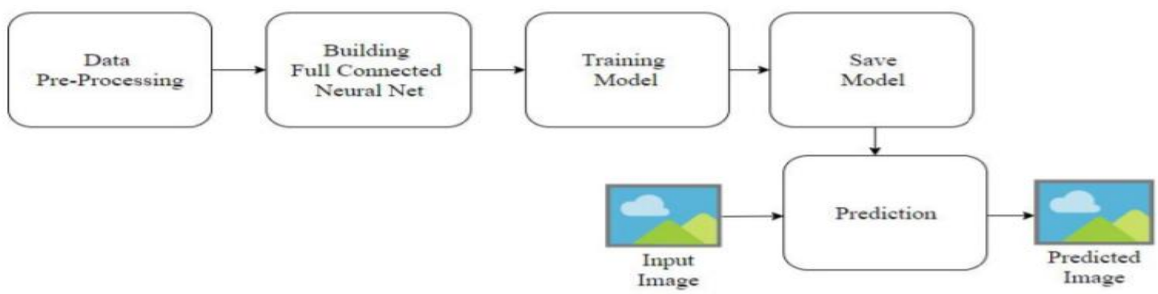


Fig 3.2.1 System Architecture for Lane Detection

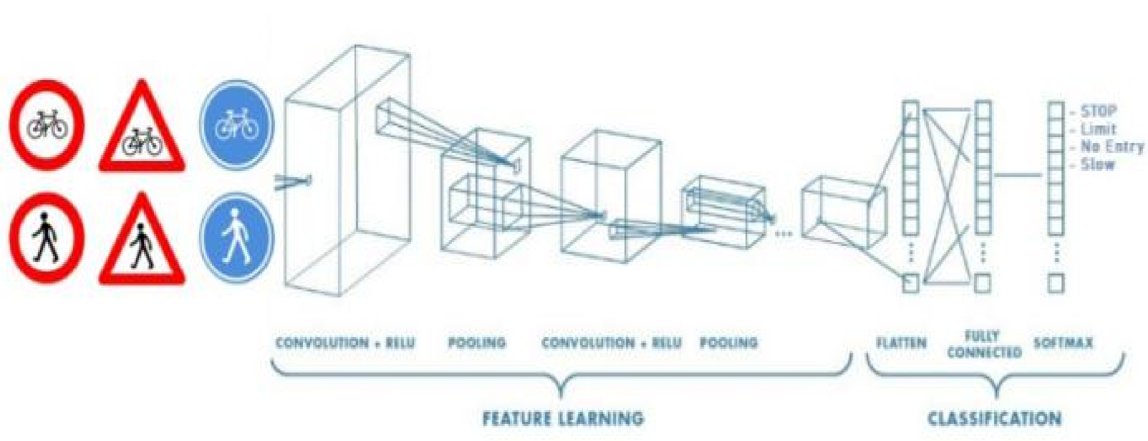


Fig 3.2.2 System Architecture for Traffic Sign Detection

3.3 FUNCTIONAL ARCHITECTURE

A functional architecture is an architectural model that describes the functions of a system and how they interact. It specifies how the functions will work together to achieve the system's goal (s). In most cases, multiple architectures can meet the requirements. The cost, schedule, efficiency, and risk implications of each architecture and its collection of associated assigned specifications are usually different. The functional architecture is used to facilitate the construction of functional and performance tests. It also aids in the creation of verification tasks that are specified to verify the functional, performance, and constraint specifications, in conjunction with the physical architecture. The primary aim of the project is to detect lane lines and recognize traffic signs in self-driving cars. Fig 3.3.1 and Fig 3.3.2 depicts the functions of the proposed system for lane and traffic sign detection.

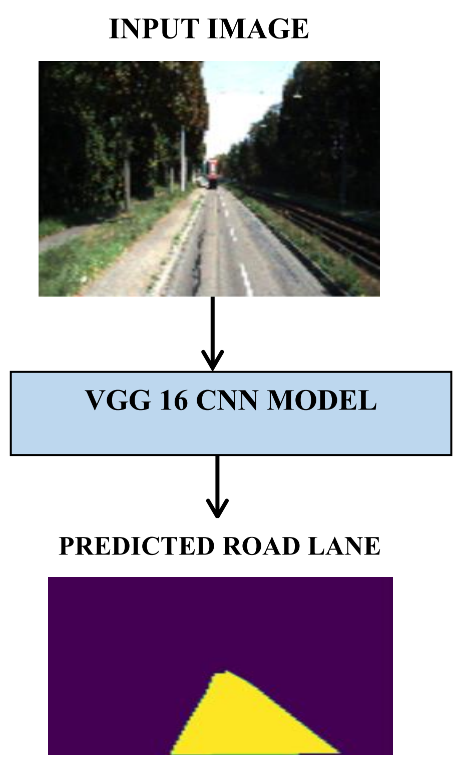


Fig 3.3.1 Functional Architecture for Lane Detection

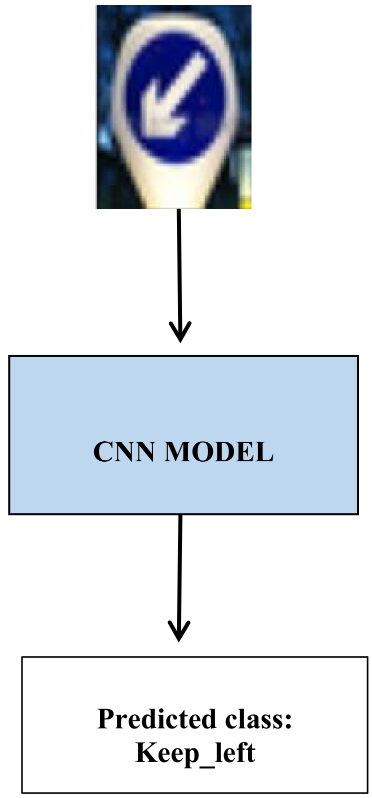


Fig 3.3.2 Functional Architecture for Traffic Sign Recognition

3.4 MODULAR DESIGN

Our approach consists of two main modules: Lane detection and traffic sign detection. Lane detection module consists of two sub-modules: Data Pre-processing and VGG 16 convolutional neural network module. Traffic sign detection consists of two sub-modules: Data Pre-processing module and Convolutional Neural Network module.

3.4.1. LANE DETECTION

3.4.1.1. DATA PRE-PROCESSING

- The dataset consists of road segmentation images from the KITTI Road/Lane Detection Evaluation 2013[2] dataset. The dataset consists of images with roads having marked and unmarked lanes.
- The given data is split into testing and training sets. The images are loaded and the constants are initialized. The images are rescaled and pre-processing is done.

3.4.1.2. VGG 16 CONVOLUTIONAL NEURAL NETWORK

This step is a composition of feature reduction and classification. The Convolution Neural Network is responsible for the network layer operation. The convolution neural network is generally used in the classification of the image but in the proposed system. Fig 3.4.1 depicts the VGG 16 CNN Architecture for lane detection.

The convolution network has a contribution of (224,224,3). The initial two layers have 64 channels with 3*3 channel sizes. They share a similar cushioning. After that there is a step (2, 2) max pool layer. Then, at that point, there are two convolutional layers with 128 channel size and channel size (3, 3). Followed by that, there are two 256-channel convolution layers of (3, 3). At long last there are 2 arrangements of 3 convolution layers and a maximum pool layer. By stacking convolution and max-pooling, we get a (7, 7, 512) include map, which we smooth into a (1, 25088) highlight vector. There are three totally associated layers after that. The main accepts the last component vector as information and results a (1, 4096) vector, the subsequent layer comparably produces a (1, 4096) vector, however the third layer yields 1000 channels, and the result of the third completely associated layer is then passed to standardize the characterization vector. Every one of the secret layers utilize sigmoid as its initiation work

Activation function: Sigmoid activation function is used. It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

Metrics: Different types of metrics are used for the model evaluation such as accuracy, loss etc.

