LANE AND TRAFFIC SIGNAL DETECTION FOR AUTONOMOUS VEHICLE USING DEEP LEARNING

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

B. Tech.

in

Computer Science and Engineering

By

Shubhashish Jena (Roll No. 16221A05A3)

COMPUTER SCIENCE AND ENGINEERING
BONAM VENKATA CHALAMAYYA ENGINEERING COLLEGE
ODALAREVU AP 533210

April 2020

In memory of

Late Mr Bonam Venkata Chalamayya "Founder Chairman" BVCE

Our deepest gratitude.

BONAFIDE CERTIFICATE

This is to certify that the project titled **LANE AND TRAFFIC SIGNAL DETECTION FOR AUTONOMOUS VEHICLE USING DEEP LEARNING** is a bonafide record of the work done by

Shubhashish Jena (Roll No. 16221A05A3)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of the **BONAM VENKATA CHALAMAYYA ENGINEERING COLLEGE, AUTONOMOUS, ODALAREVU**, during the year 2020-21.

Prof Gunamani Jena, ME Ph D	**Prof Gunamani Jena, ME Ph D**
Guide	Head of the Department

Project Viva-voce held on _____

Internal Examiner **External Examiner**

ABSTRACT

I trained a Convolutional Neural Network (CNN) to map raw pixels from three front-facing cameras placed on the dash-board directly to steering commands [1]. This end-to-end approach proved surprisingly powerful. With minimum training data from humans the system learns to drive in traffic on local roads with or without lane markings and on highways. It also operates in areas with unclear visual guidance such as in parking lots and on unpaved roads. The system automatically learns 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. Compared to explicit decomposition of the problem, such as lane marking detection, path planning and control, our end-to-end system optimizes all processing steps simultaneously. So it eventually leads to better performance and smaller systems. Better performance will result because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria, e.g., lane detection. Such criteria understandably are selected for ease of human interpretation which doesn't automatically guarantee maximum system performance. Smaller networks are possible because the system learns to solve the problem with the minimal number of processing steps. We used an NVIDIA DevBox model and Deep Learning algorithms using Python 3.7 for training and for determining where to drive. The system operates at 30 frames per second (FPS).

Keywords : Deep Learning, Regression, OpenCV, Keras, TenserFlow, NVidia model, Convolution 2D

ACKNOWLEDGEMENTS

I would like to express our deepest gratitude to the following people for guiding us through this course and without whom this project and the results achieved from it would not have reached completion.

Prof Gunamani Jena, ME Ph D, Professor, Department of Computer Science and Engineering, for helping us and guiding us in the course of this project. Without his guidance, I would not have been able to successfully complete this project. His patience and genial attitude is and always will be a source of inspiration to me.

Prof Gunamani Jena, ME Ph D, the Head of the Department, Department of Computer Science and Engineering, for allowing me to avail the facilities at the department.

I am thankful to the Principal Dr D S V Prasad for providing the facility in this institution.

I am also thankful to the Honorable Chairman Dr Bonam Rajesh for motivating, arranging training and internship related to the Project.

My sincere thanks to the Honorable Secretary Mr Bonam Kanakayya for extending the excellent infrastructure.

I am also thankful to the faculty and staff members of the Department of Computer Science and Engineering, my parents and my friends for their constant support and help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

1.1 Introduction

Autonomous car

A driverless [2] vehicle capable of fulfilling the main transportation capabilities of a traditional car. The usage and production of these cars has become a leading industry in almost every area of the world. According to the forecasts, the world's car stock will reach two billion by 2021. Over the years and centuries, this industry has gone through enormous development, as the first vehicles were only powered by steam engine, then petrol and diesel came to public mind and currently it seems that the electric propulsion will be the future. Of course, with this development, faster and more useful vehicles can be produced, but in our accelerated world with more and more cars, unfortunately the numbers of accidents have increased. In most cases, these accidents are the fault of the driver, therefore it could be theoretically replaceable with the help of self-propelled cars. Human presence is the most important part in transport at present, although there are many areas where you can use a tool or feature that helps people achieve greater efficiency. Some examples for these features are the autopilot on aircraft, the cruise control in cars, and many other tools that help decision-making. In this study, we will provide a brief summary about the development of self-driving or at least driver assisting devices

