##### DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING ANDHRA UNIVERSITY COLLEGE OF ENGINEERING

**ANDHRA UNIVERSITY**

**Accredited by NAAC with ‘A’ grade ISO 9001: 2015 certified Visakhapatnam-530003**

**2022-2023.**

**BACHELOR OF TECHNOLOGY**

**IN**

**ELECTRONICS AND COMMUNICATION ENGINEERING**

**Dusanapudi Veera Venkata Rohit : 319506507015**

**Kotharu Lakshmi Sandeep : 319506507032**

**Sripana Leela Sri : 319506507034**

Under the Guidance

of

**Prof. V. MALLESWARA RAO, ME, PhD**

***A Thesis submitted in partial fulfilment of the requirement for the***

***Award of degree of***

**STEERING CONTROL USING AI FOR SELF-DRIVING CARS**

**SUBMITTED BY**

**BONAFIDE CERTIFICATE**

This is to certify that the project report entitled “**STEERING CONTROL USING AI FOR SELF-DRIVING CARS**” is a bonafide record work done by **KOTHARU LAKSHMI SANDEEP, DUSANAPUDI VEERA VENKATA ROHIT, SRIPANA LEELA SRI** bearing the **Reg No: 319506507032, 319506507015** and **319506507034**. under the guidance of **Prof. V. MALLESWARA RAO** during the academic year 2022-2023, submitted in partial requirements for the award of the degree of Bachelor of Technology in Electronics and Communication Engineering of Andhra University of College of Engineering, Andhra University, Visakhapatnam.

**PROJECT GUIDE HEAD OF THE DEPARTMENT**

**Prof. V. MALLESWARA RAO, M.E., Ph.D. Prof. P. RAJESH KUMAR, M.E., Ph.D.**

Department Of Department Of

Electronics and Communication Engineering Electronics and Communication Engineering

Andhra University College of Engineering Andhra University College of Engineering

Visakhapatnam Visakhapatnam

##### DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING ANDHRA UNIVERSITY COLLEGE OF ENGINEERING

**ANDHRA UNIVERSITY**

**VISAKHAPATNAM**

**ACKNOWLEDGEMENT**

We are thankful and would like to express our sincere gratitude to our project guide **Prof. V.** **MALLESWARA RAO** Adjunct Professor, Department of Electronics and Communications Engineering for spending his valuable time in analysing the project at every stage and for his assistance in bringing forth our project has been proved invaluable. We mention our deep appreciation for him being there for us and guiding us through every step of the project.

We are sincerely thankful to **Prof. P. RAJESH KUMAR**, Head of the department, Electronics and Communication Engineering for his kind cooperation in the completion of work

We are sincerely thankful to **Prof. G. SASIBHUSHANA Rao**, Principal of Andhra University College of Engineering, Andhra University, Visakhapatnam for his involvement, providing required resources and his kind cooperation in the completion of work.

We are sincerely thankful to **Prof. P.V.G.D. Prasad Reddy**, Vice Chancellor of Andhra University, Visakhapatnam for his support and extending his kind cooperation for the completion of work.

We are thankful to the faculty members **Dr. G. RAJESWARA RAO**, **Prof. P. V. SRIDEVI**, **Prof. P. MALLIKARJUNA RAO**, **Prof. M.S. ANURADHA**, **Dr. S. ARUNA** and other teaching and non-teaching staff for their cooperation and encouragement in the completion of the project.

|  |  |
| --- | --- |
| **Regards,** |  |
| **SRIPANA LEELA SRI** | **319506507034** |
| **KOTHARU LAKSHMI SANDEEP** | **319506507032** |
| **DUSANAPUDI VEERA VENKATA ROHIT** | **319506507015** |

**DECLARATION**

We hereby declare that the project entitled “ **STEERING CONTROL USING AI FOR SELF-DRIVING CARS**” is an original work done at Andhra University College of Engineering, Andhra University, Visakhapatnam, submitted in partial fulfilment of the requirements for the award of **Bachelor of Technology in Electronics and Communications Engineering.** We declare that this project has not been submitted by us for any degree/diploma/anywhere else.

Sripana Leela Sri

(319506507034)

Kotharu Lakshmi Sandeep

(319506507032)

Dusanapudi Veera Venkata Rohit

(319506507015)

**STEERING CONTROL USING AI FOR SELF-DRIVING CARS**

The Advance in technology over recent years has led major Automobile Manufactures to invest in making their cars technologically more furnished. A Major push has been in the area of making cars autonomous and self-driving. Autonomous cars would not only make driving simpler but safer.

This project will aim to develop a predictive model for steering angle of self-driving cars. This model will use Visual data from cameras mounted on the car and perform the prediction of the steering angle based on this data. The Accuracy and Reliability of the model will be evaluated using various performance metrics.

This Model will be developed using Convolutional neural network (CNN) to map raw pixels from a single front Facing camera directly to steering commands. The data used in the training of the model will be collected from real-world driving scenarios and will include both normal and challenging driving conditions. With minimum training data from the humans the system learns to drive in traffic on local roads and on highways.

The system automatically learns the internal representations of the necessary processing steps such as detecting useful road features with only the human steering angle as the training signal. We never explicitly Trained it to detect, for example, the outline of roads. The results of the evaluation will be used to assess the performance of the model and identify areas for improvement. The findings of this project will contribute to the development of self-driving cars and advance the field of autonomous driving.

Overall, this project will demonstrate the feasibility of using CNNs for the prediction of steering angles with an accuracy of 96.38% in self-driving cars and highlight the importance of developing accurate predictive models for the safe and reliable operation of autonomous vehicles.

**ABSTRACT**

**ABBREVATIONS**

**ML :** Machine Learning

**DL** : Deep Learning

**SVM** : Support Vector Machine

**ANN** : Artificial Neural Network

**DT** : Decision Tree

**RF** : Random Forest

**CNN** : Convolution Neural Network

**CV** : Computer Vision

**LIST OF FIGURES**

**Page No**

**Figure 1**: Self driving car 3

**Figure 2**: DARPA urban challenge 4

**Figure 3**: Stanley car 5

**Figure 4**: Overview of autonomous systems 6

**Figure 5**: Google’s self-driving car 8

**Figure 6**: Evolution of deep models 14

**Figure 7**: General neural network 19

**Figure 8**: An example of CNN architecture for image classification 23

**Figure 9**: Schematic diagram illustrating interconnections between layers 24

**Figure 10**: Conceptual model of CNN 25

**Figure 11**: The primary calculations executed at each step of convolution layer 27

**Figure 12**: Three types of pooling operations 28

**Figure 13**: Fully connected layer 29

**Figure 14**: The architecture of LeNet 31

**Figure 15**: The architecture of AlexNet 31

**Figure 16**: Tensors and graphs 40

**Figure 17**: Camera setup for data collection 44

**Figure 18**: Representative training data samples 45

Showing various road textures and marker conditions

**Figure 19**: High-level view of the data collection 46

**Figure 20**: Training of neural network 47

**Figure 21**: the trained neural network is used to generate steering 47

commands from a single front facing camera

**Figure 22**: NVIDIA PX 2 Drive 48

**Figure 23**: CNN Architecture 51

**Figure 24**: An image from Dataset 1 53

**Figure 25**: Steering Wheel 53

**Figure 26**: Predicted Angle 53

**LIST OF TABLES**

**Page No**

**TABLE 1** : Machine Learning vs Deep Learning  **16**

**TABLE 2** : Basic CNN Architectures **30-31**

**LIST OF CONTENTS**

**lkjkjgjikjikjhijkhjjjj**

**INDEX Page No**

**CHAPTERS 1: INTRODUCTION 1 - 8**

1.1 Autonomous Driving

1.2 History and background information

1.3 Artificial intelligence in self-driving cars

1.4 Overview of autonomous driving systems

1.4.1 Traditional autonomous driving system

1.4.2 End-to-end autonomous driving system

1.5 Motivation

1.6 Objective

**CHAPTER 2: LITERATURE REVIEW 9 - 11**

2.1 Literature Review

2.2 Recent Studies

**CHAPTER 3: DEEP LEARNING 12 - 17**

3.1 Introduction

3.2 Evolution of Deep Learning

3.3 Deep Learning Approaches

3.3.1 Supervised Learning

3.3.2 Unsupervised Learning

3.3.3 Reinforcement Learning

3.3.4 Hybrid Learning

3.4 When to apply deep learning

3.5 Machine Learning Vs Deep Learning

3.6 Why deep learning

**CHAPTER 4: NEURAL NETWORKS 18 - 32**

4.1 Introduction

4.2 History of Neural Networks

4.3 types of Neural Networks

4.4 Detailed comparison of the accuracies of the different algorithms used in the project:

4.5 Convolution neural networks

4.6 Foundation of Convolution Neural Network

4.7 Concepts of Convolutional Neural Network

4.8 CNN Layers

4.9 CNN Architectures

4.10 Data Augmentation techniques

**CHAPTER 5: OpenCV & TENSORFLOW 33 - 42**

5.1 Introduction

5.2 Libraries in OpenCV

5.3 TensorFlow introduction

5.4 TensorFlow APIs

5.5 Features of TensorFlow

5.6 Why TensorFlow is popular

5.7 History of TensorFlow

5.8 How TensorFlow Works

5.9 TensorFlow Components

5.10 Where can TensorFlow run

5.11 How calculations work in TensorFlow

**CHAPTER 6: DESIGN METHODOLOGY 43 - 54**

6.1 Data Collection

6.2 Dataset

6.3 Overview of DAVE 2 System

6.4 Why NVIDIA PX 2 Drive

6.5 CAN protocol

6.6 How model is Trained

6.7 Calculations Performed in layers

6.8 Results & Analysis

6.9 Loss Factor

**CONCLUSION 55**

**FUTUTRE WORK 56**

**REFERENCES 57**

**APPENDIX 58 - 63**

**CHAPTER – 1**

**INTRODUCTION**

The push for self-driving vehicles has increased dramatically over the last few years after the success of DARPA Grand challenge. An autonomous vehicle works seamlessly when multiple components come together and work in synergy. The most important parts are the various sensors and the AI software powering the vehicle. One of the most critical functions of the AI is to predict the steering angle of the vehicle for the stretch of road lying immediately in front of it and accordingly even steer the car. Increase the computational capabilities over the years allow us to train deep neural networks to become the “brain” of these cars which then understand the surroundings of the car and make navigation decisions in real time.

Udacity held a series of challenges to create an open-source self-driving car. They released a data set of images taken while a car was being manually driven, annotated with the corresponding steering angle Applied by the human driver. The goal of one of the challenges was to find a model that, given an image taken while driving, will minimize the RMSE (root mean square error) between what the model predicts and the actual steering angle produced by a human driver.

In this project, we explore a variety of methods including deep convolutional neural networks, models based on pre-trained network to predict the steering angle values. Predicting the steering angles is one of the most important parts in self-driving cars.

CNNs have revolutionized pattern recognition. Prior to the widespread adoption of CNN’s, most pattern recognition tasks were performed using an initial stage of hand-crafted feature extraction followed by a Classifier. The breakthrough of CNNs is that features are learned automatically from training examples.

The CNN approach is especially powerful in image recognition tasks because the convolution operation captures the 2D nature of the images. Also, by using the convolution kernels to scan an entire image Relatively few parameters need to be learned compared to the total number of operations.

While CNNs with learned features have been in commercial use for over twenty years, their adoption has exploded in the last few years because of two recent developments. First, large, labelled data sets such as the Large-Scale Visual Recognition Challenge (ILSVRC) have become available for training and validation. Second, CNN learning algorithms have been implemented on the massively parallel graphics processing units (GPUs) which tremendously accelerate learning and inference.

Research on autonomous vehicle navigation was pioneered by Pomerleau (1989) when he built the autonomous land vehicle in a neural network. The model comprised of a fully connected network which Would be considered a very basic version if compared to the large models in use today. Though the network could be applied to only a few simple scenarios with minimum obstacles, recently an NVIDIA team consisting Of Bojarski, Del Testa et al carried out a study to design an end-to-end learning of self-driving cars. They trained a convolutional neural network (CNN) to map raw pixels from a single front facing camera directly to steering commands. They found that this method was surprisingly accurate, without much training from humans the model managed to learn to drive in traffic on local roads some of which had lane markings While some did not as well on highways.

Overall, this framework was successful in relatively simple real-World scenarios, such as highway lane-following and driving in flat, obstacle-free course.

1. **INTRODUCTION**

**1.1 Autonomous Driving**

The rapid development of the technologies in computer vision and machine learning has enabled researchers and industry leaders to make significant progress in achieving autonomous driving. Today, autonomous driving technology is making a prominent appearance in our society.

Vehicles equipped with advanced driver assistance systems (ADAS) can accomplish autonomous driving in several scenarios such as highway driving. These technologies aim to reduce the number and severity of road accidents. Every year, approximately 1.35 million people lose their lives in automobile accidents, and up to 50 million people suffer accident-related iǌuries (Organization, 2019). Ninety-four percent of serious crashes are due to risky driving and to errors people make while behind the wheel (highway traffic safety administration, 2019). A self-driving revolution that reduces traffic accidents has the potential to positively impact the lives of millions of people. Furthermore, once fully autonomous driving is achieved, we no longer have to keep our hands on the wheel or our eyes on the road, which would free us to read or get work done while traveling. The average American spends nearly 300 hours in their car each year (Association, 2019), which is over seven full work weeks. That is a ton of potential productivity lost. Autonomous vehicles can eliminate or at least mitigate these issues.

Googles vast computing resources are crucial to the technology used in self-driving cars. Googles self-driving cars memorize the road infrastructure in minute detail. They use computerized maps to determine where to drive, and to anticipate road signs, traffic lights and roadblocks long before they are visible to the human eye. They use specialized lasers, radar, and cameras to analyse traffic at a speed faster than the human brain can process. And they leverage the cloud to share information at blazing speed. These self-driving cars have now travelled nearly 1.5 million miles on public highways in California and Nevada.

They drive anywhere a car can legally drive. According to Sebastian Thrun, “I am confident that our self-driving cars will transform mobility. By this I mean they will affect all aspects of moving people and things around and result in a fundamentally improved infrastructure.