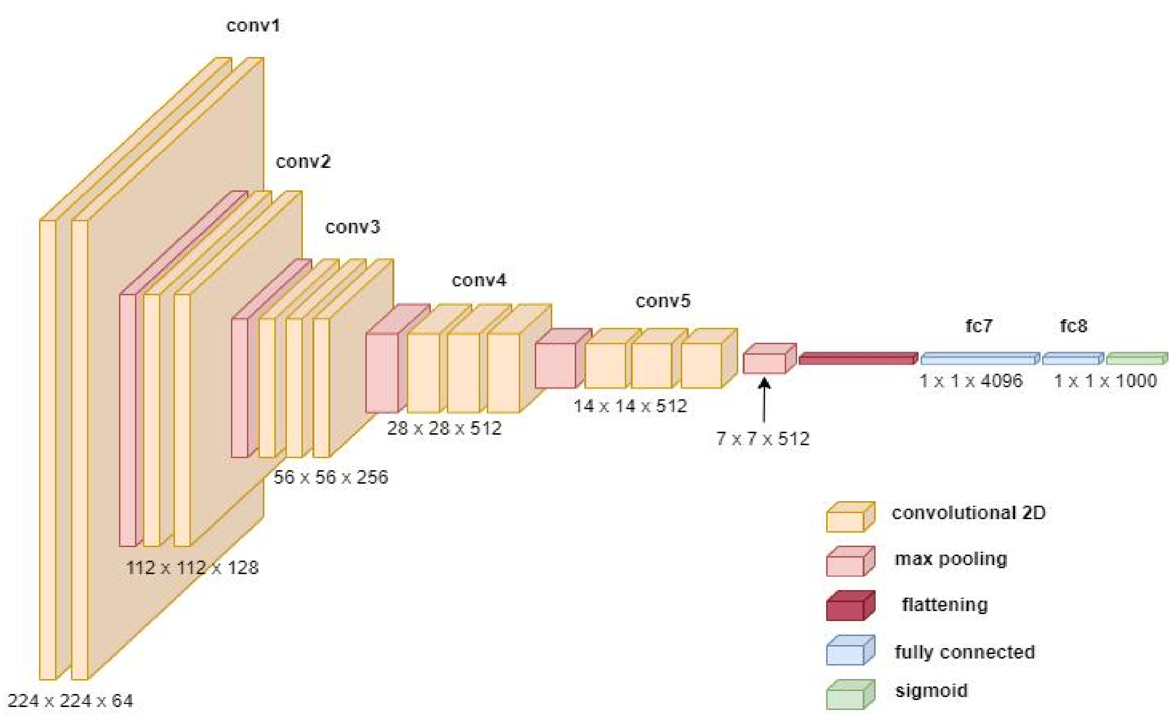


Fig 3.4.1 VGG 16 CNN Architecture for Lane Detection

3.4.2 TRAFFIC SIGN DETECTION

3.4.2.1 DATA PREPROCESSING

- The dataset consists of traffic sign images from the German Traffic Sign Recognition Benchmark dataset. [3]
- It consists of 43 different classes of traffic signs used. It has more than 40,000 images.
- Some of the classes are ‘No Entry’, ‘Keep Left’, ‘Stop’, ‘Yield’, ‘Turn right ahead’ etc.
- All the images are stored in PPM format.
- The size of the dataset is 263 MB and all the images are annotated.
- The dataset is split into training and test set and is used in a pickled format.
- The dataset is pre-processed, augmented and normalized. The training set consists of 34799 images and the testing set consists of 12630 images.

3.4.2.2 CONVOLUTIONAL NEURAL NETWORK

The CNN model requires an image as the input. The CNN model consists of 2 convolutional layers of kernel size 5x5, each followed by a max pooling layer of kernel size 2x2. The convolutional layers produce feature maps which go to the max pooling layers. The max pooling layer is followed by one flattening layer and 2 fully connected layers. Fig 3.4.2 shows the model architecture of the CNN model to detect traffic signs for self-driving cars. It takes an input of size 32x32x3 and gives the class of the traffic sign as the output. It has two convolutional layers which are followed by a 2x2 max pooling layer each. The max pooling layer is followed by the flattening layer and 2 fully connected layers.

RELU activation function is implemented after each convolutional layer. ADAM optimizer with a 0.001 learning rate was used and the CNN model was trained up to 15 epochs. Categorical cross-entropy was used as a loss function to optimize results.

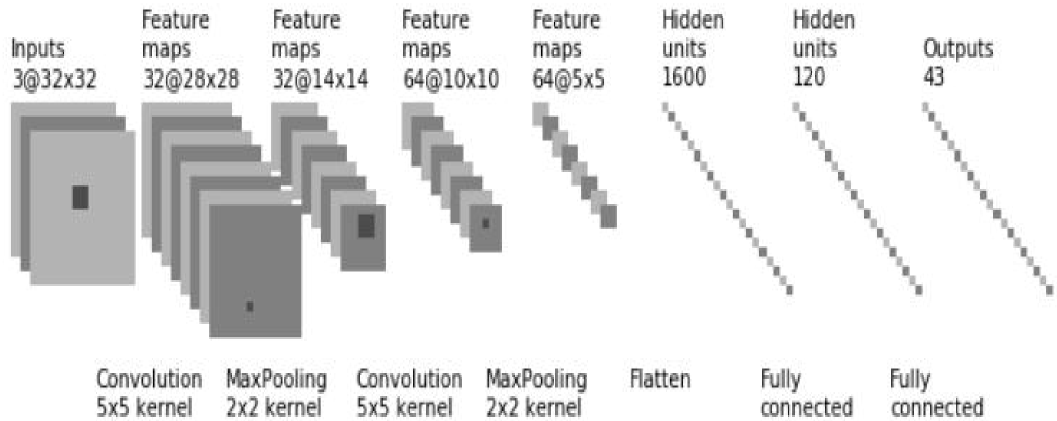


Fig 3.4.2 CNN Architecture for Traffic Sign Detection

3.5. SYSTEM REQUIREMENTS

Hardware Specification

OS/ Platform: Since it is a cross-platform web application, it doesn't require a specific platform or an OS.

Software Specification

Language: Python (3.7+)

IDE: Jupyter Notebook/Google Collab

Libraries: TensorFlow, Keras, librosa, matplotlib, Pandas and numpy.

3.6 SUMMARY

This chapter gives an overview of system design and its importance in the software life cycle. The functional architecture gives the entire functionality of the proposed system along with its modular structure and its interactions between modules.

CHAPTER 4

SYSTEM IMPLEMENTATION

4.1 GENERAL

An implementation is the computer programming and deployment of a technical specification or algorithm as a program, software component, or other computer device. For a given specification or standard, there may be several implementations. High levels of user engagement and management support are usually beneficial to system implementation and participation of users in the design and application of the system. Participation of users in the design and maintenance of information systems has a number of advantages. First, when consumers are heavily involved in system design, they gain more opportunities to shape the system to their priorities and business requirements, as well as more control over the result. Second, they are more likely to welcome change with open arms. Better solutions result from incorporating user experience and skills.

4.2 OVERVIEW OF THE SYSTEM

The following sections go through the various software and hardware requirements listed. Before integrating them to incorporate the proposed framework, it's critical to understand how each variable works on its own.

4.2.1 PYTHON

Python is a high-level, interpreted programming language that can be used for a variety of tasks. Python was created by Guido van Rossum and first published in 1991. Its design style emphasizes code readability, with a lot of white space. It has constructs that allow for simple programming at both small and large scales. Van Rossum was the language community's chairman until July 2018, when he stepped down. Python has a dynamic style structure and memory management that is automated. It has a robust standard library and supports various programming paradigms, including object-oriented, imperative, functional, and procedural. For a wide range of operating systems, Python interpreters are available. CPython, the standard Python implementation, is open source software with a community-based development model, as do virtually all of Python's other implementations. The Python Software Foundation, a non-profit organization, oversees Python and CPython. Python is designed to be a language that is simple to understand. Its formatting is clean and uncluttered, and it mostly uses English keywords instead of punctuation in other languages. It does not use curly brackets to delimit blocks, and semicolons after statements are optional, unlike many other languages. In comparison to C or Pascal, it has fewer syntactic exceptions and special cases.

4.2.2 TENSORFLOW

TensorFlow is a machine learning software library that is free and open-source. It can be used for a variety of activities, but it focuses on deep neural network training and inference. TensorFlow is a data-flow and differentiable programming-based symbolic math library. At Google, it's used for both research and development. The Google Brain team created TensorFlow for internal Google use. It was released in 2015 under the Apache License 2.0.

4.2.3 KERAS

Keras is an open-source software library for artificial neural networks that offers a Python interface. Keras serves as a user interface for TensorFlow. Keras supported a variety of backends up until version 2.3, including TensorFlow, Microsoft Cognitive Toolkit, Theano, and PlaidML. Only TensorFlow is supported as of version 2.4. It is user-friendly, scalable, and extensible, with the goal of allowing fast experimentation with deep neural networks. It was created as part of the ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System) research project, and François Chollet, a Google engineer, is the primary author and maintainer. Chollet is also the creator of the deep neural network model XCeption.

4.2.4 DEEP LEARNING MODEL

CONVOLUTIONAL NEURAL NETWORK

In deep learning, a convolutional neural network (CNN/ConvNet) is a class of deep neural networks, most commonly applied to analyze visual imagery. Here we introduced a convolutional neural network for music tagging. Convolutional neural networks are composed of multiple layers of artificial neurons. The first layer usually extracts basic features such as horizontal or diagonal edges. This output is passed on to the next layer which detects more complex features such as corners or combinational edges. As we move deeper into the network it can identify even more complex features such as objects, faces, etc.

CNN is patterned to process multidimensional array data in which the convolutional layer takes a stack of feature maps, like the pixels of those color channels, and convolves each feature map with a set of learnable filters to obtain a new stack of output feature maps as input. Based on the activation map of the final convolution layer, the classification layer outputs a set of confidence scores (values between 0 and 1) that specify how likely the image is to belong to a "class."

In a CNN, the input is a tensor with a shape: (number of inputs) x (input height) x (input width) x (input channels). After passing through a convolutional layer, the image becomes abstracted to a feature map, also called an activation map, with shape: (number of inputs) x (feature map height) x (feature map width) x (feature map channels).

Convolutional layer within a CNN generally has the following attributes:

- Convolutional filters/kernels defined by a width and height (hyper-parameters).
- The number of input channels and output channels (hyper-parameters). One layer's input channels must equal the number of output channels (also called depth) of its input.
- Additional hyperparameters of the convolution operation, such as: padding, stride, and dilation.