1.1.1 Evolution of Self-Driving Cars

Autonomous cars are those vehicles which are driven by digital technologies without any human intervention. They are capable of driving and navigating themselves on the roads by sensing the environmental impacts. Their appearance is designed to occupy less space on the road in order to avoid traffic jams and reduce the likelihood of accidents. Although the progression is gigantic, in 2017, allowed automated cars on public roads are not fully autonomous: each one needs a human driver who notices when it is necessary to take back the control over the vehicle. The dream of self-propelled [3] cars goes back to the Middle Ages, centuries before the invention of the car. An evidence for this statement comes from sketches of Leonardo De Vinci, in which he made a rough plan of them. Later, in literature and in several science fiction novels, the robots and the vehicles controlled by them,

appeared. The first driverless cars [4] were prototyped in the 1920s, but they looked different than they are today. Although the ”driver” was nominally lacking, these vehicles relied heavily on specific external inputs. One of these solutions is when the car is controlled by another car behind it. Its prototype was introduced in New York and Milwaukee known as the American Wonder” or ”Phantom Auto”.

Most of the big names [5] – Mercedes Benz, Audi, BMW, Tesla, Hyundai etc. have begun developing or forming partnerships around autonomous technology. They invested sizable resources into this, and by making this step they wanted to be leaders at the market of self-driving cars. Up to this point, numerous aids, software and sensors have been put into these cars, but we are still far from full autonomy. They use lasers that are testing the environment with the help of LIDAR ((Light Detection and Ranging). This optical technology senses the shape and movement of objects around the car; combined with the digital GPS map of the area, they detect white and yellow lines on the road, as well as all standing and moving objects on their perimeter. Autonomous vehicles can only drive themselves if the human driver can take over the control if needed.

1.1.2 Features of Driver less car

These are those features that driver less cars already use:

- Collision Avoidance
- Drifting Warning
- Blind-spot Detectors
- Enhanced cruise control
- Self-Parking

1.1.3 Levels of Autonomous Vehicle [6]

The National Highway Traffic Safety Administration (NHTSA) adopted the levels of the Society of Automotive Engineers(SAE) for automated driving systems, which provides a broad spectrum of total human participation to total autonomy. NHTSA expects automobile manufacturers to classify each vehicle in the coming years using SAE 0 to 5 levels [7]. These are the levels of SAE:

Level 0: No Automation

In this case, there is 100% of human presence. Acceleration, braking and steering are constantly controlled by a human driver, even if they support warning sounds or

safety intervention systems. This level also includes automated emergency braking.

Level 1: Driver Assistance

The computer never controls steering and accelerating or braking simultaneously. In certain driving modes, the car can take control of the steering wheel or pedals. The best examples for the first level are adaptive cruise control and parking assistance.

Level 2: Partial Automation

The driver can take his hands off the steering wheel. At this level, there are set-up options in which the car can control both pedals and the steering wheel at the same time, but only under certain circumstances. During this time the driver has to pay attention and if it is necessary, intervene. This is what Tesla Autopilot has known since 2014.

Level 3: Conditional Automation

It approaches full autonomy, but this is dangerous in terms of liability, therefore, paying attention to them is a very important element. Here the car has a certain mode that can take full responsibility for driving in certain circumstances, but the driver must take the control back when the system asks. At this level, the car can decide when to change lanes and how to respond to dynamic events on the road and it uses the human driver as a backup system.

Level 4: High Automation

It is similar to the previous level, but it is much safer. The vehicle can drive itself under suitable circumstances, and it does not need human intervention. If the car meets something that it cannot handle, it will ask for human help, but it will not endanger passengers if there is no human response. These cars are close to the fully self-driving car.