**Figure 1**. Self-Driving Car

The DARPA Urban Challenge was held on November 3, 2007, at the former George AFB in Victorville, Calif. Building on the success of the 2004 and 2005 Grand Challenges, this event required teams to build an autonomous vehicle capable of driving in traffic, performing complex maneuvers such as merging, passing, parking, and negotiating intersections. As the day wore on, it became apparent to all that this race was going to have finishers. At 1:43 pm, “Boss”, the entry of the Carnegie Mellon Team, Tartan Racing, crossed the finish line first with a run time of just over four hours. Nineteen minutes later, Stanford University’s entry, “Junior,” crossed the finish line. It was a scene that would be repeated four more times as six robotic vehicles eventually crossed the finish line, an astounding feat for the teams and proving to the world that autonomous urban driving could become a reality. This event was ground breaking as the first-time autonomous vehicles have interacted with both manned and unmanned vehicle traffic in an urban environment.

Since the 1980s, industrial leaders and researchers have been working on developing a fully autonomous vehicle that is comfortable, reliable, and safe for high-speed driving in the real world. Road tests and self-driving competitions held around the world identify shortcomings and difficulties in both software and hardware and allow autonomous driving technologies to be rapidly improved. Although some known problems remain unresolved, more and more vehicles with a certain level of autonomous driving ability are running on the road. In 1995, “No Hands Across America” was introduced as one of the first long-distance road tests for autonomous driving (Jochem and Pomerleau, 1995). A trained neural network was used to steer a vehicle driving across the United States while human drivers controlled its acceleration and braking. This was also one of the earliest types of research works to demonstrate end-to-end learning for autonomous lane keeping.

The next major competition called DARPA Grand Challenges was initiated by the Defense Advanced Research Projects Agency (DARPA) in 2003 (Rouff and Hinchey, 2011). It required autonomous vehicles to drive in off-road environments without the aid of road markings. Around the same time, the DAVE project was introduced to demonstrate an end-to-end learning system for off-road vehicle control using only visual input (Muller et al., 2006). In this project, a convolutional neural network was trained to predict steering angles based on images from left and right cameras. After the DARPA Grand Challenges competition, researchers started to tackle the challenges of urban driving in complex environments with dense traffic. Since road tests for autonomous urban driving were not permitted at that time, they faced considerable difficulty in addressing the challenges and making evaluations in real-world environments.

In 2007, the DARPA Urban Challenge was held to provide a real urban driving environment, which included an intersection and simulated highway on-ramp (Rouff and Hinchey, 2011). This competition provided opportunities for researchers to assess the capabilities and limits of autonomous driving in complex urban environments.

**Figure 2 :** DARPA Urban Challenge



**1.2 History and background information**

Besides other traditional sensors, this car had five laser range finders for measuring cross-sections of the terrain ahead up to 25m in front of the vehicle, a color camera for long-range road perception, and two 24 GHz RADAR sensors for long range detection of large obstacles. Despite winning the challenge, it left opens a few important problems like adapting to dynamic environments from a given static map or the ability to differentiate between objects with subtle differences. One of the important results observed from this race and highly relevant to this thesis research was the fact that during 4.7% of the challenge, the GPS reported 60cm error or more. This highlighted the importance of online mapping and path planning in the race. It also proved that a system solely dependent on GPS coordinates for navigation in self-driving cars is not sufficient, as the error tolerance for autonomous vehicles is around 10cm. The real time update of the global map based on the local environment helped Stanley to eliminate this problem in most of the cases.

The 2005 DARPA Grand Challenge was conducted on a desert track. Stanley won this challenge, but a lot of new challenges were foreseen from the results. The next challenge conducted by DARPA was in an urban environment. Stanford introduces the successor to Stanley named “Junior.” Junior was equipped with five laser rangefinders, a GPS-aided inertial navigation system and five radars as its environment perception sensors. The vehicle had an obstacle detection range of up to 120 meters and the ability to attain a maximum velocity of 30mph. A combination of planning, perception followed by control helped in its navigation. The software architecture primarily consisted of five modules - sensor interfaces, perception modules, navigation modules, drive-by-wire interface and global services. The perception modules were responsible for segmenting the environment data into moving vehicles and static obstacles. They also provided precision localization of the vehicle relative to the digital map of the environment. One of the major successes of this car was its successful completion of the journey with almost flawless static obstacle detection. However, it was found that the GPS -based inertial position computed by the software system was generally not accurate enough to perform reliable lane keeping without sensor feedback. Hence Junior used an error correction system for accurate localization with the help of feedback from other local sensors.

This fine-grained localization used two types of information: road reflectivity and curb-like obstacles. The reflectivity was sensed using the laser range finders, pointed towards the ground. The filter for localization was a 1-D histogram filter which was used to estimate the vehicles lateral offset relative to the provided GPS coordinates for the desired path to be followed. Based on the reflectivity and the curbs within the vehicle’s visibility, the filter would estimate the posterior distribution of any lateral offset.

**Figure 3 :** Stanley car

**1.3 Artificial intelligence in self-driving cars**

As of 2016, autonomous vehicles are no longer products of science fiction or just long-term visions of research and development departments of different corporations. The beginning of the success started with the self-driving car ”Stanley” which won the 2005 DARPA Grand Challenge

**Figure 4 :** Overview of autonomous driving systems

**1.4.1 Traditional autonomous driving system**

As a brief overview, a traditional autonomous driving system can be divided into five main components, shown in Fig 1.1, perception, localization and mapping, path planning, decision making, and vehicle control.

Similar to human vision, the perception component uses sensors to continuously scan and analyse its surrounding environment. This component usually consists of functions for obstacle detection and tracking, traffic sign recognition, and lane marker detection based on various sensor inputs. The localization and mapping component calculates the global and local locations of the vehicle and also maps the environment based on sensor data. Path planning uses perception and localization information to calculate possible safe and ideal routes for the vehicle to drive.

The decision-making component is designed to generate the optimal path based on the available routes, the vehicle’s state, and environmental information. Finally, the vehicle control module determines the driving commands, such as steering angle, acceleration, etc., that drives the vehicle along the determined optimal route.

Vehicle

Control

Decision

Making

Path

Planning

Mapping

Environment

Perception

A complete autonomous driving system can be described as an integration of already established systems of adaptive cruise control, parking assistance, and autopilots into a unified function that uses artificial intelligence (AI) and machine learning to adapt its driving behaviour based on driver preferences and data from the related safety and connectivity functions.

To date, most of these autonomous driving systems can be categorized into two main classes

1. Traditional autonomous Driving systems
2. End to End autonomous Driving Systems

**1.4 Overview of autonomous driving systems**

In a similar way like reinforcement learning it favoured offsets for which lane marker reflectivity patterns aligned with the lane markers or the road side from the supplied coordinates of the path. It also negated offsets for which an observed curb would reach into the driving corridor of the assumed coordinates. As a result, at any point in time the vehicle estimated a fine-grained offset to the measured location by the GPS-based system. A precision or lateral offset of one meter was common in the challenge. Without this error correction system, the car would have gone off the road or often hit a curb. It was observed that velocity estimates from the pose estimation system were much more stable than the position estimates, even when GPS feedback was not strong. X and Y velocities were particularly resistant to jumps because they were partially observed by wheel odometry.

**1.4.2 End-to-end autonomous driving system**

Although promising progress has been made in developing the traditional autonomous driving system, there are still many challenges towards building a fully autonomous driving vehicle.

However, there are several obvious limitations to the use of HD maps. First, the static maps are quite expensive to build and to keep current with dynamic environments that change over time. Second, the reliance of autonomous vehicles on pre-built maps limits their capability to react and adapt to new situations such as construction zones. These drawbacks have inspired research into the end-to-end learning approach for autonomous driving that does not require manual decomposition of the autonomous driving system and detailed maps.

In 1989, Autonomous Land Vehicle in a Neural Network (ALVINN), a three-layer neural network trained for the task of lane following, drove a retrofitted Army ambulance around Carnegie Mellon University under controlled field conditions without any human intervention.

ALVINN is one of the first examples of an autonomous vehicle using the end-to-end learning approach

The ALVINN net featured two kinds of sensory inputs:

1) a 30x32 image streamed from a camera mounted on the top of the vehicle

2) 8x32 image encoding range information captured by a laser range finder

The output layer of the ALVINN network could be divided into two groups of units. The first group, consisting of one unit, indicated whether the texture of the road in the current image was lighter or darker than the non-road. During testing, this unit was also recursively sent to the network’s input layer. The second group, consisting of 45 units, represented the turning curvature along the direction that the vehicle should travel in order to remain in the centre of the road. Activation of the middle part of the units represented straight driving, while activation of the left or right units represented left or right turns. In order to convert the network’s output active levels into a steering direction, a Gaussian curve with a fixed width was used to fit the output units. The peak of the best fit Gaussian determined the vehicle’s steering.

The ALVINN net inspired our work in many ways. It demonstrated that an end-to-end trained neural network could successfully drive a vehicle along a road. It also showed the importance of having sufficient variability in the training data set to cover different driving conditions. In this chapter, we will introduce an end-to-end learning method that trains a neural network to steer a vehicle for lane following

We define end-to-end autonomous driving as a single, self-contained driving system that carries out all processes automatically, from mapping based on sensory input, such as a font-facing camera, to the actions necessary for driving, such as steering, braking and acceleration. An end-to-end autonomous driving system is often designed to learn from expert demonstrations rather than depend upon manually-designed tasks and modules.

Although the end-to-end learning approach simplifies self-driving systems, it is challenging to train a model that encompasses everything from mapping very high-dimension, pixel-level sensor inputs to controlling low-dimension continuous control signals.

Two examples include improvement in mobility and use of efficient parking. Take todays cities they are full of parked cars. It is estimated, that the average car is immobile 96 percent of its lifetime. This situation leads to a world full of underused cars and occupied parking spaces. Self-driving cars will enable car sharing even in spread-out suburbs. A car will come to you just when you need it. And when you are done with it, the car will just drive away, so you will not even have to look for parking. Self-driving cars can also change the way we use our highways. The European Union has recently started a program to develop technologies for vehicle platoons on public highways.

1. To improve the safety and reliability of self-driving cars by ensuring they can accurately navigate roads and avoid accidents.
2. To explore the potential of deep learning algorithms in the field of autonomous vehicles and contribute to advancements in this field.
   1. **Motivation**

Research and development of autonomous vehicles is becoming more and more popular in the automotive industry. It is believed that autonomous vehicles are the future for easy and efficient transportation that will make for safer, less congested roadways. In 2014, according to the Department of Transportation, besides the human toll of 32,000 deaths in the US and 2.31M people injured, the costs are $1 trillion! In recent years, nearly all states have passed laws prohibiting the use of handheld devices while driving. Nevada took a different approach. In a first for any state, it passed a law that legalizes texting, provided one does so in a self-driving autonomous car. This places Nevada at the forefront of innovation.

Googles vast computing resources are crucial to the technology used in self-driving cars. Googles self-driving cars memorize the road infrastructure in minute detail. They use computerized maps to determine where to drive, and to anticipate road signs, traffic lights and roadblocks long before they are visible to the human eye. They use specialized lasers, radar, and cameras to analyse traffic at a speed faster than the human brain can process. And they leverage the cloud to share information at blazing speed. These self-driving cars have now travelled nearly 1.5 million miles on public highways in California and Nevada. They have driven from San Francisco to Los Angeles and around Lake Tahoe, and have even descended crooked Lombard Street in San Francisco. They drive anywhere a car can legally drive. According to Sebastian Thrun, “I am confident that our self-driving cars will transform mobility. By this I mean they will affect all aspects of moving people and things around and result in a fundamentally improved infrastructure.”

**Figure 5** : Google’s Self Driving car

**1.6 Objective**

**CHAPTER – 2**

**LITERATURE REVIEW**

**2. LITERATURE REVIEW**

**2.1 Literature Review**

This project that aims to develop a deep learning model for self-driving cars. The project utilizes convolutional neural networks (CNN) to predict steering angles from camera images. The CNN model is built using TensorFlow, an open-source machine learning framework developed by Google.

The project's goal is to train a model that can predict steering angles with high accuracy, allowing a self-driving car to navigate safely on the road. The training data used in the project comes from the Udacity Self-Driving Car Engineer Nanodegree Program, which includes real-world driving data captured from a camera mounted on a car.

The project's initial release in 2016 received widespread attention from the media and the deep learning community. The CNN model achieved impressive results, outperforming many traditional computers vision algorithms in terms of accuracy and robustness.

In this Literature review some of the recent studies and models developed for self-driving cars are discussed

**MODELS**

**Waymo**: Waymo is a self-driving car developed by Alphabet's subsidiary, Google. The car uses a combination of technologies, including LIDAR, radar, and cameras, to create a 3D map of its surroundings. Waymo also uses deep learning algorithms to analyse the data it collects and make decisions based on that analysis.

**Tesla Autopilot**: Tesla's Autopilot is a semi-autonomous driving system that uses a suite of sensors, including cameras, ultrasonic sensors, and radar, to enable features like automatic lane changing, summoning the car, and navigating on the highway. Tesla's Autopilot also uses deep learning algorithms to improve its performance over time.

**Cruise AV**: The Cruise AV is a self-driving car developed by General Motors. It uses a combination of LIDAR, radar, and cameras to create a 3D map of its surroundings. The car also uses a combination of deep learning algorithms and rule-based systems to make decisions based on the data it collects.

**Mobileye**: Mobileye is an Israeli-based company that develops vision-based advanced driver assistance systems (ADAS). Their technology uses a camera-based system that can identify objects and read road markings to provide advanced safety features such as lane departure warning and pedestrian detection. Mobileye has also developed a self-driving car platform called Mobileye Drive that uses a combination of cameras, radar, and LIDAR to enable autonomous driving.

**2.2 Recent Studies**

Research on autonomous vehicle navigation was pioneered by Pomerleau(1989) [6] when he built the Autonomous Land Vehicle in a Neural Network. The model comprised of a fully-connected network which would be considered a very basic version if compared to the large models in use today. Though the network could be applied to only a few simple scenarios with minimal obstacles this paper laid the foundation of end-to-end autonomous navigation. Recently, an NVIDIA team consisting of Bojarski, Del Testa et al [5]. carried out a study to design an end-to-end learning of self-driving cars. They trained a convolutional neural network(CNN) to map raw pixels from a single front facing camera directly to steering commands.

They found that this method was surprisingly accurate, without much training from humans the model managed to learn to drive in traffic on local roads some of which had lane markings while some did not as well as on highways. Overall, this framework was successful in relatively simple real-world scenarios, such as highway lane-following and driving in flat, obstacle-free courses.

The use of complicated neural network structures has increased in the process of classification of videos and object detection. These advancements are getting translated and transferred to challenges of autonomous driving.