POOLING LAYER:

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data by reducing the dimensions.
The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation.

Accepts a volume of size $w1 \times H1 \times D1$

Requires two hyper parameters:

-their spatial extent F

-the stride S

Produces a volume of size $w2 \times H2 \times D2$

where: $W2 = (W1-F)/\ S + 1$

$H2 = (H1-F)/\ S + 1$

$D2 = D1$

FULLY CONNECTED LAYER:

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is the same as a traditional multi-layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

4.3 MODEL IMPLEMENTATION

4.3.1 LANE DETECTION IN SELF-DRIVING CARS

lane_detection.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text Copy to Drive

```python
import pandas as pd
import numpy as np
import os
import random
import tensorflow as tf
import cv2
from tqdm import tqdm
import datetime
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Concatenate
from tensorflow.keras.layers import Input, Add, Conv2DTranspose
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.losses import SparseCategoricalCrossentropy, MeanSquaredError, BinaryCrossentropy
from tensorflow.keras.utils import plot_model
from tensorflow.keras import callbacks

from  matplotlib import pyplot as plt
import matplotlib.image as mpimg
from IPython.display import clear_output
%matplotlib inline

from IPython.display import HTML
from base64 import b64encode
```

Fig 4.3.1.1 Importing the Required Libraries

```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```python
# Load directories
train_data_dir = "/content/drive/MyDrive/training/image_2/"
train_gt_dir = "/content/drive/MyDrive/training/gt_image_2/"

test_data_dir = "/content/drive/MyDrive/testing/"
```

Fig 4.3.1.2 Loading the Directories

```
[ ] # Number of training examples
    TRAINSET_SIZE = int(len(os.listdir(train_data_dir)) * 0.8)
    print(f"Number of Training Examples: {TRAINSET_SIZE}")

    VALIDSET_SIZE = int(len(os.listdir(train_data_dir)) * 0.1)
    print(f"Number of Validation Examples: {VALIDSET_SIZE}")

    TESTSET_SIZE = int(len(os.listdir(train_data_dir)) - TRAINSET_SIZE - VALIDSET_SIZE)
    print(f"Number of Testing Examples: {TESTSET_SIZE}")

    Number of Training Examples: 231
    Number of Validation Examples: 28
    Number of Testing Examples: 30

[ ] # Initialize Constants
    IMG_SIZE = 128
    N_CHANNELS = 3
    N_CLASSES = 1
    SEED = 123
```

Fig 4.3.1.3 Generating Training Examples and initializing Constants

```
# Generate dataset variables
all_dataset = tf.data.Dataset.list_files(train_data_dir + "*.png", seed=SEED)
all_dataset = all_dataset.map(parse_image)

train_dataset = all_dataset.take(TRAINSET_SIZE + VALIDSET_SIZE)
val_dataset = train_dataset.skip(TRAINSET_SIZE)
train_dataset = train_dataset.take(TRAINSET_SIZE)
test_dataset = all_dataset.skip(TRAINSET_SIZE + VALIDSET_SIZE)

[ ] # Tensorflow function to rescale images to [0, 1]
    @tf.function
    def normalize(input_image: tf.Tensor, input_mask: tf.Tensor) -> tuple:
        input_image = tf.cast(input_image, tf.float32) / 255.0
        return input_image, input_mask

    # Tensorflow function to apply preprocessing transformations
    @tf.function
    def load_image_train(datapoint: dict) -> tuple:
        input_image = tf.image.resize(datapoint['image'], (IMG_SIZE, IMG_SIZE))
        input_mask = tf.image.resize(datapoint['segmentation_mask'], (IMG_SIZE, IMG_SIZE))

        if tf.random.uniform(()) > 0.5:
            input_image = tf.image.flip_left_right(input_image)
            input_mask = tf.image.flip_left_right(input_mask)

        input_image, input_mask = normalize(input_image, input_mask)

        return input_image, input_mask

    # Tensorflow function to preprocess validation images
    @tf.function
    def load_image_test(datapoint: dict) -> tuple:
        input_image = tf.image.resize(datapoint['image'], (IMG_SIZE, IMG_SIZE))
        input_mask = tf.image.resize(datapoint['segmentation_mask'], (IMG_SIZE, IMG_SIZE))

        input_image, input_mask = normalize(input_image, input_mask)
        return input_image, input_mask
```

Fig 4.3.1.4 Generating Dataset Variables

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 4s 0us/step
553476096/553467096 [==============================] - 4s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 fc1 (Dense)                 (None, 4096)              102764544

 fc2 (Dense)                 (None, 4096)              16781312

 predictions (Dense)         (None, 1000)              4097000

=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

Fig 4.3.1.5 Plotting the Model Summary

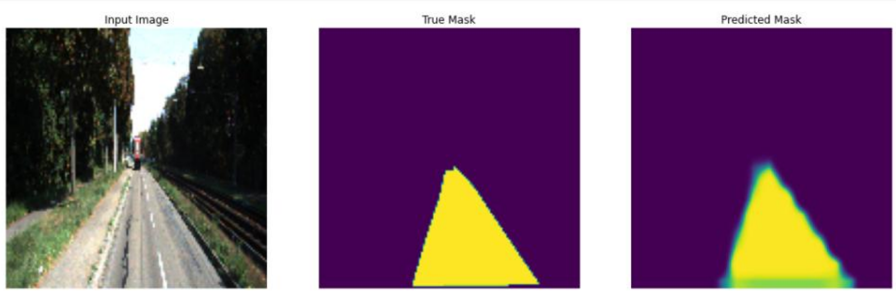
```
[ ] # Define input shape
    input_shape = (IMG_SIZE, IMG_SIZE, N_CHANNELS)
```

```
[ ] # Generate a new model using the VGG network
    # Input
    inputs = Input(input_shape)

    # VGG network
    vgg16_model = VGG16(include_top = False, weights = 'imagenet', input_tensor = inputs)

    # Encoder Layers
    c1 = vgg16_model.get_layer("block3_pool").output
    c2 = vgg16_model.get_layer("block4_pool").output
    c3 = vgg16_model.get_layer("block5_pool").output

    # Decoder
    u1 = UpSampling2D((2, 2), interpolation = 'bilinear')(c3)
    d1 = Add()([u1, c2])
    d1 = Conv2D(256, 1, activation = 'sigmoid')(d1)

    u2 = UpSampling2D((2, 2), interpolation = 'bilinear')(d1)
    d2 = Add()([u2, c1])
    d2 = Conv2D(256, 1, activation = 'sigmoid')(d2)

    # Output
    u3 = UpSampling2D((8, 8), interpolation = 'bilinear')(d2)
    outputs = Conv2D(N_CLASSES, 1, activation = 'sigmoid')(u3)

    model = Model(inputs, outputs, name = "VGG_FCN8")

    Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
    58892288/58889256 [==============================] - 0s 0us/step
    58900480/58889256 [==============================] - 0s 0us/step
```

```
[ ] m_iou = tf.keras.metrics.MeanIoU(2)
    model.compile(optimizer=Adam(),
                  loss=BinaryCrossentropy(),
                  metrics=['accuracy',m_iou])
```

Fig 4.3.1.6 Defining the VGG Network

```
[ ] model_history = model.fit(dataset['train'], epochs=EPOCHS,
                              steps_per_epoch=STEPS_PER_EPOCH,
                              validation_data = dataset["val"],
                              validation_steps=VALIDATION_STEPS,
                              callbacks = callbacks)
```




```
Sample Prediction after epoch 200

WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available metrics are: loss,accuracy,mean_io_u
WARNING:tensorflow:Can save best model only with val_loss available, skipping.
7/7 [==============================] - 8s 1s/step - loss: 0.0308 - accuracy: 0.9858 - mean_io_u: 0.4120
```

Fig 4.3.1.7 Training the Model for 200 Epochs

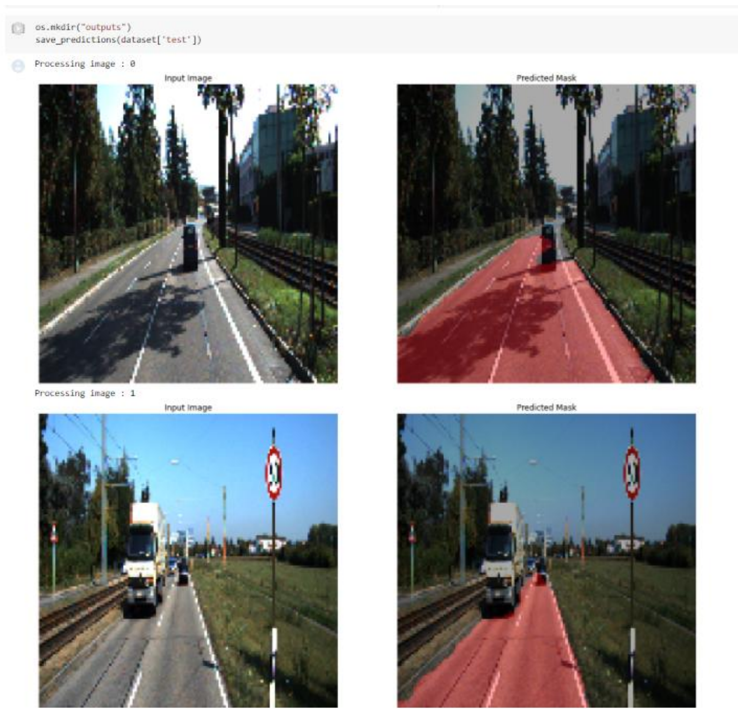


Fig 4.3.1.8 Making Predictions on the Test Data Set

4.3.2 TRAFFIC SIGN RECOGNITION IN SELF-DRIVING CARS

```
[ ] # Load pickled data
    import pickle
    import numpy as np
    import tensorflow as tf
    import random
    import csv
    import os
    from PIL import Image
    from sklearn.utils import shuffle
    from tensorflow.contrib.layers import flatten
    import matplotlib.gridspec as gridspec
    import matplotlib.pyplot as plt
    %matplotlib inline

    /usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/dtypes.py:526: FutureWarning: Passing (type
      _np_qint8 = np.dtype([("qint8", np.int8, 1)])
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/dtypes.py:527: FutureWarning: Passing (type
      _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/dtypes.py:528: FutureWarning: Passing (type
      _np_qint16 = np.dtype([("qint16", np.int16, 1)])
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/dtypes.py:529: FutureWarning: Passing (type
      _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/dtypes.py:530: FutureWarning: Passing (type
      _np_qint32 = np.dtype([("qint32", np.int32, 1)])
```