Level 5: Full Automation

At this level, as the car drives itself, human presence is not a necessity, only an opportunity. The front seats can turn backwards so passengers can talk more easily with each other, because the car does not need help in driving. All driving tasks are performed by the computer on any road under any circumstances, whether there's a human on board or not. These levels are very useful as with these we can keep track of what happens when we move from human-driven cars to fully automated

ones. This transition will have enormous consequences for our lives, our work and our future travels. As autonomous driving options are widespread, the most advanced detection, vision and control technologies allow cars to detect and monitor all objects around the car, relying on real-time object measurements. In addition, the information technology built into the vehicle is fully capable of delivering both external (field) and internal (machine) information to the car

1.2 Introduction of CNN

CNNs [8] have revolutionized pattern recognition [9]. Prior to the widespread adoption of CNNs, 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 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 [10], their adoption has exploded in the last few years because of two recent developments. First, large labeled data sets such as the Large Scale Visual Recognition Challenge (ILSVRC) [11] 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.

In this project, we describe a CNN that goes beyond pattern recognition. It learns the entire processing pipeline needed to steer an automobile. The original work was done over 14 years ago in a Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE) [12] in which a sub-scale radio control (RC) car drove through a junk-filled alley way. DAVE was trained on hours of human driving in similar, but not identical environments. The training data included video from two cameras coupled with left and right steering commands from a human operator. In many ways, DAVE-2 was inspired by the pioneering work of Pomerleau [13] who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. It demonstrated that an end-to-end trained neural network can indeed steer a car on public roads.

While DAVE demonstrated the potential of end-to-end learning, and indeed was used to justify starting the DARPA Learning Applied to Ground Robots (LAGR) program [14], DAVE's performance was not sufficiently reliable to provide a full

alternative to more modular approaches to off-road driving. DAVE's mean distance between crashes was about 20 meters in complex environments. A new effort was started at NVIDIA that sought to build on DAVE and create a robust system for driving on public roads. The primary motivation for this work is to avoid the need to recognize specific human-designated features, such as lane markings, guard rails, or other cars, and to avoid having to create a collection of "if, then, else" rules, based on observation of these features [15]. This project describes preliminary results of this new effort.

1.2.1 Overview of the DAVE-2 System

Figure 1.1 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) bus.

In order to make our system independent of the car geometry, we represent the steering command as 1/r where r is the turning radius in meters. We use 1/r instead of r to prevent a singularity when driving straight (the turning radius for driving straight is infinity). 1/r smoothly transitions through zero from left turns (negative values) to right turns (positive values). Training data contains single images sampled from the video, paired with the corresponding steering command (1/r). 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 center of the lane and rotations from the direction of the road.

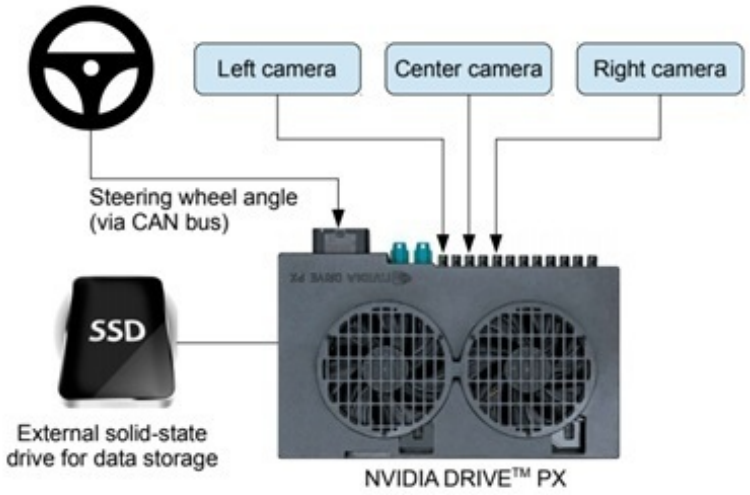

Figure 1.1: Data Collection System

Images for two specific off-center 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. A block diagram of our training system is shown in Figure 1.2.

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 algorithm.