Comma.ai [8] has proposed to learn a driving simulator that uses Generative Adversarial Networks (GANs) [3] and a Variational Auto-encoder (VAE) [2]. Their approach is able to keep predicting realistic looking video for several frames based on previous frames despite the transition model being optimized without a cost function in the pixel space. Moreover, deep reinforcement learning (RL) has also been applied to autonomous driving [7], [9]. RL has not been successful for automotive applications until some recent work, which shows the deep learning algorithm’s ability to learn good representations of the environment. Inspired by the success of deep reinforcement learning in learning of games, [7] has proposed a framework for autonomous driving using deep RL it is an end-to end Deep Reinforcement learning pipeline for autonomous driving which integrates RNNs to account for POMDP scenarios. The framework was tested for lane keep assist algorithm.

**CHAPTER – 3**

**DEEP LEARNING**

Deep learning techniques which implement deep neural networks became popular due to the increase of high-performance computing facility. Deep learning achieves higher power and flexibility due to its ability to process a large number of features when it deals with unstructured data. Deep learning algorithm passes the data through several layers; each layer is capable of extracting features progressively and passes it to the next layer. Initial layers extract low-level features, and succeeding layers combines features to form a complete representation. Section 2 gives an overview of the evolution of deep learning models. Section 3 provides a brief idea about the different learning approaches, such as supervised learning, unsupervised learning, and hybrid learning. Supervised learning uses labelled data to train the neural network. In supervised learning, the network uses unlabelled data and learns the recurring patterns. Hybrid learning combines supervised and unsupervised methods to get a better result. Deep learning can be implemented using different architectures such as architectures like Unsupervised Pre-trained Networks, Convolutional Neural Networks, Recurrent Neural Networks, and Recursive Neural Networks, which are described in section 4. Section 5 introduces various training methods and optimization techniques that help in achieving better results. Section 6 describes the frameworks which allow us to develop tools that offer a better programming environment. Despite the various challenges in deep learning applications, many exciting applications that may rule the world

The last five years starting from the year 2012 have ushered in a lot of success in the world of machine learning especially due to the boom in deep learning. Although it may seem that deep neural networks were invented very recently, they were conceived of in the 1980s. Although these early architectures were not in the exact structure that is present today, their underlying concept is very similar. Before diving into the detailed working mechanism of Convolutional Neural Networks (CNNs), revisiting their origin and why they became successful in the recent years can lead to a better understanding of deep learning. presented the first general, working learning algorithm for supervised deep feedforward multilayer perceptron. In 1971, described a deep network with 8 layers. It was trained on a computer identification system known as ”Alpha.” Other Deep Learning working architectures, especially those built from ANNs date back to 1980.

The architecture of this network was relatively simple compared to networks that are present today. It composed of alternate cells known as simple and complex cells in a sandwich type of architecture used for unsupervised learning. While the simple cells had modifiable parameters, the complex cells were used for pooling. Due to various constraints, one of them being limited processing power in the hardware, these networks didn’t perform quite as well as alternate techniques. Backpropagation, one of the fundamental concepts in learning a network, was first applied by Yann LeCun et al. to a deep neural network for the purpose of recognizing handwritten ZIP codes on mail for the US Postal Service. The input of the network consisted of normalized images of isolated digits. Despite being applied almost 20 years back, it produced excellent results with a 1% error rate for the specific application. Due to the hardware constraints, it wasn’t suitable for general use at the time. The time to train the full network took approximately 3 days. The first convolutional neural network with backpropagation as we know today was proposed in by Yann LeCun et al. in 1998 for document recognition.

This was a 6-layer network composed of three convolutional layers, two pooling layer (subsampling) and a fully connected layer in the end. The name of the network was LeNet-5. The detailed explanation of each layer in a convolutional neural network will be explained in the next section. Although, these kinds of networks were very successful in handling smaller size images or other problems like character or word recognition, it was thought until 2011, that these networks would not be able to handle larger more complex images and problems, hence the use of traditional methods like object detectors using hand-tuned features and classifiers

**3. INTRODUCTION TO DEEP LEARNING**

**3.1 Introduction**

First Generation of Artificial Neural networks (ANN) was composed of perceptron’s in neural layers, which were limited in computations. The second generation calculated the error rate and backpropagated the error. Restricted Boltzmann machine overcame the limitation of backpropagation, which made the learning easier. Then other networks are evolved eventually Figure.1 illustrates a timeline showing the evolution of deep models along with the traditional model. The performance of classifiers using deep learning improves on a large scale with an increased quantity of data when compared to traditional learning methods. Figure.2 depicts the performance of traditional machine learning algorithms and deep learning algorithm. The performance of traditional machine learning algorithms becomes stable when it reaches the threshold of training data whereas the deep learning upturns it’s performance with increased amount of data. Now a days deep learning is used in a lot many applications such as Google’s voice and image recognition, Netflix and Amazon’s recommendation engines, Apple’s Siri, automatic email and text replies, chatbots etc

**Figure 6 :** Evolution of Deep Models

Deep neural networks are composed of different approaches

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning
4. Hybrid Learning

Deep neural networks are successful in Supervised learning, Unsupervised learning, Reinforcement learning, as well as hybrid learning

**3.3 Deep Learning Approaches**

**3.2 Evolution of Deep Learning**

**3.3.1 Supervised Learning**

In supervised learning, the input variables represented as X are mapped to output variables represented as Y by using an algorithm to learn the mapping function.

Y = f(X) …... (1)

The aim of the learning algorithm is to approximate the mapping function to predict the output (Y) for a new input (X). The error from the predictions made during training can be used to correct the output. Learning can be stopped when all the inputs are trained to get the targeted output. Regression for solving regression problems, Support Vector machines used for classification, Random Forest for classification as well as regression problems.

**3.3.2 Unsupervised Learning**

In unsupervised learning, we have the input data only and no corresponding output to map. This learning aims to learn about data by modelling the distribution in data. Algorithms can be able to discover the exciting structure present in the data. Clustering problems and association problems use Unsupervised learning. The unsupervised learning algorithms such as K-means algorithm is used in clustering problems, Apriori algorithm is used in association problems.

**3.3.3 Reinforcement Learning**

Reinforcement learning uses a system of reward and punishment to train the algorithm. In this, the algorithm or an agent learns from its environment. The agent gets rewards for correct performance and penalty for incorrect performance. For example, consider the case of a self-driving car, the agent gets a reward for driving safely to destination and penalty for going off-road. Similarly, in the case of a program for playing chess, the reward state may be winning the game and the penalty for being checkmated. The agent tries to maximize the reward and minimize the penalty. In reinforcement learning, the algorithm is not told how to perform the learning; however, it works through the problem on its own.

**3.3.4 Hybrid Learning**

Hybrid learning refers to architectures that make use of generative (unsupervised) as well as discriminative (supervised) components. The combination of different architectures can be used to design a hybrid deep neural network. They are used for action recognition of humans using action bank features and are expected to produce much better results.

**3.4 When to apply deep learning**

Machine intelligence is useful in many situations which is equal or better than human experts in some cases, meaning that DL can be a solution to the following problems:

• Cases where human experts are not available.

• Cases where humans are unable to explain decisions made using their expertise (language  
 understanding, medical decisions, and speech recognition).

• Cases where the problem solution updates over time (price prediction, stock preference, weather prediction, and tracking).

• Cases where solutions require adaptation based on specific cases (personalization, biometrics).

• Cases where size of the problem is extremely large and exceeds our inadequate reasoning abilities (sentiment analysis, matching ads to Facebook, calculation webpage ranks)

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Machine Learning** | **Deep Learning** |
| **Data Dependency** | Although machine learning depends on the  huge amount of data, it can work with a  smaller amount of data. | Deep Learning algorithms highly depend on a large amount of data, so we need to feed a large amount of data for good performance. |
| **Execution time** | Machine learning algorithm takes less time to train the model than deep learning, but it takes a long-time duration to test the model. | Deep Learning takes a long execution time to train the model, but less time to test the model. |
| **Hardware Dependencies** | Since machine learning models do not need much amount of data, so they can work on low-end machines. | The deep learning model needs a huge amount of data to work efficiently, so they need GPU's and hence the high-end machine. |
| **Feature Engineering** | Machine learning models need a step of feature extraction by the expert, and then it proceeds further. | Deep learning is the enhanced version of machine learning, so it does not need to develop the feature extractor for each problem; instead, it tries to learn high-level features from the data on its own. |
| **Problem-solving approach** | To solve a given problem, the traditional ML model breaks the problem in sub-parts, and after solving each part, produces the final result. | The problem-solving approach of a deep learning model is different from the traditional ML model, as it takes input for a given problem, and produce the end result. Hence it follows the end-to-end approach. |
| **Interpretation of result** | The interpretation of the result for a given problem is easy. As when we work with machine learning, we can interpret the result easily, it means why this result occur, what was the process. | The interpretation of the result for a given problem is very difficult. As when we work with the deep learning model, we may get a better result for a given problem than the machine learning model, but we cannot Find why this particular outcome occurred,  and the reasoning. |
| **Type of data** | Machine learning models mostly require data in a structured form. | Deep Learning models can work with structured and unstructured data both as they rely on the layers of the Artificial neural network. |
| **Suitable for** | Machine learning models are suitable for solving simple or bit-complex problems. | Deep learning models are suitable for solving complex problems. |

**Table: 1**

**3.5 Machine Learning Vs Deep Learning**

**3.6 Why deep learning?**

Several performance features may answer this question, e.g.

1. **Universal Learning Approach**: Because DL has the ability to perform in approximately all application domains, it is sometimes referred to as universal learning.
2. **Robustness**: In general, precisely designed features are not required in DL techniques. Instead, the optimized features are learned in an automated fashion related to the task under consideration. Thus, robustness to the usual changes of the input data is attained.
3. **Generalization**: Different data types or different applications can use the same DL technique, an approach frequently referred to as transfer learning (TL) which explained in the latter section. Furthermore, it is a useful approach in problems where data is insufficient.
4. **Scalability**: DL is highly scalable. ResNet, which was invented by Microsoft, comprises 1202 layers and is frequently applied at a supercomputing scale. Lawrence Livermore National Laboratory (LLNL), a large enterprise working on evolving frameworks for networks, adopted a similar approach, where thousands of nodes can be implemented.

* Deep learning allows the model to learn features automatically from the data, reducing the need for manual feature engineering.
* Deep learning models can handle large amounts of data, making them suitable for complex tasks such as image and speech recognition.
* Deep learning models can be used for both supervised and unsupervised learning tasks.
* Deep learning models have shown state-of-the-art performance in many domains, including computer vision, natural language processing, and speech recognition.
* Deep learning models can learn from raw input data, eliminating the need for preprocessing steps such as feature scaling and normalization.
* Deep learning models can generalize well to unseen data, making them suitable for real-world applications.
* Deep learning models can learn hierarchical representations of data, allowing them to capture complex patterns in the data.
* Deep learning models can be trained using large-scale distributed computing systems, allowing for faster model training.
* Deep learning models can be used for transfer learning, where a pre-trained model is fine-tuned for a specific task, reducing the need for large amounts of labeled data.
* Deep learning models have the potential to revolutionize many industries, including healthcare, finance, and transportation, by enabling new applications and improving existing ones

**CHAPTER – 4**

**NEURAL NETWORKS**

**4. NEURAL NETWORKS**

**4.1. Introduction**

Neural networks rely on training data to learn and improve their accuracy over time. However, once these learning algorithms are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity. Tasks in speech recognition or image recognition can take minutes versus hours when compared to the manual identification by human experts. One of the most well-known neural networks is Google’s search algorithm.

**Neural networks** are artificial systems that were inspired by biological neural networks. These systems learn to perform tasks by being exposed to various datasets and examples without any task-specific rules. The idea is that the system generates identifying characteristics from the data they have been passed without being programmed with a pre-programmed understanding of these datasets. Neural networks are based on computational models for threshold logic. Threshold logic is a combination of algorithms and mathematics. Neural networks are based either on the study of the brain or on the application of neural networks to artificial intelligence. The work has led to improvements in finite automata theory. Components of a typical neural network involve neurons, connections which are known as synapses, weights, biases, propagation function, and a learning rule. Neurons will receive an input pj(t) from predecessor neurons that have an activation aj(t), threshold, an activation function f, and an output function fout. Connections consist of connections. weights and biases which rules how neuron transfers output to neuron. Propagation computes the input and outputs the output and sums the predecessor neurons function with the weight. The learning of neural network basically refers to the adjustment in the free parameters i.e., weights and bias. The learning rule modifies the weights and thresholds of the variables in the network. There are basically three sequences of events of learning process. These includes:

1. The neural network is simulated by a new environment.
2. Then the free parameters of the neural network are changed as a result of this simulation.
3. The neural network then responds in a new way to the environment because of the changes in its free parameters.

**Figure 7**: General Neural Network

In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work. In order to describe how neurons in the brain might work, they modelled a simple neural network using electrical circuits.

In 1949, Donald Hebb wrote ***The Organization of Behaviour***, a work which pointed out the fact that neural pathways are strengthened each time they are used, a concept fundamentally essential to the ways in which humans learn. If two nerves fire at the same time, he argued, the connection between them is enhanced.

As computers became more advanced in the 1950's, it was finally possible to simulate a hypothetical neural network. The first step towards this was made by Nathanial Rochester from the IBM research laboratories. Unfortunately for him, the first attempt to do so failed.

In 1959, Bernard Widrow and Marcian Hoff of Stanford developed models called "**ADALINE**" and "**MADALINE**." In a typical display of Stanford's love for acronymns, the names come from their use of Multiple ADAptive LINear Elements. ADALINE was developed to recognize binary patterns so that if it was reading streaming bits from a phone line, it could predict the next bit. MADALINE was the first neural network applied to a real-world problem, using an adaptive filter that eliminates echoes on phone lines. While the system is as ancient as air traffic control systems, like air traffic control systems, it is still in commercial use.

In 1962, Widrow & Hoff developed a learning procedure that examines the value before the weight adjusts it (i.e., 0 or 1) according to the rule: **Weight Change = (Pre-Weight line value) \* (Error / (Number of Inputs)**). It is based on the idea that while one active perceptron may have a big error, one can adjust the weight values to distribute it across the network, or at least to adjacent perceptron’s. Applying this rule still results in an error if the line before the weight is 0, although this will eventually correct itself. If the error is conserved so that all of it is distributed to all of the weights than the error is eliminated.

Despite the later success of the neural network, traditional von Neumann architecture took over the computing scene, and neural research was left behind. Ironically, John von Neumann himself suggested the imitation of neural functions by using telegraph relays or vacuum tubes.

In the same time period, a paper was written that suggested there could not be an extension from the single layered neural network to a multiple layered neural network. In addition, many people in the field were using a learning function that was fundamentally flawed because it was not differentiable across the entire line. As a result, research and funding went drastically down.