Fig 4.3.2.1 Importing the Required Libraries

```
[ ] training_file = '/content/drive/MyDrive/train.p'
    validation_file = '/content/drive/MyDrive/valid.p'
    testing_file = '/content/drive/MyDrive/test.p'

    with open(training_file, mode='rb') as f:
        train = pickle.load(f)
    with open(validation_file, mode='rb') as f:
        valid = pickle.load(f)
    with open(testing_file, mode='rb') as f:
        test = pickle.load(f)

    X_train, y_train = train['features'], train['labels']
    X_valid, y_valid = valid['features'], valid['labels']
    X_test, y_test = test['features'], test['labels']
```

```
[ ] print(len(y_train))
    print(len(y_valid))
    print(len(y_test))

    print(X_test.shape)
    print(test['labels'])
```

```
34799
12630
12630
(12630, 32, 32, 3)
[16  1 38 ...  6  7 10]
```

Fig 4.3.2.2 Loading the Directories

```
[ ] print('number of data for each class (augmented dataset):')
    show_dataset_summary(train)
```

```
number of data for each class (augmented dataset):
```

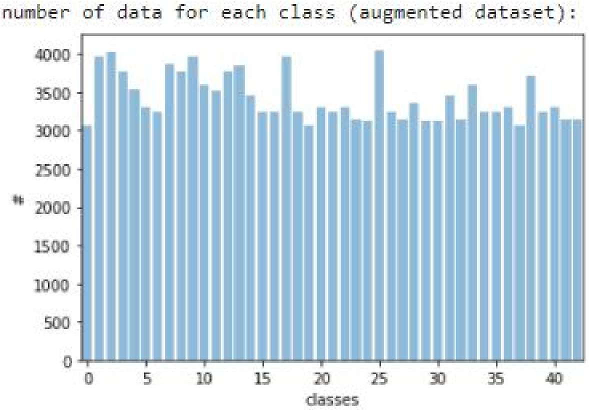


Fig 4.3.2.3 Generating the Dataset Plot

```
[ ] # tensorflow graph input
    x = tf.placeholder(tf.float64, (None, 32, 32, 3))
    x = tf.cast(x, tf.float32)
    y = tf.placeholder(tf.uint8, (None))
    one_hot_y = tf.one_hot(y, n_classes)
    keep_prob = tf.placeholder(tf.float32)
```

```
[ ] # parameters
    rate = 0.001
    EPOCHS = 15
    BATCH_SIZE = 4096
    display_step = 2
    save_step = 5

    # do train?
    do_train = 1

    # select device to be used
    device_type = "/gpu:0"
```

Fig 4.3.2.4 Initializing the Parameters

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 4s 0us/step
553476096/553467096 [==============================] - 4s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 fc1 (Dense)                 (None, 4096)              102764544

 fc2 (Dense)                 (None, 4096)              16781312

 predictions (Dense)         (None, 1000)              4097000

=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

Fig 4.3.2.5 Plotting the Model Summary

```
def LeNet_he(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x32.
    conv1_w = tf.Variable(tf.truncated_normal(shape=(5,5,3,32), mean=mu, stddev=np.sqrt(2/(5*5*3))))
    conv1_b = tf.Variable(tf.zeros(32))
    conv1 = tf.nn.conv2d(x, conv1_w, strides=[1,1,1,1], padding='VALID') + conv1_b

    # batch normalization
    mean_, var_ = tf.nn.moments(conv1, [0,1,2])
    conv1 = tf.nn.batch_normalization(conv1, mean_, var_, 0, 1, 0.0001)


    # Activation.
    conv1 = tf.nn.relu(conv1)

    # Pooling. Input = 28x28x32. Output = 14x14x32.
    conv1 = tf.nn.max_pool(conv1, ksize=[1,2,2,1],strides=[1,2,2,1], padding='VALID')

    # Layer 2: Convolutional. Output = 10x10x64.
    conv2_w = tf.Variable(tf.truncated_normal(shape=(5,5,32,64), mean=mu, stddev=np.sqrt(2/(5*5*32))))
    conv2_b = tf.Variable(tf.zeros(64))
    conv2 = tf.nn.conv2d(conv1, conv2_w, strides=[1,1,1,1], padding='VALID') + conv2_b

    # batch normalization
    mean_, var_ = tf.nn.moments(conv2, [0,1,2])
    conv2 = tf.nn.batch_normalization(conv2, mean_, var_, 0, 1, 0.0001)


    # Activation.
    conv2 = tf.nn.relu(conv2)

    # Pooling. Input = 10x10x64. Output = 5x5x64.
    conv2 = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')

    # Flatten. Input = 5x5x64. Output = 1600.
    fc0 = flatten(conv2)

    # Layer 3: Fully Connected. Input = 1600. Output = 120.
    fc1_w = tf.Variable(tf.truncated_normal(shape=(1600,120), mean=mu, stddev=np.sqrt(2/(1600))))
    fc1_b = tf.Variable(tf.zeros(120))
    fc1 = tf.matmul(fc0, fc1_w) + fc1_b

    # batch normalization
    mean_, var_ = tf.nn.moments(fc1, axes=[0])
    fc1 = tf.nn.batch_normalization(fc1, mean_, var_, 0, 1, 0.0001)
```

Fig 4.3.2.6 Defining the LeNet Architecture

```
    if do_train == 0:
#        epoch = epoch_to_restore
        saver.restore(sess, tf.train.latest_checkpoint('nets/'))
        print("Model restored.")

        # calculate training accuracy
        batch_size_for_cal = 10000
        n_train_right = 0
        offset = 0
        tstep = np.floor(X_train_.shape[0]/10000)
        for t in range(tstep.astype(int)):
            if X_train_.shape[0] - (batch_size_for_cal + offset) < 0:
                batch_size_for_cal = X_train_.shape[0] - offset
            n_train_right += sess.run(accuracy_operation,
                                  feed_dict={x: X_train_[offset:offset+batch_size_for_cal]
                                             y: y_train[offset:offset+batch_size_for_cal]}

        train_acc = n_train_right/X_train_.shape[0]

        # validation and test accuracy
        valid_acc = sess.run(accuracy_operation, feed_dict={x: X_valid_, y: y_valid})
        test_acc = sess.run(accuracy_operation, feed_dict={x: X_test_, y:y_test})
        print("Train accuracy: %.3f" % (train_acc))
        print("Validation accuracy: %.3f" % (valid_acc))
        print("Test accuracy: %.3f" % (test_acc))
```

```
Training...

Epoch: 001/015, loss: 2.629365683, train acc: 0.596, valid acc: 0.640
Epoch: 003/015, loss: 0.939661688, train acc: 0.856, valid acc: 0.833
Epoch: 005/015, loss: 0.552436625, train acc: 0.913, valid acc: 0.861
Epoch: 007/015, loss: 0.400535308, train acc: 0.932, valid acc: 0.871
Epoch: 009/015, loss: 0.317606640, train acc: 0.948, valid acc: 0.876
Epoch: 011/015, loss: 0.271784269, train acc: 0.956, valid acc: 0.875
Epoch: 013/015, loss: 0.238957231, train acc: 0.971, valid acc: 0.874
Epoch: 015/015, loss: 0.208246055, train acc: 0.955, valid acc: 0.867
Test accuracy: 0.867
```

Fig 4.3.2.7 Training the Model for 200 Epochs

```
[ ] ### Run the predictions here and use the model to output the prediction for each image.
    ### Make sure to pre-process the images with the same pre-processing pipeline used earlier.
    ### Feel free to use as many code cells as needed.
    X_img_test = normalize_images(img_set)

    with tf.Session(config=config) as sess:
        saver.restore(sess, tf.train.latest_checkpoint('nets/'))
        predict_type = sess.run(tf.argmax(logits, 1), feed_dict={x: X_img_test})
        print(predict_type)

    INFO:tensorflow:Restoring parameters from nets/traffic_sign_lenet-10
    [28 39 12  2 14]

[ ] print(predict_type.dtype)
    print(predict_type[0])
    print(predict_type[1])

    int64
    28
    39

[ ] with open('signnames.csv') as csvfile:
        reader = csv.DictReader(csvfile)
        SignNames = []
        for row in reader:
            csv_ = row['SignName']
            SignNames.append(csv_)

[ ] for predict_ind in range(len(predict_type)):
        print('File name: %-025s Predicted name: %s'%(target_img_file_names[predict_ind][5:-4] , SignNames[predict_type[predict_ind]]))

    File name: childeren_crossing        Predicted name: Children crossing
    File name: keep_left                 Predicted name: Keep left
    File name: prioirty_road             Predicted name: Priority road
    File name: speed_limit_50            Predicted name: Speed limit (50km/h)
    File name: stop                      Predicted name: Stop
```

Fig 4.3.2.8 Making Predictions on the Test Data Set

4.4. SUMMARY

This chapter brought out the analysis of each technology used to develop this project. This chapter also brought out the basic system implementations such as the platforms, languages, and tools used here. And the screenshots of different modules implemented using those platforms, languages, and tools.