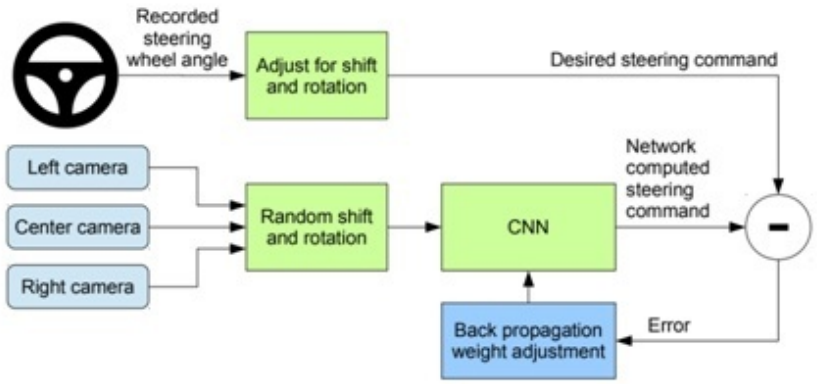

Figure 1.2: Block diagram for Training Neural Network

Once trained, the network can generate steering from the video images of a single center camera. This configuration is shown in Figure 1.3.

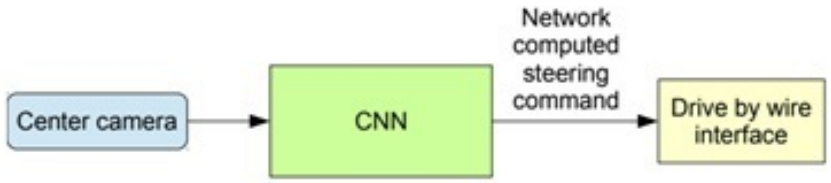

Figure 1.3: Training Network for Steering Command

1.3 Modules

1. Lane Detection
2. Traffic sign classification
3. Deep Learning training using Convolution Neural Network (Nvidia model).
4. Vehicle behavioral cloning
5. Simulation on Open source self driving car simulator.

1.3.1 Lane Detection

Packages used:

OPENCV, NumPy

Capture any real time or stored video and convert it into gray-scale. Go for preprocessing (applying Gaussian filter to reduce the noise of the video frames and detect sharp contrasting zones).

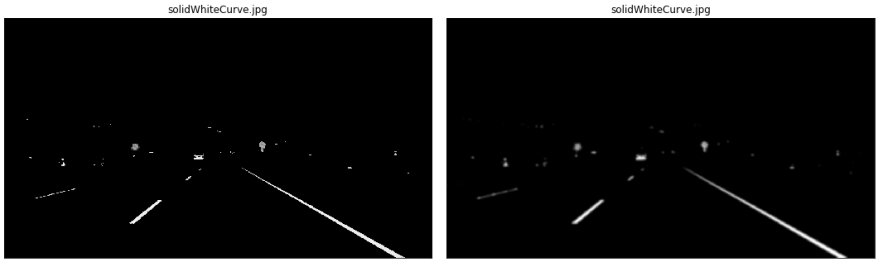

Figure 1.4: Gray-scale to Gauusian Blurr

Then we go for Canny Edge Detection implementation on our Gaussian Blurred image to highlight the edges in the image. The image is shown below :

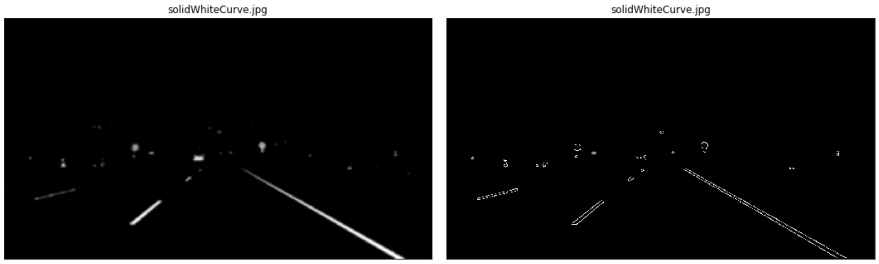

Figure 1.5: Gauusian blurr gray-scale to Canny Image

Remove top three fifth in order to remove the environmental background details like sky trees hills mountain building etc.
Imagine and create a central triangular structure displaying the lane as it is region of our interest.