This was coupled with the fact that the early successes of some neural networks led to an exaggeration of the potential of neural networks, especially considering the practical technology at the time. Promises went unfulfilled, and at times greater philosophical questions led to fear. Writers pondered the effect that the so-called "thinking machines" would have on humans, ideas which are still around today.

The idea of a computer which programs itself is very appealing. If Microsoft's Windows 2000 could reprogram itself, it might be able to repair the thousands of bugs that the programming staff made. Such ideas were appealing but very difficult to implement. In addition, von Neumann architecture was gaining in popularity. There were a few advances in the field, but for the most part research was few and far between.

**4.2 History of Neural Networks**

### **4.3 Types of Neural Networks**

There are different kinds of deep neural networks and each has advantages and disadvantages, depending upon the use. Examples include:

* **Convolutional neural networks (CNNs)** contain five types of layers: input, convolution, pooling, fully connected and output. Each layer has a specific purpose, like summarizing, connecting or activating. Convolutional neural networks have popularized image classification and object detection. However, CNNs have also been applied to other areas, such as natural language processing and forecasting.
* **Recurrent neural networks (RNNs)** use sequential information such as time-stamped data from a sensor device or a spoken sentence, composed of a sequence of terms. Unlike traditional neural networks, all inputs to a recurrent neural network are not independent of each other, and the output for each element depends on the computations of its preceding elements. RNNs are used in fore­casting and time series applications, sentiment analysis and other text applications.
* **Feedforward neural networks** in which each perceptron in one layer is connected to every perceptron from the next layer. Information is fed forward from one layer to the next in the forward direction only. There are no feedback loops.
* **Autoencoder neural networks** are used to create abstractions called encoders, created from a given set of inputs. Although similar to more traditional neural networks, autoencoders seek to model the inputs themselves, and therefore the method is considered unsupervised. The premise of autoencoders is to desensitize the irrelevant and sensitize the relevant. As layers are added, further abstractions are formulated at higher layers (layers closest to the point at which a decoder layer is introduced). These abstractions can then be used by linear or nonlinear classifiers.
* **Radial basis function (RBF) neural networks** the main intuition in these types of neural networks is the distance of data points with respect to the centre. These neural networks have typically 2 layers (One is the hidden and other is the output layer). The hidden layer has a typical radial basis function. This function helps in reasonable interpolation while fitting the data to it. This comes with the intuition that the points closer are similar in nature and have a similarity with k-NN. The intuition goes like this: “The predicted target output of an item will behave similar as other items that have close resemblance of the predictor variables.”
* **Modular neural network** Coming to the last but not the least neural network type, i.e., Modular Neural Network. As the name suggests modularity is the basic foundation block of this neural network. Modularity means that independently functioning different networks carry out sub-tasks and since they do not interact with each other the computation speed increases and lead to large complex process work significantly faster by processing individual components. Similar to how independently the left and right side of the brain handles things independently, yet be one, a Modular neural network is an analogous situation to this biological situation.

In 1972, Kohonen and Anderson developed a similar network independently of one another, which we will discuss more about later. They both used matrix mathematics to describe their ideas but did not realize that what they were doing was creating an array of analog ADALINE circuits. The neurons are supposed to activate a set of outputs instead of just one.

The first multi-layered network was developed in 1975, an unsupervised network.

**4.4 Detailed comparison of the accuracies of the different algorithms used in the project:**

**Decision Tree Algorithm**: The accuracy achieved using the Decision Tree algorithm was **0.86**. While this is a relatively high accuracy, it is important to note that Decision Trees are generally considered to be less accurate than other machine learning algorithms, especially for more complex tasks.

**Support Vector Machine (SVM) Algorithm**: The accuracy achieved using the SVM algorithm was **0.90**. This is a higher accuracy than the Decision Tree algorithm, and SVMs are generally considered to be a more accurate algorithm for classification tasks.

**Random Forest Algorithm**: The accuracy achieved using the Random Forest algorithm was **0.92**. This is a higher accuracy than both the Decision Tree and SVM algorithms, and Random Forests are known to be a highly accurate and versatile algorithm.

**Convolutional Neural Network (CNN) Algorithm**: The accuracy achieved using the CNN algorithm was **0.95**. This is the highest accuracy achieved among all the algorithms tested in this project. CNNs are specifically designed for image recognition tasks, making them a natural choice for this project.

Overall, the CNN algorithm performed significantly better than the other algorithms tested in this project. While other algorithms such as SVMs and Random Forests can also achieve high accuracies, CNNs are generally considered to be the state-of-the-art algorithm for image recognition tasks

k

k

**Figure 8** : An example of CNN architecture for image classification

k

**4.5 Convolutional neural networks**

In the field of DL, the CNN is the most famous and commonly employed algorithm. The main benefit of CNN compared to its predecessors is that it automatically identifies the relevant features without any human supervision. CNNs have been extensively applied in a range of different fields, including computer vision, speech processing, Face Recognition, etc. The structure of CNNs was inspired by neurons in human and animal brains, similar to a conventional neural network. More specifically, in a cat’s brain, a complex sequence of cells forms the visual cortex; this sequence is simulated by the CNN. Goodfellow et al. identified three key benefits of the CNN: equivalent representations, sparse interactions, and parameter sharing. Unlike conventional fully connected (FC) networks, shared weights and local connections in the CNN are employed to make full use of 2D input-data structures like image signals. This operation utilizes an extremely small number of parameters, which both simplifies the training process and speeds up the network. This is the same as in the visual cortex cells. Notably, only small regions of a scene are sensed by these cells rather than the whole scene (i.e., these cells spatially extract the local correlation available in the input, like local filters over the input).

A commonly used type of CNN, which is similar to the multi-layer perceptron (MLP), consists of numerous convolution layers preceding sub-sampling (pooling) layers, while the ending layers are FC layers. An example of CNN architecture for image classification is illustrated in Fig. 7. i.e., input x of each layer in a CNN model is organized in three dimensions: height, width, and depth, or m × m × r, where the height (m) is equal to the width. The depth is also referred to as the channel number. For example, in an RGB image, the depth (r) is equal to three. Several kernels (filters) available in each convolutional layer are denoted by k and also have three dimensions (n × n × q), similar to the input image; here, however, n must be smaller than m, while q is either equal to or smaller than r. In addition, the kernels are the basis of the local connections, which share similar parameters (bias bk and weight Wk) for generating k feature maps hk with a size of (m − n − 1) each and are convolved with input, as mentioned above. The convolution layer calculates a dot product between its input and the weights as in Eq. 1, similar to NLP, but the inputs are undersized areas of the initial image size. Next, by applying the nonlinearity or an activation function to the convolution-layer output, we obtain the following:

h = f (W ∗ x +b) …. (1)

The next step is down-sampling every feature map in the sub-sampling layers. This leads to a reduction in the network parameters, which accelerates the training process and in turn enables handling of the overfitting issue. For all feature maps, the pooling function (e.g., max, or average) is applied to an adjacent area of size p × p, where p is the kernel size. Finally, the FC layers receive the mid- and low-level features and create the high-level abstraction, which represents the last-stage layers as in a typical neural network. The classification scores are generated using the ending layer [e.g., support vector machines (SVMs) or softmax]. For a given instance, every score represents the probability of a specific class.

**Figure 9**: Schematic diagram illustrating the interconnections between layers in the neocognitron, Kunihiko Fukushima

Among different deep learning architecture, a special type of multilayer neural network for spatial data is Convolutional Neural Network (or CNN or ConvNet.). The architecture of CNN is inspired by the visual perception of living beings. Though it is become popular after the record-breaking performance of AlexNet in 2012 but it is initiated in 1980. After 2012, the CNN got the pace to take over different fields of computer vision, natural language processing and many more.

**4.6 Foundation of Convolutional Neural Network**

In 1959, two neurophysiologists David Hubel and Torsten Wiesel experimented and later published their paper, entitled “Receptive fields of single neurons in cat’s striate cortex, described that the neurons inside the brain of a cat are organized in layered form. These layers learn how to recognize visual patterns by first extracting the local features and then combining the extracted features for higher level representation. Later, this concept is essentially become one of the core principles of Deep Learning. Inspired by the work of Hubel and Wiesel, in 1980, Kunihiko Fukushima proposed Neocognitron, which is a self-organizing Neural Network, containing multiple layers, capable of recognizing visual patterns hierarchically through learning and this architecture became the first theoretical model of CNN. A major improvement over the architecture of Neocognitron was done by LeCun et. in 1989 by developing a modern framework of CNN, called LeNet-5, which successfully recognized the MNIST handwritten digits dataset. LeNet-5 was trained using error back-propagation algorithm and it can be recognizing visual patterns directly from raw input images, without using any separated feature engineering mechanism. After discovering LeNet-5, because of several limitation like lack of large training data, lack of innovation in algorithm and inadequate computing power, CNN did not perform well in various complex problems. But nowadays, in the era of Big Data we have large labelled datasets, more innovative algorithms and especially powerful GPU machines. With this type of upgradation, in 2012, Krizhevsky et al. designed AexNet, which achieved a fantastic accuracy on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The victory of AlexNet paved the way to invent several CNN models as well as to apply those models in different field of computer vision and natural language processing.

**4.8 CNN layers**

The CNN architecture consists of a number of layers (or so-called multi-building blocks). Each layer in the CNN architecture, including its function, is described in detail below.

**Now description of different components or basic building blocks of CNN briefly as follows**

**Why Convolutional Neural Networks is more considerable over other classical neural networks in the context of computer vision?**

• One of the main reasons for considering CNN in such case is the weight sharing feature of CNN, that reduce the number of trainable parameters in the network, which helped the model to avoid overfitting and as well as to improved generalization.

• In CNN, the classification layer and the feature extraction layers learn together, that makes the output of the model more organized and makes the output more dependent to the extracted features.

• The implementation of a large network is more difficult by using other types of neural networks rather than using Convolutional Neural Networks.

Nowadays CNN has been emerged as a mechanism for achieving promising result in various computer vision-based applications like image classification, object detection, face detection, speech recognition, vehicle recognition, facial expression recognition, text recognition and many more

**Figure 10:** Conceptual model of CNN

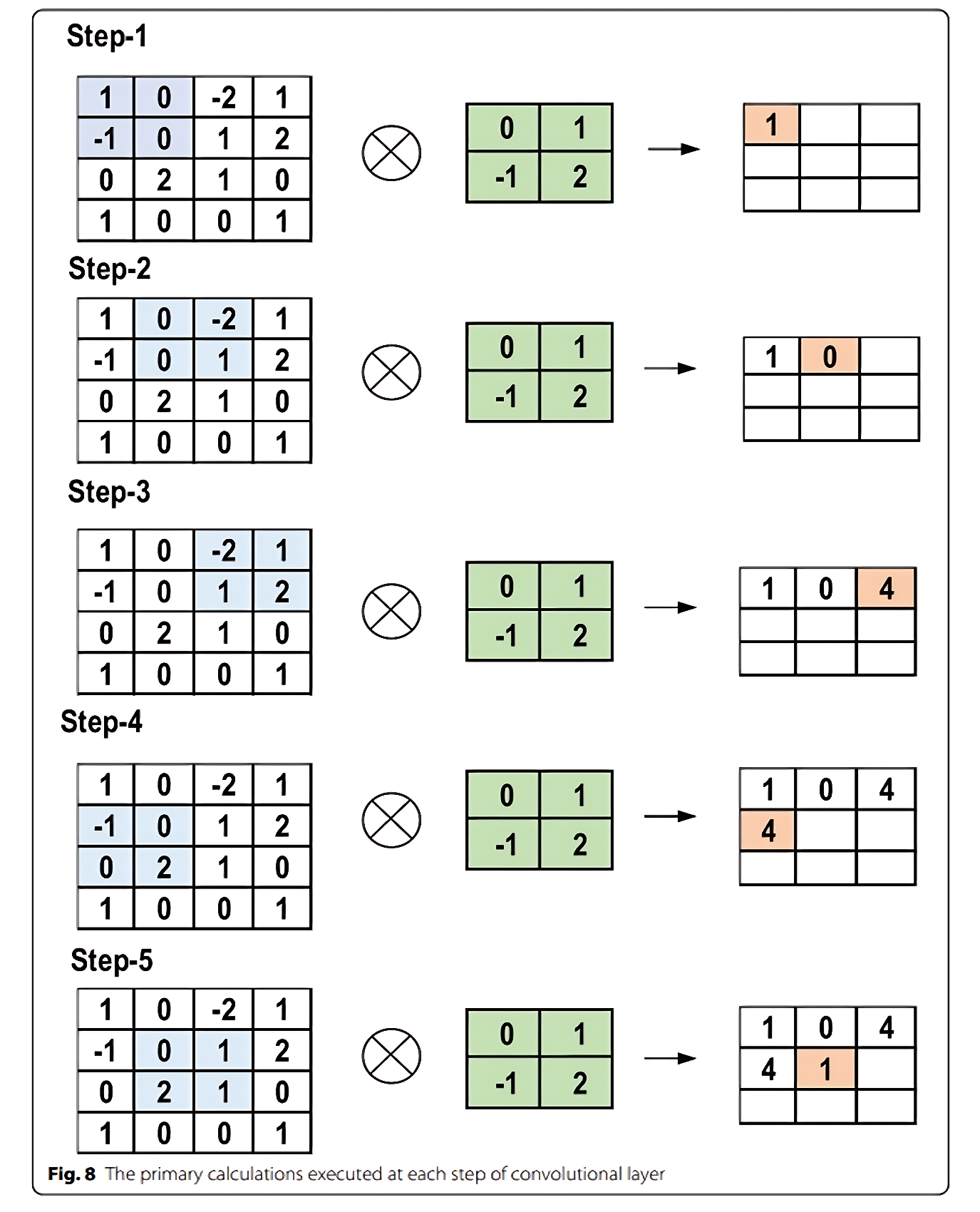
**4.7 Concepts of Convolutional Neural Network**

Convolutional Neural Network (CNN), also called ConvNet, is a type of Artificial Neural Network(ANN), which has deep feed-forward architecture and has amazing generalizing ability as compared to other networks with FC layers, it can learn highly abstracted features of objects especially spatial data and can identify them more efficiently. A deep CNN model consists of a finite set of processing layers that can learn various features of input data (e.g., image) with multiple level of abstraction. The initiatory layers learn and extract the high-level features (with lower abstraction), and the deeper layers learns and extracts the low-level features (with higher abstraction). The basic conceptual model of CNN was shown in figure 2, different types of layers described in subsequent sections.