CHAPTER 5
SYSTEM TESTING AND PERFORMANCE ANALYSIS

5.1 GENERAL

Once the design aspect of the system is finalized the system enters the testing phase. Testing is an integral part of the entire development and maintenance process. Testing is mainly performed to identify errors. It is used for quality assurance. The goal of the testing phase is to verify that the specification has been accurately completed and incorporated into the design, as well as to ensure the correctness of the design itself. Testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and to verify that the software product is fit for use.

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- Meets the requirements that guided its design and development
- Responds correctly to all kinds of inputs
- Performs its functions within an acceptable time
- Is sufficiently usable
- Can run in its intended environments

- Achieves the general result of its stakeholder's desire.

As the number of possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources. As a result, software testing typically attempts to execute a program or application with the intent of finding software bugs. The job of testing is an iterative process as when one bug is fixed; it can illuminate other, deeper bugs, or can even create new ones.

5.2 TEST CASE

Some of the cases are listed below,

- The lane detection model is trained to detect lane lines in an given image.
- The input of the model should be the image containing the road.
- The traffic sign detection model is trained to predict the class of the traffic sign in a given image.
- The input of the model should be the image containing the traffic sign.

5.3 PERFORMANCE MEASURES

Performance measurement is the process of collecting, analyzing and/or reporting information regarding the performance of an individual, group, organization, system or component. Three metrics are taken into consideration, namely

- Training accuracy and loss
- Validation accuracy and loss

- Accuracy score
- Confusion matrix

5.4 PERFORMANCE ANALYSIS

5.4.1 LANE DETECTION IN SELF-DRIVING CARS

The proposed work is evaluated using different parameters. The KITTI Road/Lane Detection Evaluation 2013dataset [8] is divided into training and testing sets. We calculate the loss, train accuracy and valid accuracy for the model. The loss function keeps decreasing with every epoch and the accuracy keeps increasing. After training the model for 200 epochs, the model gave an accuracy of about 98.58% and loss value of 0.0308.

Accuracy score: 0.98%

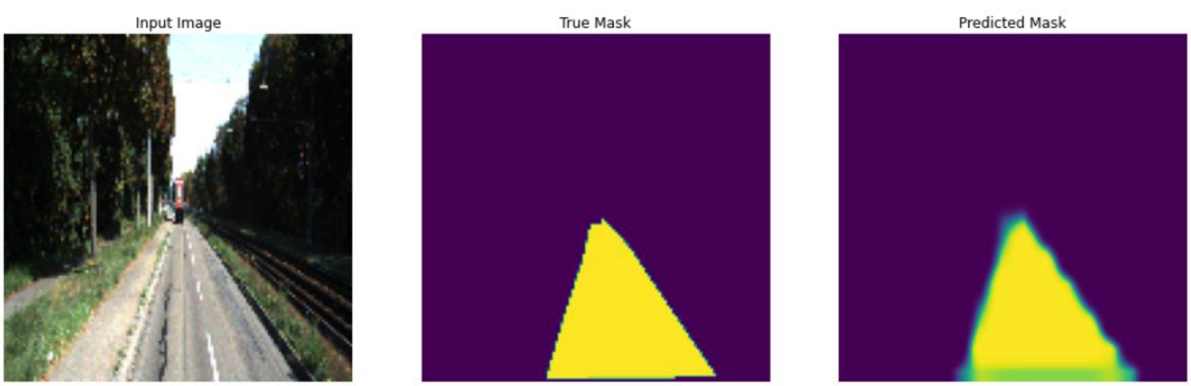


Fig 5.4.1.1 Sample prediction after 200 epochs

Fig 5.4.1.2 Sample Predictions for Lane Segmentation

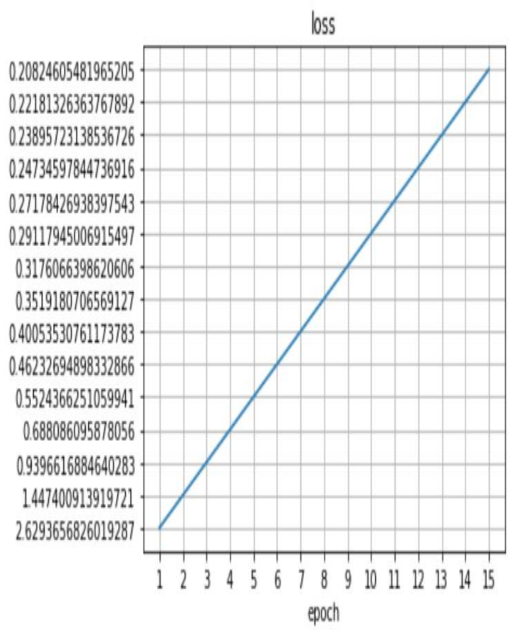
5.4.2 TRAFFIC SIGN DETECTION IN SELF-DRIVING CARS

The proposed work is evaluated using different parameters. The training data of the German Traffic Sign Recognition Benchmark dataset [15] has 34799 image samples and the testing data has 12630 image samples. We calculate the loss, train accuracy and valid accuracy for the CNN model as shown in Table 5.4.2.1.

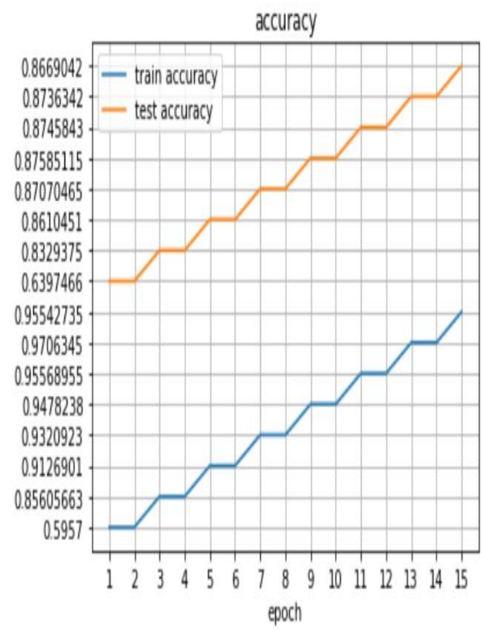
Table 5.4.2.1 Performance evaluation of CNN model for traffic sign detection

Epoch	Loss	train_acc	valid acc
1	2.629365683	0.596	0.640
3	0.939661688	0.856	0.833
5	0.552436625	0.913	0.861
7	0.400535308	0.932	0.871
9	0.317606640	0.948	0.876
11	0.271784269	0.956	0.875
13	0.238957231	0.971	0.874
15	0.208246055	0.955	0.867

The loss function keeps decreasing with every epoch and the accuracy keeps increasing. The training set gave an accuracy of 95% at the end of the 15th epoch. . From the plot of loss in fig 5.4.2.1a, it can be seen that the model has comparable performance. From the plot of accuracy in fig 5.4.2.1b it can be seen that the model has not over-learned the training dataset, showing comparable skill on both the training and testing datasets. The test dataset gave an accuracy of 86.7%.



a. model loss plot



b. model accuracy plot

Fig 5.4.2.1 Performance plots for CNN for Traffic Sign Detection

Fig 5.4.2.2 Test Images and their Predicted Classes

Fig 5.4.2.2 shows 5 test images and their corresponding predicted classes. A confusion matrix describes the performance of a classification model. It is drawn using a set of test data for which the true values are known. Fig 5.4.2.3 shows the normalized confusion matrix for the CNN model for traffic sign detection.