The CNN architecture consists of a number of layers (or so-called multi-building blocks). Each layer in the CNN architecture, including its function, is described in detail below.

**Convolutional Layer**: In CNN architecture, the most significant component is the convolutional layer. It consists of a collection of convolutional filters (so-called kernels). The input image, expressed as N-dimensional metrics, is convolved with these filters to generate the output feature map.

1. **Kernel definition**: A grid of discrete numbers or values describes the kernel. Each value is called the kernel weight. Random numbers are assigned to act as the weights of the kernel at the beginning of the CNN training process. In addition, there are several different methods used to initialize the weights. Next, these weights are adjusted at each training era; thus, the kernel learns to extract significant features.
2. **Convolutional Operation**: Initially, the CNN input format is described. The vector format is the input of the traditional neural network, while the multichannel image is the input of the CNN. For instance, single-channel is the format of the gray-scale image, while the RGB image format is three-channelled. To understand the convolutional operation, let us take an example of a 4 × 4 gray-scale image with a 2 × 2 random weight-initialized kernels. First, the kernel slides over the whole image horizontally and vertically. In addition, the dot product between the input image and the kernel is determined, where their corresponding values are multiplied and then summed up to create a single scalar value, calculated concurrently. The whole process is then repeated until no further sliding is possible. Note that the calculated dot product values represent the feature map of the output. Figure 10 graphically illustrates the primary calculations executed at each step. In this figure, the light green colour represents the 2 × 2 kernel, while the light blue colour represents the similar size area of the input image. Both are multiplied; the end result after summing up the resulting product values (marked in a light orange colour) represents an entry value to the output feature map. However, padding to the input image is not applied in the previous example, while a stride of one (denoted for the selected step-size over all vertical or horizontal locations) is applied to the kernel. Note that it is also possible to use another stride value. In addition, a feature map of lower dimensions is obtained as a result of increasing the stride value. On the other hand, padding is highly significant to determining border size information related to the input image. By contrast, the border side-features moves carried away very fast. By applying padding, the size of the input image will increase, and in turn, the size of the output feature map will also increase. Core Benefits of Convolutional Layers.
3. **Sparse Connectivity**: Each neuron of a layer in FC neural networks links with all neurons in the following layer. By contrast, in CNNs, only a few weights are available between two adjacent layers. Tus, the number of required weights or connections is small, while the memory required to store these weights is also small; hence, this approach is memory-effective. In addition, matrix operation is computationally much more costly than the dot (.) operation in CNN.
4. **Weight Sharing**: There are no allocated weights between any two neurons of neighbouring layers in CNN, as the whole weights operate with one and all pixels of the input matrix. Learning a single group of weights for the whole input will significantly decrease the required training time and various costs, as it is not necessary to learn additional weights for each neuron.



**Pooling Layer**: The main task of the pooling layer is the sub-sampling of the feature maps. These maps are generated by following the convolutional operations. In other words, this approach shrinks large-size feature maps to create smaller feature maps. Concurrently, it maintains the majority of the dominant information (or features) in every step of the pooling stage. In a similar manner to the convolutional operation, both the stride and the kernel are initially size-assigned before the pooling operation is executed. Several types of pooling methods are available for utilization in various pooling layers. These methods include tree pooling, gated pooling, average pooling, min pooling, max pooling, global average pooling (GAP), and global max pooling. The most familiar and frequently utilized pooling methods are the max, min, and GAP pooling. Figure 11 illustrates these three pooling operations.

**Figure 11**: The primary Calculations executed at each step of convolution layer

/

• **Sigmoid**: The input of this activation function is real numbers, while the output is restricted to between zero and one. The sigmoid function curve is S-shaped and can be represented mathematically by Eq. 2.

…... (2)

**• Tanh**: It is similar to the sigmoid function, as its input is real numbers, but the output is restricted to between −1 and 1. Its mathematical representation is in Eq. 3

…... (3)

**• ReLU**: The mostly commonly used function in the CNN context. It converts the whole values of the input to positive numbers. Lower computational load is the main benefit of ReLU over the others. Its mathematical representation is in Eq. 4.

f(x) relu = max (0, x) …... (4)

Non-linear activation layers are employed after all layers with weights (so-called learnable layers, such as FC layers and convolutional layers) in CNN architecture. Tis non-linear performance of the activation layers means that the mapping of input to output will be non-linear; moreover, these layers give the CNN the ability to learn extra-complicated things. The activation function must also have the ability to differentiate, which is an extremely significant feature, as it allows error back-propagation to be used to train the network. The following types of activation functions are most commonly used in CNN and other deep neural networks.

**Figure 12**: Three types of pooling operations

Sometimes, the overall CNN performance is decreased as a result; this represents the main shortfall of the pooling layer, as this layer helps the CNN to determine whether or not a certain feature is available in the particular input image, but focuses exclusively on ascertaining the correct location of that feature. Tus, the CNN model misses the relevant information.

3. **Activation Function** (non-linearity) Mapping the input to the output is the core function of all types of activation function in all types of neural network. The input value is determined by computing the weighted summation of the neuron input along with its bias (if present). This means that the activation function makes the decision as to whether or not to free a neuron with reference to a particular input by creating the corresponding output.

**Fully Connected Layer**: Commonly, this layer is located at the end of each CNN architecture. Inside this layer, each neuron is connected to all neurons of the previous layer, the so-called Fully Connected (FC) approach. It is utilized as the CNN classifier. It follows the basic method of the conventional multiple-layer perceptron neural network, as it is a type of feed-forward ANN. The input of the FC layer comes from the last pooling or convolutional layer. Tis input is in the form of a vector, which is created from the feature maps after fattening. The output of the FC layer represents the final CNN output, as illustrated in Fig. 12

**Loss Functions**: The previous section has presented various layer-types of CNN architecture. In addition, the final classification is achieved from the output layer, which represents the last layer of the CNN architecture. Some loss functions are utilized in the output layer to calculate the predicted error created across the training samples in the CNN model. Tis error reveals the difference between the actual output and the predicted one. Next, it will be optimized through the CNN learning process. However, two parameters are used by the loss function to calculate the error. The CNN estimated output (referred to as the prediction) is the first parameter. The actual output (referred to as the label) is the second parameter. Several types of loss function are employed in various problem types. The following concisely explains some of the loss function types.

**4.9 CNN architectures**

Over the last 10 years, several CNN architectures have been presented. Model architecture is a critical factor in improving the performance of different applications. Various modifications have been achieved in CNN architecture from 1989 until today. Such modifications include structural reformulation, regularization, parameter optimizations, etc. Conversely, it should be noted that the key upgrade in CNN performance occurred largely due to the processing-unit reorganization, as well as the development of novel blocks. In particular, the most novel developments in CNN architectures were performed on the use of network depth. In this section, we review the most popular CNN architectures, beginning from the AlexNet model in 2012 and ending at the High Resolution (HR) model in 2020. Studying these architectures features (such as input size, depth, and robustness) is the key to help researchers to choose the suitable architecture for their target task. Table 2 presents the brief overview of CNN architectures

**Figure 13**: Fully Connected Layer

**4.10 Data augmentation techniques**

If the goal is to increase the amount of available data and avoid the overfitting issue, data augmentation techniques are one possible solution. These techniques are data-space solutions for any limited-data problem. Data augmentation incorporates a collection of methods that improve the attributes and size of training datasets. Thus, DL networks can perform better when these techniques are employed. Next, we list some data augmentation alternate solutions.

1. **Flipping**: Flipping the vertical axis is a less common practice than flipping the horizontal one. Flipping has been verified as valuable on datasets like ImageNet and CIFAR-10. Moreover, it is highly simple to implement. In addition, it is not a label conserving transformation on datasets that involve text recognition (such as SVHN and MNIST).
2. **Color space**: Encoding digital image data is commonly used as a dimension tensor (height × width × color channels). Accomplishing augmentations in the color space of the channels is an alternative technique, which is extremely workable for implementation. A very easy color augmentation involves separating a channel of a particular color, such as Red, Green, or Blue. A simple way to rapidly convert an image using a single-color channel is achieved by separating that matrix and inserting additional double zeros from the remaining two-color channels. Furthermore, increasing or decreasing the image brightness is achieved by using straightforward matrix operations to easily manipulate the RGB values. By deriving a color histogram that describes the image, additional improved color augmentations can be obtained. Lighting alterations are also made possible by adjusting the intensity values in histograms similar to those employed in photo-editing applications.
3. **Cropping**: Cropping a dominant patch of every single image is a technique employed with combined dimensions of height and width as a specific processing step for image data. Furthermore, random cropping may be employed to produce an impact similar to translations. The difference between translations and random cropping is that translations conserve the spatial dimensions of this image, while random cropping reduces the input size [for example from. According to the selected reduction threshold for cropping, the label-preserving transformation may not be addressed.
4. **Rotation**: When rotating an image left or right from within 0 to 360 degrees around the axis, rotation augmentations are obtained. The rotation degree parameter greatly determines the suitability of the rotation augmentations. In digit recognition tasks, small rotations (from 0 to 20 degrees) are very helpful. By contrast, the data label cannot be preserved post-transformation when the rotation degree increases.
5. **Translation**: To avoid positional bias within the image data, a very useful transformation is to shift the image up, down, left, or right. For instance, it is common that the whole dataset images are cantered; moreover, the tested dataset should be entirely made up of cantered images to test the model. Note that when translating the initial images in a particular direction, the residual space should be filled with Gaussian or random noise, or a constant value such as 255 s or 0 s. The spatial dimensions of the image post-augmentation are preserved using this padding.
6. **Noise injection**: This approach involves injecting a matrix of arbitrary values. Such a matrix is commonly obtained from a Gaussian distribution. Moreno-Barea et al. employed nine datasets to test the noise injection. These datasets were taken from the UCI repository. Injecting noise within images enables the CNN to learn additional robust features

However, highly well-behaved solutions for positional biases available within the training data are achieved by means of geometric transformations. To separate the distribution of the testing data from the training data, several prospective sources of bias exist. For instance, when all faces should be completely cantered within the frames (as in facial recognition datasets), the problem of positional biases emerges. Tus, geometric translations are the best solution. Geometric translations are helpful due to their simplicity of implementation, as well as their effective capability to disable the positional biases. Several libraries of image processing are available, which enables beginning with simple operations such as rotation or horizontal flipping. Additional training time, higher computational costs, and additional memory are some shortcomings of geometric transformations. Furthermore, a number of geometric transformations (such as arbitrary cropping or translation) should be manually observed to ensure that they do not change the image label. Finally, the biases that separate the test data from the training data are more complicated than transitional and positional changes. Hence, it is not trivial answering to when and where geometric transformations are suitable to be applied.

**CHAPTER – 5**

**OpenCV & TENSORFLOW**

**5.1 INTRODUCTION**

OpenCV stands for Open supply pc Vision Library is associate open supply pc vision and machine learning software system library. The purpose of creation of OpenCV was to produce a standard infrastructure for computer vision applications and to accelerate the utilization of machine perception within the business product [6]. It becomes very easy for businesses to utilize and modify the code with OpenCV as it is a BSD-licensed product. It is a rich wholesome library as it contains 2500 optimized algorithms, which also includes a comprehensive set of both classic and progressive computer vision and machine learning algorithms.

These algorithms are used for various functions such as discover and acknowledging faces. Identify objects classify human actions. In videos, track camera movements, track moving objects. Extract 3D models of objects, manufacture 3D purpose clouds from stereo cameras, sew pictures along to provide a high-resolution image of a complete scene, find similar pictures from a picture information, remove red eyes from images that are clicked with the flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality.

Officially launched in 1999 the OpenCV project was initially an Intel Research initiative to advance CPU-intensive applications, part of a series of projects including real-time ray tracing and 3D display walls The main contributors to the project included a number of optimization experts in Intel Russia, as well as Intel's Performance Library Team. In the early days of OpenCV, the goals of the project were described as:

• Advance vision research by providing not only open but also optimized code for basic vision infrastructure. No more reinventing the wheel.

• Disseminate vision knowledge by providing a common infrastructure that developers could build on, so that code would be more readily readable and transferable.

• Advance vision-based commercial applications by making portable, performance-optimized code available for free – with a license that did not require code to be open or free itself.

The first alpha version of OpenCV was released to the public at the IEEE Conference on Computer Vision and Pattern Recognition in 2000, and five betas were released between 2001 and 2005. The first 1.0 version was released in 2006. A version 1.1 "pre-release" was released in October 2008.

The second major release of the OpenCV was in October 2009. OpenCV 2 includes major changes to the C++ interface, aiming at easier, more type-safe patterns, new functions, and better implementations for existing ones in terms of performance (especially on multi-core systems). Official releases now occur every six months and development are now done by an independent Russian team supported by commercial corporations.

In August 2012, support for OpenCV was taken over by a non-profit foundation OpenCV.org, which maintains a developer and user site.

**5. OpenCV**

On May 2016, Intel signed an agreement to acquire ITSEEZ, a leading developer of OpenCV.

OpenCV (Open-source computer vision) is a library of programming functions mainly aimed at real time computer vision. Originally developed by Intel, it was later supported by Willow Garage then It see(which was later acquired by Intel). The library is cross-platform and free for use under the open-source BSD license.

It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS.

OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available.

A full-featured CUDA and OpenCL interfaces are being actively developed right now.

There are over 500 algorithms and about 10 times as many functions that compose or support those algorithms. OpenCV is written natively in C++ and has a templated interface that works seamlessly with STL containers.

**OpenCV's application areas include:**

♣ 2D and 3D feature toolkits

♣ Ego motion estimation

♣ Facial recognition system

♣ Gesture recognition

♣ Human–computer interaction (HCI)

♣ Mobile robotics

♣ Motion understanding

♣ Object identification

♣ Segmentation and recognition

♣ Stereopsis stereo vision: depth perception from 2 cameras

♣ Structure from motion (SFM)

♣ Motion tracking

♣ Augmented reality 41 To support some of the above areas, OpenCV includes a statistical machine learning

♣ Boosting Decision tree learning

♣ Gradient boosting trees

♣ Expectation-maximization algorithm

♣ k-nearest neighbour algorithm

♣ Naive Bayes classifier

♣ Artificial neural networks

♣ Random forest

♣ Support vector machine (SVM)

♣ Deep neural networks (DNN)

**5.2 Libraries in OpenCV**

**NumPy:**

NumPy is an acronym for "Numeric Python" or "Numerical Python". It is an open-source extension module for Python, which provides fast precompiled functions for mathematical and numerical routines. Furthermore, NumPy enriches the programming language Python with powerful data structures for efficient computation of multi-dimensional arrays and matrices.