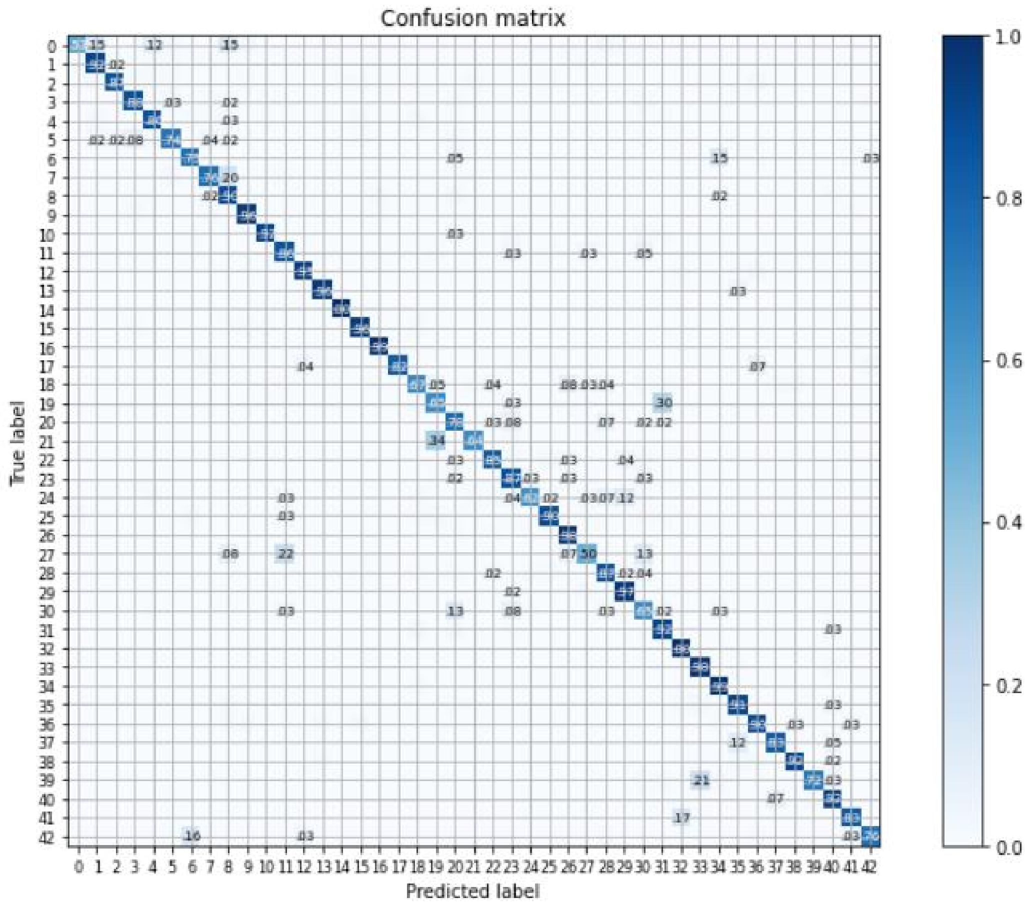


Fig 5.4.2.3 Confusion Matrix for the CNN Model for Traffic Sign Detection

5.5 SUMMARY

This chapter brought out the general description of the different testing processes applicable to the entire project development. It also considers the performance analysis of a system that proved to improve the level of user satisfaction compared to the existing system.

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

The model presented deep learning methodology for Lane segmentation and traffic sign detection in self-driving cars. Lane segmentation was performed on the KITTI Road/Lane Detection Evaluation 2013[8] dataset using a VGG16 CNN model. It performed well and segmented lanes correctly in most of the test images with an accuracy of about 98.58 %. For traffic sign detection the German Traffic Sign Recognition Benchmark dataset [15] was used. A CNN model with ADAM optimizer was trained to give an accuracy of 95%. After all the explanatory analysis of both models, it is clear that both the models provided a satisfactory result. Both models performed with high accuracy. The performances of both the models have been analyzed carefully. The proposed methodology gives greater accuracy when compared to models using other non-deep learning methodologies.

6.2 FUTURE WORK

There are many more opportunities for further research in this area, particularly training a classifier with a larger dataset. On further enhancement of the system, we aim to implement a centralized web application for our model which makes it easier and even more accessible to everyone. We'll try looking for more data and better data augmentation techniques, as well as further improving the model.

6.3 APPENDICES

LaneDetection.ipynb

```
import pandas as pd
import numpy as np
import os
import random
import tensorflow as tf
import cv2
from tqdm import tqdm
import datetime
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Concatenate
from tensorflow.keras.layers import Input, Add, Conv2DTranspose
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.losses import SparseCategoricalCrossentropy, MeanSquaredError, BinaryCrossentropy
from tensorflow.keras.utils import plot_model
from tensorflow.keras import callbacks

from  matplotlib import pyplot as plt
import matplotlib.image as mpimg
from IPython.display import clear_output
%matplotlib inline
```

```python
from IPython.display import HTML
from base64 import b64encode
from google.colab import drive
drive.mount('/content/drive')
Mounted at /content/drive
# Load directories
train_data_dir = "/content/drive/MyDrive/training/image_2/"
train_gt_dir = "/content/drive/MyDrive/training/gt_image_2/"
test_data_dir = "/content/drive/MyDrive/testing/"
# Number of training examples
TRAINSET_SIZE = int(len(os.listdir(train_data_dir)) * 0.8)
print(f"Number of Training Examples: {TRAINSET_SIZE}")
VALIDSET_SIZE = int(len(os.listdir(train_data_dir)) * 0.1)
print(f"Number of Validation Examples: {VALIDSET_SIZE}")
TESTSET_SIZE = int(len(os.listdir(train_data_dir)) - TRAINSET_SIZE - VALIDSET_SIZE)
print(f"Number of Testing Examples: {TESTSET_SIZE}")
Number of Training Examples: 231
Number of Validation Examples: 28
Number of Testing Examples: 30
# Initialize Constants
IMG_SIZE = 128
N_CHANNELS = 3
N_CLASSES = 1
SEED = 123
# Function to load image and return a dictionary
```

```
def parse_image(img_path: str) -> dict:
    image = tf.io.read_file(img_path)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.convert_image_dtype(image, tf.uint8)
     # Three types of img paths: um, umm, uu

 # gt image paths: um_road, umm_road, uu_road
    mask_path = tf.strings.regex_replace(img_path, "image_2", "gt_image_2")
    mask_path = tf.strings.regex_replace(mask_path, "um_", "um_road_")
    mask_path = tf.strings.regex_replace(mask_path, "umm_", "umm_road_")
    mask_path = tf.strings.regex_replace(mask_path, "uu_", "uu_road_")
    mask = tf.io.read_file(mask_path)
    mask = tf.image.decode_png(mask, channels=3)
    non_road_label = np.array([255, 0, 0])
    road_label = np.array([255, 0, 255])
    other_road_label = np.array([0, 0, 0])

# Generate dataset variables
all_dataset = tf.data.Dataset.list_files(train_data_dir + "*.png", seed=SEED)
all_dataset = all_dataset.map(parse_image)
train_dataset = all_dataset.take(TRAINSET_SIZE + VALIDSET_SIZE)
val_dataset = train_dataset.skip(TRAINSET_SIZE)
train_dataset = train_dataset.take(TRAINSET_SIZE)
test_dataset = all_dataset.skip(TRAINSET_SIZE + VALIDSET_SIZE)

# Tensorflow function to rescale images to [0, 1]
@tf.function
```

```
def normalize(input_image: tf.Tensor, input_mask: tf.Tensor) -> tuple:
    input_image = tf.cast(input_image, tf.float32) / 255.0
    return input_image, input_mask
BATCH_SIZE = 32
BUFFER_SIZE = 1000

dataset = {"train": train_dataset, "val": val_dataset, "test": test_dataset}

# -- Train Dataset --#
dataset['train']                    =                    dataset['train'].map(load_image_train,
num_parallel_calls=tf.data.AUTOTUNE)
dataset['train'] = dataset['train'].shuffle(buffer_size=BUFFER_SIZE, seed=SEED)
dataset['train'] = dataset['train'].repeat()
dataset['train'] = dataset['train'].batch(BATCH_SIZE)
dataset['train'] = dataset['train'].prefetch(buffer_size=tf.data.AUTOTUNE)

#-- Testing Dataset --#
dataset['test'] = dataset['test'].map(load_image_test)
dataset['test'] = dataset['test'].batch(BATCH_SIZE)
dataset['test'] = dataset['test'].prefetch(buffer_size=tf.data.AUTOTUNE)
print(dataset['train'])
print(dataset['val'])
print(dataset['test'])

# Get VGG-16 network as backbone
vgg16_model = VGG16()
vgg16_model.summary()
```

```python
# Define input shape
input_shape = (IMG_SIZE, IMG_SIZE, N_CHANNELS)
# Generate a new model using the VGG network
# Input
inputs = Input(input_shape)

# VGG network
vgg16_model = VGG16(include_top = False, weights = 'imagenet', input_tensor =
inputs)

# Encoder Layers
c1 = vgg16_model.get_layer("block3_pool").output
c2 = vgg16_model.get_layer("block4_pool").output
c3 = vgg16_model.get_layer("block5_pool").output

# Decoder
u1 = UpSampling2D((2, 2), interpolation = 'bilinear')(c3)
d1 = Add()([u1, c2])
d1 = Conv2D(256, 1, activation = 'sigmoid')(d1)

u2 = UpSampling2D((2, 2), interpolation = 'bilinear')(d1)
d2 = Add()([u2, c1])
d2 = Conv2D(256, 1, activation = 'sigmoid')(d2)

# Output
u3 = UpSampling2D((8, 8), interpolation = 'bilinear')(d2)
outputs = Conv2D(N_CLASSES, 1, activation = 'sigmoid')(u3)
```

```python
model = Model(inputs, outputs, name = "VGG_FCN8")
m_iou = tf.keras.metrics.MeanIoU(2)
model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=['accuracy',m_iou])
def create_mask(pred_mask: tf.Tensor) -> tf.Tensor:
    # Round to closest
    pred_mask = tf.math.round(pred_mask)

    # [IMG_SIZE, IMG_SIZE] -> [IMG_SIZE, IMG_SIZE, 1]
    pred_mask = tf.expand_dims(pred_mask, axis=-1)
    return pred_mask

# Function to show predictions
def show_predictions(dataset=None, num=1):
    if dataset:
        # Predict and show image from input dataset
        for image, mask in dataset.take(num):
            pred_mask = model.predict(image)
            display_sample([image[0], true_mask, create_mask(pred_mask)])
    else:
        # Predict and show the sample image
        inference = model.predict(sample_image)
        display_sample([sample_image[0], sample_mask[0],
                        inference[0]])
```

```
for image, mask in dataset['train'].take(1):
    sample_image, sample_mask = image, mask
show_predictions()
# Callbacks and Logs
class DisplayCallback(callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        clear_output(wait=True)
        show_predictions()
        print ('\nSample Prediction after epoch {}\n'.format(epoch+1))

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-
%H%M%S"))

callbacks = [
    DisplayCallback(),
    callbacks.TensorBoard(logdir, histogram_freq = -1),
    callbacks.EarlyStopping(patience = 10, verbose = 1),
    callbacks.ModelCheckpoint('best_model.h5', verbose = 1, save_best_only =
True)
]

# Set Variables
EPOCHS = 200
STEPS_PER_EPOCH = TRAINSET_SIZE // BATCH_SIZE
VALIDATION_STEPS = VALIDSET_SIZE // BATCH_SIZE
model_history = model.fit(dataset['train'], epochs=EPOCHS,
                          steps_per_epoch=STEPS_PER_EPOCH,
```

```
                    validation_data = dataset["val"],
                    validation_steps=VALIDATION_STEPS,
                    callbacks = callbacks)
# Function to calculate mask over image
def weighted_img(img, initial_img, α=1., β=0.5, γ=0.):
    return cv2.addWeighted(initial_img, α, img, β, γ)

# Function to process an individual image and it's mask
def process_image_mask(image, mask):
    # Round to closest
    mask = tf.math.round(mask)

    # Convert to mask image
    zero_image = np.zeros_like(mask)
    mask = np.dstack((mask, zero_image, zero_image))
    mask = np.asarray(mask, np.float32)

    # Convert to image image
    image = np.asarray(image, np.float32)

    # Get the final image
    final_image = weighted_img(mask, image)

    return final_image
# Function to save predictions
def save_predictions(dataset):
```

```
# Function to save the images as a plot
def save_sample(display_list, index):
    plt.figure(figsize=(18, 18))

    title = ['Input Image', 'Predicted Mask']

    for i in range(len(display_list)):
        plt.subplot(1, len(display_list), i+1)
        plt.title(title[i])
        plt.imshow(tf.keras.preprocessing.image.array_to_img(display_list[i]))
        plt.axis('off')
    plt.savefig(f"outputs/{index}.png")
    plt.show()
os.mkdir("outputs")
save_predictions(dataset['test'])
```