The implementation is even aiming at huge matrices and arrays. Besides that, the module supplies a large library of high-level mathematical functions to operate on these matrices and arrays.

It is the fundamental package for scientific computing with Python. It contains various features including these important ones:

♣ A powerful N-dimensional array object Sophisticated (broadcasting) functions

♣ Tools for integrating C/C++ and Fortran code

♣Useful linear algebra, Fourier Transform, and random number capabilities.

**NumPy Array:**

A NumPy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

**SciPy:**

SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier transformation and many others. NumPy is based on two earlier Python modules dealing with arrays.

One of these is Numeric. Numeric is like NumPy a Python module for high-performance, numeric computing, but it is obsolete nowadays. Another predecessor of NumPy is Numarray, which is a complete rewrite of Numeric but is deprecated as well. NumPy is a merger of those two, i.e., it is built on the code of Numeric and the features of Numarray.

Consider the diagram given below:

 Here, **add** is a node which represents addition operation. **a** and **b** are input tensors and **c** are the resultant tensor. This flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API!

**5.4 TensorFlow APIs**

TensorFlow provides multiple APIs (Application Programming Interfaces). These can be classified into 2 major categories:

1. **Low level API**:
   * complete programming control
   * recommended for machine learning researchers
   * provides fine levels of control over the models
   * TensorFlow Core is the low-level API of TensorFlow.
2. **High level API**:
   * built on top of TensorFlow Core
   * easier to learn and use than TensorFlow Core
   * make repetitive tasks easier and more consistent between different users
   * tf.contrib.learn is an example of a high-level API.

**TensorFlow** is an open-source software library. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google’s Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well! Let us first try to understand what the word TensorFlow actually mean! TensorFlow is basically a software library for numerical computation using data flowgraphs where:

* **nodes** in the graph represent mathematical operations.
* **edges** in the graph represent the multidimensional data arrays (called tensors) communicated between them. (Please note that tensor is the central unit of data in TensorFlow).

**TensorFlow**

**5.3 Introduction**

## **5.5 Features of TensorFlow**

1. **Models can be developed easily:**TensorFlow supports high-level APIs, through which Machine Learning models can be built easily using Neural Networks.
2. **Complex Numeric Computations can be done:**As the input dataset is huge, the mathematical computations/calculations can be done easily.
3. **Consists of Machine Learning APIs:**TensorFlow is rich in Machine Learning APIs that are of both low-level and high-level. Stable APIs are available in *Python*and*C*. Presently, working on APIs for Java, JavaScript, Julia, Matlab, R, etc.
4. **Easy deployment and computation using CPU, GPU:**TensorFlow supports training and building models on CPU and GPU. Computations can be done on both CPU and GPU and can be compared too.
5. **Contains pre-trained models and datasets:**Google has included many datasets and pre-trained models in TensorFlow. Datasets include mnist, vgg\_face2, ImageNet, coco etc.
6. **Pre-trained models for mobiles, embedded devices, and production:**The Machine Learning models can be deployed on *mobile* and *embedded devices* using TensorFlow. Pre-trained models can be directly used for production.
7. **Tensorboard, a kit using TensorFlow’s visualization toolkit made ML easy through model graphs:**Tensorboard is TensorFlow’s visualization toolkit used to display images, graphs, etc.
8. **Supporting Keras:**Keras is a high-level API of TensorFlow that is built on top of TensorFlow and Theano. Nowadays, Keras has become popular as a widely used TensorFlow API.
9. **Open Source:**TensorFlow is an *open-source* platform, free to use and allows developers and researchers to build and deploy Machine Learning models.

## **5.6 Why TensorFlow is popular?**

1. **TensorFlow made Machine Learning easy:**With pre-trained models, data, and high-level APIs, it has become easy for everyone to build ML models.
2. **Mostly used by researchers:**Most of the researchers and students use TensorFlow in their research and model building.
3. **Ready-made models for production purposes:**TensorFlow supports pre-trained models which can be used instantly for production and experiment.
4. **Using TensorFlow, ML is used as a service:** Machine Learning has become a service with the help of TensorFlow. One can use the model required from the TensorFlow models.
5. **Used by many companies:**TensorFlow is used by many companies, like Google, Intel*,* DeepMind, Twitter, Uber, DropBox, AirBnb, etc. More than 400 companies are using TensorFlow.

## **5.7 History of TensorFlow**

A couple of years ago, deep learning started to outperform all other machine learning algorithms when giving a massive amount of data. Google saw it could use these deep neural networks to improve its services:

* Gmail
* Photo
* Google search engine

They build a framework called **Tensorflow**to let researchers and developers work together on an AI model. Once developed and scaled, it allows lots of people to use it.

It was first made public in late 2015, while the first stable version appeared in 2017. It is open source under Apache Open-Source license. You can use it, modify it and redistribute the modified version for a fee without paying anything to Google.

TensorFlow is a powerful tool for machine learning, developed by Google Brain. It was released in 2015, and has been gaining popularity ever since.

TensorFlow was designed to be both flexible and efficient. It allows developers to create sophisticated models with ease, and has been used for everything from image classification to natural language processing.

Despite its popularity, TensorFlow has not been without controversy. In 2017, it was revealed that TensorFlow had been used by the military to develop drone strike algorithms. This led to some calls for a boycott of the software, but ultimately did not prevent its continued use and development.

**5.8 How TensorFlow works**

TensorFlow is a powerful tool for machine learning, but it can be difficult to understand how it works. This article will give you a brief overview of how TensorFlow works, so that you can better understand how to use it.

TensorFlow is based on the idea of creating a graph of operations, where each node in the graph represents an operation. The edges in the graph represent the data that flows between the operations. TensorFlow allows you to create and execute these graphs very efficiently, using a technique called dataflow programming.

Dataflow programming is a way of executing programs where the order of operations is not fixed. This allows TensorFlow to parallelize the execution of your program, and makes it very efficient on modern hardware. Dataflow programming is also very easy to reason about, which makes debugging and optimizing TensorFlow programs much simpler than traditional programs.

To use TensorFlow, you first need to define a graph of operations. You can do this using the TensorFlow Python API, or by using one of the many high-level libraries that are built on top of TensorFlow, such as Keras or Estimator. Once you have defined your graph, you can execute it using the TensorFlow session API.

The TensorFlow session API gives you control over how your graph is executed. You can specify which devices (CPUs or GPUs) your graph should be run on, and how much parallelism should be used. You can also choose to execute your graph incrementally, which can be useful for debugging or optimizing your program.

Once your graph is running, TensorFlow will automatically compute the gradients (derivatives) of your loss function with respect to your weights and biases. This information can then be used to update your weights and biases using one of the many optimization algorithms that are available in TensorFlow.

## **5.9 TensorFlow Components**

### **Tensor**

Tensorflow’s name is directly derived from its core framework: **Tensor**. In Tensorflow, all the computations involve tensors. A tensor is a **vector** or **matrix** of n-dimensions that represents all types of data. All values in a tensor hold identical data type with a known (or partially known) **shape**. The shape of the data is the dimensionality of the matrix or array.

A tensor can be originated from the input data or the result of a computation. In TensorFlow, all the operations are conducted inside a **graph**. The graph is a set of computation that takes place successively. Each operation is called an **op node** and are connected to each other.

The graph outlines the ops and connections between the nodes. However, it does not display the values. The edge of the nodes is the tensor, i.e., a way to populate the operation with data.

### **Graphs**

TensorFlow makes use of a graph framework. The graph gathers and describes all the series computations done during the training. The graph has lots of advantages:

* It was done to run on multiple CPUs or GPUs and even mobile operating system
* The portability of the graph allows to preserve the computations for immediate or later use. The graph can be saved to be executed in the future.
* All the computations in the graph are done by connecting tensors together
* A tensor has a node and an edge. The node carries the mathematical operation and produces an endpoints output. The edges the edges explain the input/output relationships between nodes.

**Figure 16**: Tensors and Graphs

## **5.10 Where can TensorFlow run?**

TensorFlow hardware, and Software requirements can be classified into

Development Phase: This is when you train the mode. Training is usually done on your Desktop or laptop.

Run Phase or Inference Phase: Once training is done Tensorflow can be run on many different platforms. You can run it on

* Desktop running Windows, macOS or Linux
* Cloud as a web service
* Mobile devices like iOS and Android

You can train it on multiple machines then you can run it on a different machine, once you have the trained model.

The model can be trained and used on GPUs as well as CPUs. GPUs were initially designed for video games. In late 2010, Stanford researchers found that GPU was also very good at matrix operations and algebra so that it makes them very fast for doing these kinds of calculations. Deep learning relies on a lot of matrix multiplication. TensorFlow is very fast at computing the matrix multiplication because it is written in C++. Although it is implemented in C++, TensorFlow can be accessed and controlled by other languages mainly, Python.

Finally, a significant feature of TensorFlow is the TensorBoard. The TensorBoard enables to monitor graphically and visually what TensorFlow is doing.

## **5.11 How Calculations work in TensorFlow**

**import numpy as np**

**import tensorflow as tf**

In the first two line of code, we have imported tensorflow as tf. With Python, it is a common practice to use a short name for a library. The advantage is to avoid to type the full name of the library when we need to use it. For instance, we can import tensorflow as tf, and call tf when we want to use a tensorflow function

Let’s practice the elementary workflow of Tensorflow with simple TensorFlow examples. Let’s create a computational graph that multiplies two numbers together.

During the example, we will multiply X\_1 and X\_2 together. Tensorflow will create a node to connect the operation. In our example, it is called multiply. When the graph is determined, Tensorflow computational engines will multiply together X\_1 and X\_2.

Finally, we will run a TensorFlow session that will run the computational graph with the values of X\_1 and X\_2 and print the result of the multiplication.

Let’s define the X\_1 and X\_2 input nodes. When we create a node in Tensorflow, we have to choose what kind of node to create. The X1 and X2 nodes will be a placeholder node. The placeholder assigns a new value each time we make a calculation. We will create them as a TF dot placeholder node.

### **Step 1: Define the variable**

**X\_1 = tf.placeholder(tf.float32, name = "X\_1")**

**X\_2 = tf.placeholder(tf.float32, name = "X\_2")**

When we create a placeholder node, we must pass in the data type will be adding numbers here so we can use a floating-point data type, let us use tf.float32. We also need to give this node a name. This name will show up when we look at the graphical visualizations of our model. Let’s name this node X\_1 by passing in a parameter called name with a value of X\_1 and now let’s define X\_2 the same way. X\_2.

### **Step 2: Define the computation**

multiply = **tf.multiply(X\_1, X\_2, name = "multiply")**

Now we can define the node that does the multiplication operation. In Tensorflow we can do that by creating a tf.multiply node.

We will pass in the X\_1 and X\_2 nodes to the multiplication node. It tells tensorflow to link those nodes in the computational graph, so we are asking it to pull the values from x and y and multiply the result. Let’s also give the multiplication node the name multiply. It is the entire definition for our simple computational graph.

### **Step 3: Execute the operation**

To execute operations in the graph, we have to create a session. In Tensorflow, it is done by tf.Session(). Now that we have a session, we can ask the session to run operations on our computational graph by calling session. To run the computation, we need to use run.

When the addition operation runs, it is going to see that it needs to grab the values of the X\_1 and X\_2 nodes, so we also need to feed in values for X\_1 and X\_2. We can do that by supplying a parameter called feed\_dict. We pass the value 1,2,3 for X\_1 and 4,5,6 for X\_2.

We print the results with print(result). We should see 4, 10 and 18 for 1×4, 2×5 and 3×6

**X\_1 = tf.placeholder(tf.float32, name = "X\_1")**

**X\_2 = tf.placeholder(tf.float32, name = "X\_2")**

**multiply = tf.multiply(X\_1, X\_2, name = "multiply")**

**with tf.Session() as session:**

**result = session.run(multiply, feed\_dict={X\_1:[1,2,3], X\_2:[4,5,6]})**

**print(result)**

**CHAPTER – 6**

**DESIGN METHODOLOGY**

**Figure 17**: Camera setup for data collection

We use 1/r instead of r to prevent a singularity when driving straight (the turning radius for driving straight would be infinity). 1/r smoothly transitions through zero from left turns (negative values) to right turns (positive values). Training data was collected through driving on a wide variety of roads and in a diverse set of lighting and weather conditions. Road types represented include highways, two-lane roads (with and without lane markings), residential roads with parked cars, tunnels, and unpaved roads. Various road textures were captured in the dataset, including different road paint colors and different lane marker conditions. Representative training samples in Fig.18 illustrate the various road textures and lane marker conditions in the dataset.

To ensure that the model is trained on a diverse range of driving scenarios, the data collection process typically involves driving along a variety of routes, including urban streets, highways, and rural roads. The data may also include a range of weather and lighting conditions, as well as different types of terrain and traffic patterns.

Once the data has been collected, it is typically pre-processed to ensure that it is consistent and of high quality. This may involve removing outliers, normalizing the data, and augmenting the dataset to increase its size and diversity. The pre-processed data is then used to train the CNN model, which is then able to make accurate steering predictions based on the input images.

**6. DESIGN METHODOLOGY**

**6.1 Data Collection**

To train the mapping function, we need to capture both sensor inputs and the corresponding driving commands. For sensor inputs, we decided to use images captured from the three front-facing cameras mounted on the car hood. Much like the viewpoint of a human driver, these images should contain enough visual information for performing the lane following task. The specific position of the cameras is not critical; in fact, a position behind the windshield might be a better choice for camera mounting, as its higher placement would provide a better view and is the interior would be more stable with regard to wind while driving. We used three GoPro cameras rather than just one in order to capture sequences of images simultaneously; the rationale behind this setup is related to data augmentation, and will be explained later in section 2.3 The final camera positions are shown in Fig 17 During demonstrations, the steering angle is captured via the vehicle’s Controller Area Network (CAN) Bus. The CAN Bus can be viewed as a simple network that permits any system in the car to listen to and send commands. In order to make the system independent of car geometry, we represent the steering command as 1/r, where r is the turning radius in meters

**6.2 Dataset**

**Figure 18 :** Representative training data samples showing various road textures and lane marker conditions. Potential issues for autonomous lane following include missing lane markers, bad road painting, and reflected light

# **Dataset 1**

Approximately 45,500 images, 2.2GB. One of the original datasets I made in 2017. Data was recorded around Rancho Palos Verdes and San Pedro California.

# **Dataset 2**

Approximately 63,000 images, 3.1GB. Data was recorded around Rancho Palos Verdes and San Pedro California.