TrafficSignDetection.ipynb

```
# Load pickled data
import pickle
import numpy as np
import tensorflow as tf
import random
import csv
import os
from PIL import Image
from sklearn.utils import shuffle
from tensorflow.contrib.layers import flatten
```

```
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
%matplotlib inline
training_file = '/content/drive/MyDrive/train.p'
validation_file = '/content/drive/MyDrive/valid.p'
testing_file = '/content/drive/MyDrive/test.p'

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
print(len(y_train))
print(len(y_valid))
print(len(y_test))

print(X_test.shape)
print(test['labels'])
# Number of training examples
n_train = len(y_train)

# Number of testing examples.
```

```python
n_test = len(y_test)

# What's the shape of an traffic sign image?
image_shape = X_train.shape[1:4]

# How many unique classes/labels there are in the dataset.
n_classes = np.max(y_train) - np.min(y_train) + 1
print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
def show_dataset_summary(pickle_dict):
    X, y = pickle_dict['features'], pickle_dict['labels']
    n_classes = np.max(y_train) - np.min(y_train) + 1
    n_data_of_classes = np.zeros((n_classes,))
    for i in range(n_classes):
        n_data_of_classes[i] = len(y[y == i])
    classes_num = [i for i in range(n_classes)]
    plt.figure()
    plt.bar(classes_num, n_data_of_classes, align="center", alpha=.5 )
    plt.xlabel('classes')
    plt.ylabel('#')
    plt.xlim([0-.5, classes_num[-1]+.5])
    plt.show()
def plot_test_images(images, nc = 15, nr = 4):
    ct = 0
    fig = plt.figure(figsize=(nc, nr))
```

```
    gs = gridspec.GridSpec(nr, nc)
    gs.update(wspace=0.0, hspace=0.0)
    for i in range(nr * nc):
        ax = fig.add_subplot(gs[ct])
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(images[ct,:,:,:])
        ct += 1
    return fig
ind_ = np.random.randint(n_train, size=33)
fig = plot_test_images(X_train[ind_, :, :, :], nc=11, nr=3)
plt.savefig('./data_example.png')



 # initialize augmented (X,y)
    X_train_augmented = X_train
    y_train_augmented = y_train

    for i in range(X_train.shape[0]):
        # If you have less than 3000 data, you increase the number of data by random 
cropping.
        if n_data_of_classes[y_train[i]] <= 3000:
            N_rand = np.floor(3000/n_data_of_classes[y_train[i]])
            N_rand = N_rand.astype(int)
for j in range(N_rand):
             N_now += 1
```

```
# load training images
X_train_PIL = Image.fromarray(X_train[i,:,:,:])

# set width(=height) and start points for random cropping
rw = np.floor(random.random()*12 + 18) # random width (18~30)
rw = int(rw)
rs = np.floor((32 - rw) * random.random()) # random crop start point
rs = int(rs)

# randomly crop and reshape to (32,32,3)
randomly_cropped_image = X_train_PIL.crop((rs,rs,rs+rw,rs+rw))
distorted_image = randomly_cropped_image.resize((32,32), Image.ANTIALIAS)

# randomly adjust brightness
enhancer = ImageEnhance.Brightness(distorted_image)
distorted_image_ = enhancer.enhance(random.random())

# convert image to uint8 array
distorted_image_ = np.array(distorted_image_, dtype=np.uint8)

# append distorted image on X_train
distorted_image_ = distorted_image_[np.newaxis,:] # (expand dimension from (32,32,3) to (1,32,32,3))
X_train_augmented = np.append(X_train_augmented, distorted_image_, axis=0)
y_train_augmented = np.append(y_train_augmented, [y_train[i]], axis=0)
```

```
        printProgressBar(N_now, N_rand_total, prefix = 'Progress:', suffix = 'Complete', length = 50)

    print("augmented training images are generated.")
    print("%d -> %d" %(X_train.shape[0], X_train_augmented.shape[0]))
if generate_distorted_images == True:
    # make dictionary
    train_augmented = {'features': X_train_augmented, 'labels': y_train_augmented}

    # save augmented training dataset
    adata_name = "train_aug.p"
    with open(adata_name, "wb") as f:
        pickle.dump(train_augmented, f)
with open("/content/drive/MyDrive/train_aug.p", mode="rb") as f:
    train = pickle.load(f)

X_train, y_train = train['features'], train['labels']
print('number of data for each class (augmented dataset):')
show_dataset_summary(train)
number of data for each class (augmented dataset):

# normalize images into [-0.5, 0.5]
# after normalization, images become float64 type.
def normalize_images(X_):
    return X_ / 255 - 0.5
```

```
X_train_ = normalize_images(X_train)
print('training set is normalized')

X_valid_ = normalize_images(X_valid)
print('Validation set is normalized')

X_test_ = normalize_images(X_test)
print('Test set is normalized')
training set is normalized
Validation set is normalized
Test set is normalized
# tensorflow graph input
x = tf.placeholder(tf.float64, (None, 32, 32, 3))
x = tf.cast(x, tf.float32)
y = tf.placeholder(tf.uint8, (None))
one_hot_y = tf.one_hot(y, n_classes)
keep_prob = tf.placeholder(tf.float32)
# parameters
rate = 0.001
EPOCHS = 15
BATCH_SIZE = 4096
display_step = 2
save_step = 5

# do train?
do_train = 1
```

```python
# select device to be used
device_type = "/gpu:0"
### LeNet with batch normalization / He initialization

def LeNet_he(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x32.
    conv1_w = tf.Variable(tf.truncated_normal(shape=(5,5,3,32), mean=mu, stddev=np.sqrt(2/(5*5*3))))
    conv1_b = tf.Variable(tf.zeros(32))
    conv1 = tf.nn.conv2d(x, conv1_w, strides=[1,1,1,1], padding='VALID') + conv1_b

    # batch normalization
    mean_, var_ = tf.nn.moments(conv1, [0,1,2])
    conv1 = tf.nn.batch_normalization(conv1, mean_, var_, 0, 1, 0.0001)

    # Activation.
    conv1 = tf.nn.relu(conv1)

    # Pooling. Input = 28x28x32. Output = 14x14x32.
    conv1 = tf.nn.max_pool(conv1, ksize=[1,2,2,1],strides=[1,2,2,1], padding='VALID')
```

```
# Layer 2: Convolutional. Output = 10x10x64.
conv2_w = tf.Variable(tf.truncated_normal(shape=(5,5,32,64), mean=mu, stddev=np.sqrt(2/(5*5*32))))
conv2_b = tf.Variable(tf.zeros(64))
conv2 = tf.nn.conv2d(conv1, conv2_w, strides=[1,1,1,1], padding='VALID') + conv2_b

# batch normalization
mean_, var_ = tf.nn.moments(conv2, [0,1,2])
conv2 = tf.nn.batch_normalization(conv2, mean_, var_, 0, 1, 0.0001)

# Activation.
conv2 = tf.nn.relu(conv2)

# Pooling. Input = 10x10x64. Output = 5x5x64.
conv2 = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')

# Flatten. Input = 5x5x64. Output = 1600.
fc0 = flatten(conv2)

# Layer 3: Fully Connected. Input = 1600. Output = 120.
fc1_w = tf.Variable(tf.truncated_normal(shape=(1600,120), mean=mu, stddev=np.sqrt(2/(1600))))
fc1_b = tf.Variable(tf.zeros(120))
fc1 = tf.matmul(fc0, fc1_w) + fc1_b
```

```
    # batch normalization
    mean_, var_ = tf.nn.moments(fc1, axes=[0])
    fc1 = tf.nn.batch_normalization(fc1, mean_, var_, 0, 1, 0.0001)

    # Activation.
    fc1 = tf.nn.relu(fc1)

    # Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_w = tf.Variable(tf.truncated_normal(shape=(120,84), mean=mu,
stddev=np.sqrt(2/120)))
    fc2_b = tf.Variable(tf.zeros(84))
    fc2 = tf.matmul(fc1, fc2_w) + fc2_b

    # batch normalization
    mean_, var_ = tf.nn.moments(fc2, axes=[0])
    fc2 = tf.nn.batch_normalization(fc2, mean_, var_, 0, 1, 0.0001)

    # Activation.
    fc2 = tf.nn.relu(fc2)

    print('LeNet w/ He initialziation is ready')
LeNet w/ He initialziation is ready
with tf.device(device_type):

#    logits = LeNet(x, keep_prob)
    logits = LeNet_he(x)
```

```
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=one_hot_y)
    loss_operation = tf.reduce_mean(cross_entropy)
    optimizer = tf.train.AdamOptimizer(learning_rate = rate)
    training_operation = optimizer.minimize(loss_operation)
    correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
    accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()

  # training
    if do_train == 1:
        # initialize TensorFlow variables
        sess.run(tf.global_variables_initializer())
        num_examples = len(X_train_)
        print("Training...")
        print()

        # epoch
        for epoch in range(EPOCHS):
            avg_loss = 0.
            total_batch = int(num_examples/BATCH_SIZE)
            X_train_, y_train = shuffle(X_train_, y_train)
            for offset in range(0, num_examples, BATCH_SIZE):
                end = offset + BATCH_SIZE
                batch_x, batch_y = X_train_[offset:end], y_train[offset:end]
                sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})
```

```
            avg_loss += sess.run(loss_operation, feed_dict={x: batch_x, y: batch_y})/total_batch
        if epoch % display_step == 0:
            train_acc = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y})
            valid_acc = sess.run(accuracy_operation, feed_dict={x: X_valid_, y: y_valid})
            print("Epoch: %03d/%03d, loss: %.9f, train acc: %.3f, valid acc: %.3f"
                  % (epoch + 1, EPOCHS, avg_loss, train_acc, valid_acc))
    if epoch % save_step == 0:
            saver.save(sess, "nets/traffic_sign_lenet-" + str(epoch))

    # calculate training accuracy
    batch_size_for_cal = 10000
    n_train_right = 0
    offset = 0
    tstep = np.floor(X_train_.shape[0]/10000)
    for t in range(tstep.astype(int)):
        if X_train_.shape[0] - (batch_size_for_cal + offset) < 0:
            batch_size_for_cal = X_train_.shape[0] - offset
        n_train_right += sess.run(accuracy_operation,
                                  feed_dict={x: X_train_[offset:offset+batch_size_for_cal],
                                             y: y_train[offset:offset+batch_size_for_cal]}) * batch_size_for_cal

    train_acc = n_train_right/X_train_.shape[0]
```

```
        # validation and test accuracy
        valid_acc = sess.run(accuracy_operation, feed_dict={x: X_valid_, y: y_valid})
        test_acc = sess.run(accuracy_operation, feed_dict={x: X_test_, y:y_test})
        print("Train accuracy: %.3f" % (train_acc))
        print("Validation accuracy: %.3f" % (valid_acc))
        print("Test accuracy: %.3f" % (test_acc))
with open('LeNet_He_BatchNorm.csv', 'r') as csvfile:
    data = []
    reader = csv.reader(csvfile)
    for row in reader:
        data.append(row)
plt.xlabel('epoch')
plt.grid()
plt.savefig('./result_loss.png')
plt.show()
plt.figure()
plt.plot(data[1:,0], data[1:,2], label='train accuracy')
plt.plot(data[1:,0], data[1:,3], label='test accuracy')
plt.title('accuracy')
plt.xlabel('epoch')
plt.legend(loc='best')
plt.grid()
plt.savefig('./result_acc.png')
plt.show()
```