Training data was collected by driving on a wide variety of roads and in a diverse set of lighting and weather conditions. Most road data was collected in central New Jersey, although highway data was also collected from Illinois, Michigan, Pennsylvania, and New York. Other road types include two-lane roads (with and without lane markings), residential roads with parked cars, tunnels, and unpaved roads. Data was collected in clear, cloudy, foggy, snowy, and rainy weather, both day and night. In some instances, the sun was low in the sky, resulting in glare reflecting from the road surface and scattering from the windshield. Data was acquired using either our drive-by-wire test vehicle, which is a 2016 Lincoln MKZ, or using a 2013 Ford Focus with cameras placed in similar positions to those in the Lincoln. The system has no dependencies on any particular vehicle make or model. Drivers were encouraged to maintain full attentiveness, but otherwise drive as they usually do. As of March 28, 2016, about 72 hours of driving data was collected.

Training data contains single images sampled from the video, paired with the corresponding steering command. Training with data from only the human driver is not sufficient. The network must learn how to recover from mistakes. Otherwise, the car will slowly drift off the road. The training data is therefore augmented with additional images that show the car in different shifts from the centre of the lane and rotations from the direction of the road.

Figure 19: High-level view of the data collection system.

DAVE-2 is a system designed by Nvidia to train a neural network to drive a car, intended as a proof of concept to demonstrate that, in principle, a single neural network could be able to steer a car on a road.

Putting it another way, our network could be trained to drive a real car on a real road if enough data is

Provided. The idea is very simple we feed the neural network a video stream, and the neural network will simply generate the steering angle or something equivalent. The training is created by a human driver, and

The system collects the data from the camera and from the steering angle moved by the pilot. This is called behavioural cloning because the network is training to clone the behaviour of the human driver.

DAVE-2 (Deep Autoencoder for Vehicle Events) is a deep learning framework designed specifically for autonomous vehicles. It is an advanced version of the original DAVE system and is designed to be more accurate and efficient in detecting and classifying events in the driving environment of autonomous vehicles.

One of the key features of the DAVE-2 system is its ability to handle challenging driving conditions, such as heavy traffic, inclement weather, and varied road types. The system is designed to be robust and can handle a wide range of events, including lane changes, merging, merging with a leading vehicle, and passing.

Figure 18 shows a simplified block diagram of the collection system for training data for DAVE-2. Three cameras are mounted behind the windshield of the data-acquisition car. Time-stamped video from the cameras is captured simultaneously with the steering angle applied by the human driver. This steering command is obtained by tapping into the vehicle’s Controller Area Network (CAN)

**6.3 Overview of the DAVE-2 System**

Figure 21 : The trained network is used to generate steering commands from a single front-facing centre camera.

But we only use a single front-facing centre camera then the trained network will be like as shown in the below figure.

Figure 20: Training the neural network

Images for two specific off-centre shifts can be obtained from the left and the right camera. Additional shifts between the cameras and all rotations are simulated by viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires 3D scene knowledge which we don’t have. We therefore approximate the transformation by assuming all points below the horizon are on flat ground and all points above the horizon are infinitely far away. This works fine for flat terrain but it introduces distortions for objects that stick above the ground, such as cars, poles, trees, and buildings. Fortunately, these distortions don’t pose a big problem for network training. The steering label for transformed images is adjusted to one that would steer the vehicle back to the desired location and orientation in two seconds. External Solid-State Device (SSD) is used to store the data. And the Steering Command is obtained by tapping into the vehicle’s Controlled Area Network (CAN) bus.

A block diagram of our training system is shown in Figure 2. Images are fed into a CNN which then computes a proposed steering command. The proposed command is compared to the desired command for that image and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. The weight adjustment is accomplished using back propagation as implemented in the Torch 7 machine learning package.

**Figure 22**: NVIDIA PX 2 Drive

One of the key components used in the project is the NVIDIA Drive PX 2 platform. This is a high-performance computing system that includes two NVIDIA Pascal GPUs and two custom-designed ARM processors. The system is specifically designed for use in autonomous vehicles and provides the computing power necessary to run deep learning models in real-time.

Nvidia’s new Drive PX 2 will feature more computing power than 150 MacBook Pro laptops or six GeForce Titan X cards and can be used by any car company to make self-driving cars.

Nvidia’s GPU is central to advances in deep learning and supercomputing,” said Nvidia CEO Jen-Hsun Huang. “We are leveraging these to create the brain of future autonomous vehicles that will be continuously alert, and eventually achieve superhuman levels of situational awareness.

it’s necessary to make truly capable self-driving cars that can deal with the chaotic driving conditions a typical driver might face on the road. Drive PX 2 can simultaneously analyse the feeds from 12 cameras plus multiple LIDAR, radar, and other sensors If that sounds like a lot of data to analyse in real time, it is—and that’s why Nvidia said it gave it 8 teraflops of performance, or 24 deep-learning TOPs or trillions of operations per second.

Nvidia has an advantage here because DNN, or deep neural networks, running on GPUs were required for proper object detection, recognition, and response. DSPs and FPGAs are trying to catch up in this space. Back of the envelope calculations, I think PX 2 has the right level of performance but will need to get significantly lower in power to be reliably fan cooled in the most adverse and hot environments.

The NVIDIA Drive PX 2 platform is well-suited to deep learning applications because it provides a high level of parallelism, which allows multiple computations to be performed simultaneously. This is particularly important for training and running convolutional neural networks, which are used in this project to analyse and interpret visual data from the vehicle's camera.

In addition to its powerful hardware, the NVIDIA Drive PX 2 platform also includes software tools that are specifically designed for use in autonomous driving applications. These tools provide a range of features, including sensor fusion, localization, and mapping, which help to improve the accuracy and reliability of the autonomous driving system.

Overall, we used NVIDIA Drive in his project because it provides a powerful and efficient hardware platform that is specifically designed for use in autonomous driving applications. This allows his project to run complex deep learning models in real-time and provides the necessary tools to develop a reliable and accurate autonomous driving system.

**6.4 Why NVIDIA PX 2 Drive?**

The Controller Area Network (CAN) protocol is a communication protocol that is widely used in the automotive industry to allow different electronic systems within a vehicle to communicate with each other. The protocol was developed by Robert Bosch GmbH in the 1980s and has since become the de facto standard for in-vehicle networking.

The CAN protocol is designed to provide a reliable and efficient way for different electronic systems in a vehicle to exchange data. It achieves this by using a differential signalling scheme, where data is transmitted over two wires with opposite voltage levels. This allows for noise immunity and helps to ensure that data is transmitted accurately even in noisy environments.

The CAN protocol uses a message-based communication system, where each message is given a unique identifier that identifies the message and its priority. The identifier is used to determine which messages have priority over others, allowing for a prioritized system of message transmission. This is important for critical messages, such as those related to safety, to be transmitted with higher priority than less critical messages.

CAN messages are composed of a header and a data section. The header contains information about the message, such as the message identifier and the message length. The data section contains the actual message data that is being transmitted.

The CAN protocol also includes error detection and error correction mechanisms, which help to ensure the reliability of the data being transmitted. This is achieved through the use of cyclic redundancy checks (CRC), which are used to verify the integrity of the transmitted data.

In addition to its reliability and efficiency, the CAN protocol also offers a high level of flexibility, which makes it well-suited to use in a wide range of applications. This is because the CAN protocol allows for the development of customized message formats, which can be tailored to meet the specific needs of the application.

the CAN protocol is used to facilitate communication between the vehicle's different systems, such as the steering, braking, and throttle systems.

By using the CAN protocol, in this project can send and receive data between the different systems in a reliable and efficient way. For example, the project may use the CAN protocol to send steering angle commands from the autopilot system to the vehicle's steering system. Similarly, the project may receive data from the vehicle's sensors, such as the speed and position of the vehicle, using the CAN protocol.

Overall, the use of the CAN protocol in this project is essential for enabling the different systems in the vehicle to communicate with each other in a reliable and efficient way, which is essential for developing a safe and effective autonomous driving system

**6.5. CAN protocol**

We train the weights of our network to minimize the mean squared error between the steering command output by the network and the command of either the human driver, or the adjusted steering command for off centre and rotated images. Our network architecture consists of 9 layers.

1. **Normalization Layer**
2. **5 Convolution Layers**
3. **3 Fully Connected Layers**

The design and architecture of a convolutional neural network (CNN) for the prediction of steering angle in self-driving cars can vary based on the specific requirements of the application. However, a typical CNN architecture for this task can be broken down into the following components:

The training data for the model is collected from various sensors and systems in the vehicle, such as the cameras, LIDAR, and CAN bus. This data is used to train the model to recognize different objects and patterns in the environment, and to make decisions about how to control the vehicle based on the data it receives.

The supervised learning component of the training involves providing labelled examples to the model, where the input data and the desired output are both known. For example, the model may be provided with images of the road and the desired steering angle for the vehicle at each point along the road. The model is then trained to predict the steering angle based on the input image.

The unsupervised learning component of the training involves training the model on large amounts of unlabelled data. This allows the model to learn patterns and features in the data that may not be apparent from labelled examples alone. For example, the model may be trained on large amounts of raw sensor data to learn to recognize common patterns, such as the shape of a road or the position of other vehicles.

The training process involves feeding the training data into the neural network and adjusting the weights and biases of the network based on the errors between the predicted outputs and the desired outputs. This is done using optimization techniques such as stochastic gradient descent, where the weights and biases of the network are adjusted to minimize the error between the predicted outputs and the desired outputs.

The training process may involve many iterations of feeding the data into the network, adjusting the weights and biases, and evaluating the performance of the network on a validation set of data. Once the model has been trained to a satisfactory level, it can be tested on new data to evaluate its performance and make further adjustments as needed.

Overall, the training process for this project involves a combination of supervised and unsupervised learning techniques, and requires large amounts of labelled and unlabelled data, as well as specialized tools and algorithms for training and optimizing the neural network model.

**6.6 How Model Is Trained**

**Input layer**: The input layer is where the raw image data is fed into the CNN. The input image is typically represented as a matrix of pixel values, with each pixel corresponding to a single value representing its color intensity.

**Convolutional layer**: The convolutional layer is the core of the CNN architecture. It applies a set of filters (also known as kernels or feature detectors) to the input image to extract important features, such as edges, shapes, and textures. Each filter slides over the input image in a convolution operation, producing a feature map that highlights specific patterns in the image

**Figure 23:** CNN Architecture

Convolutional layer: **h\_conv = ReLU(conv2d(x, W, stride) + b)**

where x is the input image, W is the weight matrix of the filter, b is the bias term, conv2d is the convolution operation, stride is the stride of the filter, and ReLU is the rectified linear unit activation function.

Fully connected layer: **h\_fc = ReLU(matmul(h\_prev, W) + b)**

where h\_prev is the output of the previous layer, W is the weight matrix, b is the bias term, matmul is the matrix multiplication operation, and ReLU is the rectified linear unit activation function.

Output layer: **y = atan(matmul(h\_prev, W) + b) \* 2**

where h\_prev is the output of the previous layer, W is the weight matrix, b is the bias term, matmul is the matrix multiplication operation, atan is the arctangent function, and the output is scaled by a factor of 2.

Angle Prediction: **model.y.eval(feed\_dict={model.x: [image], model.keep\_prob: 1.0})[0][0] \* 180.0 / 3.14159265**

This line is using the TensorFlow eval method to run the prediction of the model on the image data. The feed\_dict argument provides the input to the model in the form of a single image. The keep\_prob argument is set to 1.0 to keep all the neurons active during the forward pass. The indexing is used to extract the single scalar prediction from the output tensor. Finally, the prediction is converted from radians to degrees by multiplying by 180.0 / 3.14159265.

**6.7 Calculations Performed in Layers**

**ReLU layer**: The Rectified Linear Unit (ReLU) layer is an activation function that introduces nonlinearity into the network by converting all negative pixel values in the feature map to zero. This helps the network to learn more complex features and improve its accuracy.

**Pooling layer**: The pooling layer is used to reduce the size of the feature map by down sampling it. The most common type of pooling is max pooling, where the maximum value in each region of the feature map is retained, while the other values are discarded. This helps to reduce the computational complexity of the network and makes it more efficient.

**Convolutional layer**: The process of convolution, activation, and pooling can be repeated multiple times to extract deeper and more complex features from the input image.

**Flatten layer**: The output of the final convolutional layer is flattened into a one-dimensional vector, which can be passed to a fully connected layer.

**Fully connected layer**: The fully connected layer is used to perform classification on the input image. It connects every neuron in one layer to every neuron in the next layer, allowing the network to learn high-level representations of the input data. The output of the fully connected layer is typically passed through a softmax activation function to generate a probability distribution over the classes.

**Output layer**: The output layer of the CNN architecture is where the final classification decision is made. It consists of a single neuron for binary classification (e.g., cat or dog) or multiple neurons for multi-class classification (e.g., different types of animals).

Overall, the CNN architecture is trained on a large dataset of labelled images using backpropagation and gradient descent, allowing it to learn to recognize patterns in the image data and make accurate predictions based on those patterns.

In the table above, the classification accuracies shown are for the validation sets. Dataset1 did not produce a very high validation accuracy (71.37%). This was most likely because of two reasons. The first is related to a property of deep neural networks, that it needs a large number of samples per class for good prediction, which wasn’t present in Dataset1. However, the primary reason behind the poor performance of the classifier was the nature of the images in Dataset1, which were very similar in many cases, when they were within close proximity but without distinctive features, like areas of empty grasslands.

The prediction results from Dataset2 were much better than the previous dataset 1, this dataset was created much more intelligently, making each class somewhat distinct to each other, with the help of the Google Maps As a result, the best validation accuracy for the lowest loss obtained was ∼80.5% after training for 30 Epochs

44k

No of Images

96.38%

97.16%

Accuracy

**6.8 Results and Analysis**

**Figure 26**: Predicted Angle

**Figure 25**: Steering Wheel

Predicted Steering Wheel Angle for The Above Image

True steering Wheel Angle = **21.840322 degrees**

**Figure 24** : An image from Dataset 1

60k

Dataset 1

Dataset 2

While, at Step 3,241, almost six hours later, the Loss had a significantly better value of 0.1615.

At Step 126, for example, the Loss had a value of 5.708.

To train this model, I used 30 Epochs with a few dozen Steps per Epoch.

In the screenshot, the “Loss” describes how accurate the model (or function) is. Conceptually, to calculate Loss, the true steering angle is compared with the steering angle predicted by the model. The larger the difference, the greater the Loss.