REFERENCES

[1] Berghoff, C., Bielik, P., Neu, M., Tsankov, P. and Twickel, A.V., 2021, June. “Robustness testing of AI systems: A case study for traffic sign recognition”. In IFIP International Conference on Artificial Intelligence Applications and Innovations (pp. 256-267). Springer, Cham.

[2] Cao, J., Zhang, J. and Jin, X., 2021. “A traffic-sign detection algorithm based on improved sparse r-cnn”. IEEE Access, 9, pp.122774-122788.

[3] Çetinkaya, M. and Acarman, T., 2021, May. “Traffic Sign Detection by Image Preprocessing and Deep Learning”. In 2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS) (pp. 1165-1170). IEEE.

[4] https://valientemott.com/blog/blog-self-driving-cars-pros-and-cons/.

[5] Jin, Y., Fu, Y., Wang, W., Guo, J., Ren, C. and Xiang, X., 2020, “Multi-feature fusion and enhancement single shot detector for traffic sign recognition”, *IEEE Access*, *8*, pp.38931-38940.

[6] J. Zhang, Z. Xie, J. Sun, X. Zou and J. Wang, 2020,"A Cascaded R-CNN With Multiscale Attention and Imbalanced Samples for Traffic Sign Detection," in IEEE Access, vol. 8, pp. 29742-29754

[7] Kanagaraj, N., Hicks, D., Goyal, A., Tiwari, S. and Singh, G., 2021, "Deep learning using computer vision in self driving cars for lane and traffic sign detection," in International Journal of System Assurance Engineering and Management, vol. 12 ,no. 6, pp.1011-1025.

[8]KITTI Road/Lane Detection Evaluation 2013 Dataset, http://www.cvlibs.net/datasets/kitti/eval_road.php .

[9] Kortli, Y., Gabsi, S., Voon, L.F.L.Y., Jridi, M., Merzougui, M. and Atri, M., 2022. “Deep embedded hybrid CNN-LSTM network for lane detection on

NVIDIA Jetson Xavier NX. Knowledge-Based Systems”, p.107941.

[10]Liu, Z., Du, J., Tian, F. and Wen, J., 2019, “MR-CNN: A multi-scale region-based convolutional neural network for small traffic sign recognition”, *IEEE Access*, *7*, pp.57120-57128.

[11]Liu, Z., Shen, C., Qi, M. and Fan, X., 2020, “Sadanet: Integrating scale-aware and domain adaptive for traffic sign detection.”, *IEEE Access*, *8*, pp.77920-77933.

[12]M. Marzougui, A. Alasiry, Y. Kortli and J. Baili, 2020,"A Lane Tracking Method Based on Progressive Probabilistic Hough Transform," in IEEE Access, vol. 8, pp. 84893-84905

[13] P. Lu, C. Cui, S. Xu, H. Peng and F. Wang, 2021, "SUPER: A Novel Lane Detection System," in IEEE Transactions on Intelligent Vehicles, vol. 6, no. 3, pp. 583-593.

[14] Tang, J., Li, S. and Liu, P., 2021. “A review of lane detection methods based on deep learning. Pattern Recognition”, 111, p.107623.

[15]The German Traffic Sign Recognition Benchmark dataset, https://benchmark.ini.rub.de/gtsrb_news.html.

[16] Y. Zhang, J. Wang, X. Wang and J. M. Dolan, 2018, "Road-Segmentation-Based Curb Detection Method for Self-Driving via a 3D-LiDAR Sensor," in IEEE Transactions on Intelligent Transportation Systems, vol. 19, no. 12, pp. 3981-3991.

[17] Y. Zhang, Z. Lu, D. Ma, J. -H. Xue and Q. Liao, 2021,"Ripple-GAN: Lane Line Detection With Ripple Lane Line Detection Network and Wasserstein GAN," in IEEE Transactions on Intelligent Transportation Systems, vol. 22, no. 3, pp. 1532-1542.

[18] Zou, Q., Jiang, H., Dai, Q., Yue, Y., Chen, L. and Wang, Q., 2019. “Robust lane detection from continuous driving scenes using deep neural networks”. IEEE transactions on vehicular technology, 69(1), pp.41-54.