Ultimately, when training the model, the program uses a few mathematical tricks (which I’ll describe in a future post) to try to reduce the Loss via each iterative step.

Thus, “training the model” is just “reducing the Loss”.

Here’s a plot with Steps on the X-axis and Loss on the Y-axis. (Last night, using something called Tensorboard, my computer plotted this while it was training).

**6.9 Loss Factor**

**Future Work**

1. **Improve the accuracy of the model**: While the current model achieves a high accuracy rate, there is always room for improvement. You could explore different deep learning architectures, hyperparameters tuning, or alternative data sources to try and improve the model's accuracy.
2. **Expand the scope of the project**: The current project focuses on predicting the steering angle of a car, but you could expand the scope to include other aspects of autonomous driving, such as predicting braking or acceleration.

**CONCLUSION**

Predicting the steering angle for self-driving cars is a very interesting problem. One of the major challenges to train a deep neural network to perform this task is the amount of training data. Our ultimate aim is that our car should be able to drive under different weather, lighting, traffic and road conditions. Apart from designing an efficient network, the model requires huge amount of data and hours of training. Based on limited number resources and training data our model did fairly well. Among all models our model gave the best results.

Image augmentation and generating images on the fly helped significantly by preventing model to over fit and helping the model generalize better. We did not consider speed, torque and throttle for this task. Incorporating these metrics can help model improve performance and their values can also be predicted moving the model closer to a fully functional self-driving car model. One can also experiment with deep reinforcement learning to solve this problem. Because of the nature of the input, which is a series on consequence images about the same environment evolving by time. Using recurrent neural network or RNN with memory unit can be useful.

Generate adversarial models, GANs, could be used to augment data set by generating different conditions on track while driving. For example, GAN could be used to generate images on track during adverse weather conditions. To simulate driving in real world with minimal error, there is substantial research that still needs to be done on the subject before models like these can be deployed widely to transport the public

50

1. **Integrate the model into a real car**: While the current project is a proof-of-concept, the next step could be to integrate the model into a real car and test it under real-world conditions.
2. **Develop a user interface**: In order for the model to be useful to drivers, it needs to be presented in a user-friendly way. You could develop a graphical user interface (GUI) that displays the predicted steering angle and other relevant information to the driver.
3. **Incorporate additional sensor data**: While the current model only uses camera data, other sensors such as lidar, radar, or ultrasonic sensors could be used to improve the accuracy of the model and make it more robust in different driving scenarios.
4. **Investigate the model's interpretability**: Deep learning models can be notoriously difficult to interpret, but understanding how the model is making its predictions could be important for safety and regulatory purposes. You could investigate techniques for interpreting the model's predictions and visualizing its internal representations.
5. **Consider transfer learning**: Transfer learning involves using a pre-trained model as a starting point for training a new model. You could investigate the potential benefits of transfer learning in this project by using a pre-trained model on a related task, such as object detection, as a starting point for training the autopilot model.
6. **Implement data augmentation techniques**: Data augmentation involves applying transformations to the training data in order to increase its diversity and improve the model's ability to generalize to new data. You could experiment with different data augmentation techniques, such as image rotation or flipping, to see if they improve the accuracy of the model.
7. **Develop a more robust training pipeline**: Currently, the model is trained using a simple script. However, a more robust training pipeline could make it easier to experiment with different architectures and hyperparameters, and could also allow for distributed training across multiple GPUs or nodes.

**REFERENCES**

[1] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). <https://arxiv.org/abs/1511.07289>.

[2] C. Doersch. Tutorial on variational autoencoders. https: //arxiv.org/abs/1606.05908

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. <https://arxiv.org/abs/1502.03167>.

[5] NVIDIA. End to end learning for self-driving cars. https: //arxiv.org/abs/1604.07316.

[6] D. A. Pomerleau. Alvinn, an autonomous land vehicle in a neural network. http://repository.cmu. edu/cgi/viewcontent.cgi?article=2874& context=compsci.

[7] A. E. Sallab, M. Abdou, et al. Deep reinforcement learning framework for autonomous driving. https://arxiv. org/abs/1704.02532.

[8] E. Santana and G. Hotz. Learning a driving simulator. <https://arxiv.org/abs/1608.01230>.

[9] S. Shalev-Shwartz, S. Shammah, et al. Safe, multi-agent, reinforcement learning for autonomous driving. https:// arxiv.org/abs/1610.03295.

[10] N. Srivastava, G. Hinton, et al. Dropout: A simple way to prevent neural networks from overfitting. https://www.cs.toronto.edu/˜hinton/ absps/JMLRdropout.pdf.

[11] L. Torrey and J. Shavlik. Transfer learning. http:// citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.146.1515&rep=rep1&type=pdf.

**DRIVING DATASET CODE**

import cv2

import random

#import numpy as np

xs = []

ys = []

#points to the end of the last batch

train\_batch\_pointer = 0

val\_batch\_pointer = 0

#read data.txt

with open("driving\_dataset/data.txt") as f:

for line in f:

xs.append("driving\_dataset/" + line.split()[0])

#the paper by Nvidia uses the inverse of the turning radius,

#but steering wheel angle is proportional to the inverse of turning radius

#so the steering wheel angle in radians is used as the output

ys.append(float(line.split()[1]) \* 3.14159265 / 180)

#get number of images

num\_images = len(xs)

#shuffle list of images

c = list(zip(xs, ys))

random.shuffle(c)

xs, ys = zip(\*c)

train\_xs = xs[:int(len(xs) \* 0.8)]

train\_ys = ys[:int(len(xs) \* 0.8)]

val\_xs = xs[-int(len(xs) \* 0.2):]

val\_ys = ys[-int(len(xs) \* 0.2):]

num\_train\_images = len(train\_xs)

num\_val\_images = len(val\_xs)

def LoadTrainBatch(batch\_size):

global train\_batch\_pointer

x\_out = []

y\_out = []

for i in range(0, batch\_size):

x\_out.append(cv2.resize(cv2.imread(train\_xs[(train\_batch\_pointer + i) % num\_train\_images])[-150:], (200, 66)) / 255.0)

y\_out.append([train\_ys[(train\_batch\_pointer + i) % num\_train\_images]])

train\_batch\_pointer += batch\_size

return x\_out, y\_out

def LoadValBatch(batch\_size):

global val\_batch\_pointer

x\_out = []

y\_out = []

for i in range(0, batch\_size):

x\_out.append(cv2.resize(cv2.imread(val\_xs[(val\_batch\_pointer + i) % num\_val\_images])[-150:], (200, 66)) / 255.0)

y\_out.append([val\_ys[(val\_batch\_pointer + i) % num\_val\_images]])

val\_batch\_pointer += batch\_size

return x\_out, y\_out

**APPENDIX**

**DATASET WORKING**

import tensorflow.compat.v1 as tf

tf.disable\_v2\_behavior()

import model

import cv2

from subprocess import call

import os

#check if on windows OS

windows = False

if os.name == 'nt':

windows = True

sess = tf.InteractiveSession()

saver = tf.train.Saver()

saver.restore(sess, "save/model.ckpt")

img = cv2.imread('steering\_wheel\_image.jpg',0)

rows,cols = img.shape

smoothed\_angle = 0

i = 0

while(cv2.waitKey(10) != ord('q')):

full\_image = cv2.imread("driving\_dataset/" + str(i) + ".jpg")

image = cv2.resize(full\_image[-150:], (200, 66)) / 255.0

degrees = model.y.eval(feed\_dict={model.x: [image], model.keep\_prob: 1.0})[0][0] \* 180.0 / 3.14159265

if not windows:

call("clear")

print("Predicted steering angle: " + str(degrees) + " degrees")

cv2.imshow("frame", full\_image)

#make smooth angle transitions by turning the steering wheel based on the difference of the current angle

#and the predicted angle

smoothed\_angle += 0.2 \* pow(abs((degrees - smoothed\_angle)), 2.0 / 3.0) \* (degrees - smoothed\_angle) / abs(degrees - smoothed\_angle)

M = cv2.getRotationMatrix2D((cols/2,rows/2),-smoothed\_angle,1)

dst = cv2.warpAffine(img,M,(cols,rows))

cv2.imshow("steering wheel", dst)

i += 1

cv2.destroyAllWindows()

def LoadValBatch(batch\_size):

global val\_batch\_pointer

x\_out = []

y\_out = []

for i in range(0, batch\_size):

x\_out.append(cv2.resize(cv2.imread(val\_xs[(val\_batch\_pointer + i) % num\_val\_images])[-150:], (200, 66)) / 255.0)

y\_out.append([val\_ys[(val\_batch\_pointer + i) % num\_val\_images]])

val\_batch\_pointer += batch\_size

return x\_out, y\_out

**MODEL**

import tensorflow.compat.v1 as tf

tf.disable\_v2\_behavior()

#import scipy

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1)

return tf.Variable(initial)

def bias\_variable(shape):

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

def conv2d(x, W, stride):

return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding='VALID')

x = tf.placeholder(tf.float32, shape=[None, 66, 200, 3])

y\_ = tf.placeholder(tf.float32, shape=[None, 1])

x\_image = x

#first convolutional layer

W\_conv1 = weight\_variable([5, 5, 3, 24])

b\_conv1 = bias\_variable([24])

h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1, 2) + b\_conv1)

#second convolutional layer

W\_conv2 = weight\_variable([5, 5, 24, 36])

b\_conv2 = bias\_variable([36])

h\_conv2 = tf.nn.relu(conv2d(h\_conv1, W\_conv2, 2) + b\_conv2)

#third convolutional layer

W\_conv3 = weight\_variable([5, 5, 36, 48])

b\_conv3 = bias\_variable([48])

h\_conv3 = tf.nn.relu(conv2d(h\_conv2, W\_conv3, 2) + b\_conv3)

#fourth convolutional layer

W\_conv4 = weight\_variable([3, 3, 48, 64])

b\_conv4 = bias\_variable([64])

h\_conv4 = tf.nn.relu(conv2d(h\_conv3, W\_conv4, 1) + b\_conv4)

#fifth convolutional layer

W\_conv5 = weight\_variable([3, 3, 64, 64])

b\_conv5 = bias\_variable([64])

h\_conv5 = tf.nn.relu(conv2d(h\_conv4, W\_conv5, 1) + b\_conv5)

#FCL 1

W\_fc1 = weight\_variable([1152, 1164])

b\_fc1 = bias\_variable([1164])

h\_conv5\_flat = tf.reshape(h\_conv5, [-1, 1152])

h\_fc1 = tf.nn.relu(tf.matmul(h\_conv5\_flat, W\_fc1) + b\_fc1)

keep\_prob = tf.placeholder(tf.float32)

h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)

#FCL 2

W\_fc2 = weight\_variable([1164, 100])

b\_fc2 = bias\_variable([100])

h\_fc2 = tf.nn.relu(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2)

h\_fc2\_drop = tf.nn.dropout(h\_fc2, keep\_prob)

#FCL 3

W\_fc3 = weight\_variable([100, 50])

b\_fc3 = bias\_variable([50])

h\_fc3 = tf.nn.relu(tf.matmul(h\_fc2\_drop, W\_fc3) + b\_fc3)

h\_fc3\_drop = tf.nn.dropout(h\_fc3, keep\_prob)

#FCL 3

W\_fc4 = weight\_variable([50, 10])

b\_fc4 = bias\_variable([10])

h\_fc4 = tf.nn.relu(tf.matmul(h\_fc3\_drop, W\_fc4) + b\_fc4)

h\_fc4\_drop = tf.nn.dropout(h\_fc4, keep\_prob)

#Output

W\_fc5 = weight\_variable([10, 1])

b\_fc5 = bias\_variable([1])

y = tf.multiply(tf.atan(tf.matmul(h\_fc4\_drop, W\_fc5) + b\_fc5), 2) #scale the atan output

**TRAINING THE MODEL**

import os

import tensorflow.compat.v1 as tf

tf.disable\_v2\_behavior()

from tensorflow.core.protobuf import saver\_pb2

import driving\_data

import model

LOGDIR = './save'

sess = tf.InteractiveSession()

L2NormConst = 0.001

train\_vars = tf.trainable\_variables()

loss = tf.reduce\_mean(tf.square(tf.subtract(model.y\_, model.y))) + tf.add\_n([tf.nn.l2\_loss(v) for v in train\_vars]) \* L2NormConst

train\_step = tf.train.AdamOptimizer(1e-4).minimize(loss)

sess.run(tf.global\_variables\_initializer())

# create a summary to monitor cost tensor

tf.summary.scalar("loss", loss)

# merge all summaries into a single op

merged\_summary\_op = tf.summary.merge\_all()

saver = tf.train.Saver(write\_version = saver\_pb2.SaverDef.V2)

# op to write logs to Tensorboard

logs\_path = './logs'

summary\_writer = tf.summary.FileWriter(logs\_path, graph=tf.get\_default\_graph())

epochs = 30

batch\_size = 100

# train over the dataset about 30 times

for epoch in range(epochs):

for i in range(int(driving\_data.num\_images/batch\_size)):

xs, ys = driving\_data.LoadTrainBatch(batch\_size)

train\_step.run(feed\_dict={model.x: xs, model.y\_: ys, model.keep\_prob: 0.8})

if i % 10 == 0:

xs, ys = driving\_data.LoadValBatch(batch\_size)

loss\_value = loss.eval(feed\_dict={model.x:xs, model.y\_: ys, model.keep\_prob: 1.0})

print("Epoch: %d, Step: %d, Loss: %g" % (epoch, epoch \* batch\_size + i, loss\_value))

# write logs at every iteration

summary = merged\_summary\_op.eval(feed\_dict={model.x:xs, model.y\_: ys, model.keep\_prob: 1.0})

summary\_writer.add\_summary(summary, epoch \* driving\_data.num\_images/batch\_size + i)

if i % batch\_size == 0:

if not os.path.exists(LOGDIR):

os.makedirs(LOGDIR)

checkpoint\_path = os.path.join(LOGDIR, "model.ckpt")

filename = saver.save(sess, checkpoint\_path)

print("Model saved in file: %s" % filename)

print("Run the command line:\n" \

"--> tensorboard --logdir=./logs " \

"\nThen open http://0.0.0.0:6006/ into your web browser")

if i % batch\_size == 0:

if not os.path.exists(LOGDIR):

os.makedirs(LOGDIR)

checkpoint\_path = os.path.join(LOGDIR, "model.ckpt")

filename = saver.save(sess, checkpoint\_path)

print("Model saved in file: %s" % filename)

print("Run the command line:\n" \

"--> tensorboard --logdir=./logs " \

"\nThen open http://0.0.0.0:6006/ into your web browser")