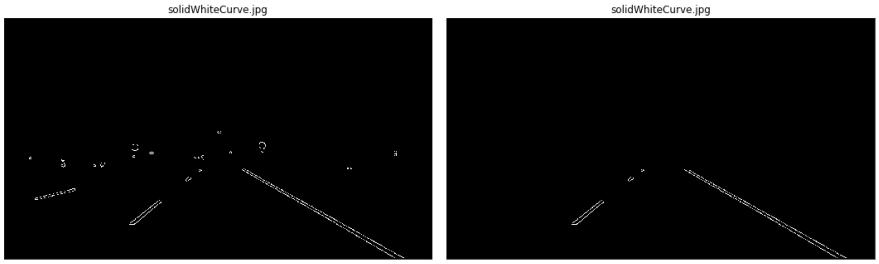

Figure 1.6: Canny vs Segmented Canny Image (Region of Interest)

Apply Houghline transformation for edge detection and detect lane on which we are driving.

OpenCV

OpenCV-Python is a library of Python bindings designed to solve computer vision problems.

OpenCV is a vast library that helps in providing various functions for image and video operations. With OpenCV, we can capture a video from the camera. It lets you create a video capture object which is helpful to capture videos through webcam and then you may perform desired operations on that video.

Steps to capture a video:
Use cv2.VideoCapture() to get a video capture object for the camera. Set up an infinite while loop and use the read() method to read the frames using the above created object.
Use cv2.imshow() method to show the frames in the video. Breaks the loop when the user clicks a specific key.

NumPy

Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.
NumPy is the fundamental package for scientific computing with Python.Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Gaussian Filtering

In this approach, instead of a box filter consisting of equal filter coefficients, a Gaussian kernel is used. It is done with the function, cv2.GaussianBlur(). We should specify the width and height of the kernel which should be positive and odd. We also should specify the standard deviation in the X and Y directions, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as equal to sigmaX. If both are given as zeros, they are calculated from the kernel size. Gaussian filtering is highly effective in removing Gaussian noise from the image.

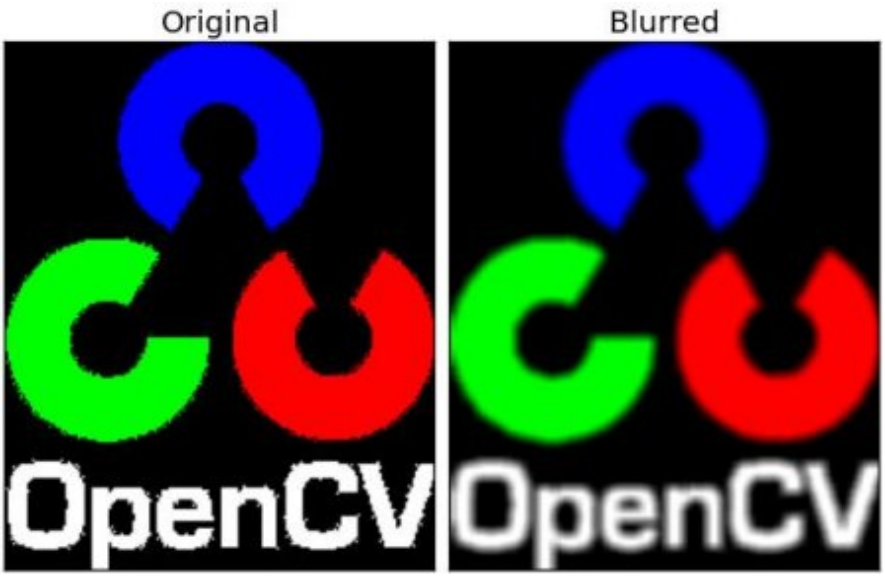

Figure 1.7: Gauusian Blurr

The Hough Transform is a method that is used in image processing to detect any shape, if that shape can be represented in mathematical form. It can detect the shape even if it is broken or distorted a little bit.
We will see how Hough Transform works for line detection using the HoughLine Transform method. To apply the Houghline method, first an edge detection of the specific image is desirable.

A line can be represented as

$$y = mx + c \tag{1.1}$$

or in parametric form, as

$$r = x\cos\theta + y\sin\theta \tag{1.2}$$

where r is the perpendicular distance from origin to the line, and θ is the angle formed by this perpendicular line and horizontal axis measured in counter-clockwise (That direction varies on how you represent the coordinate system. This representation is used in OpenCV). So any line can be represented in these two terms (r, θ).

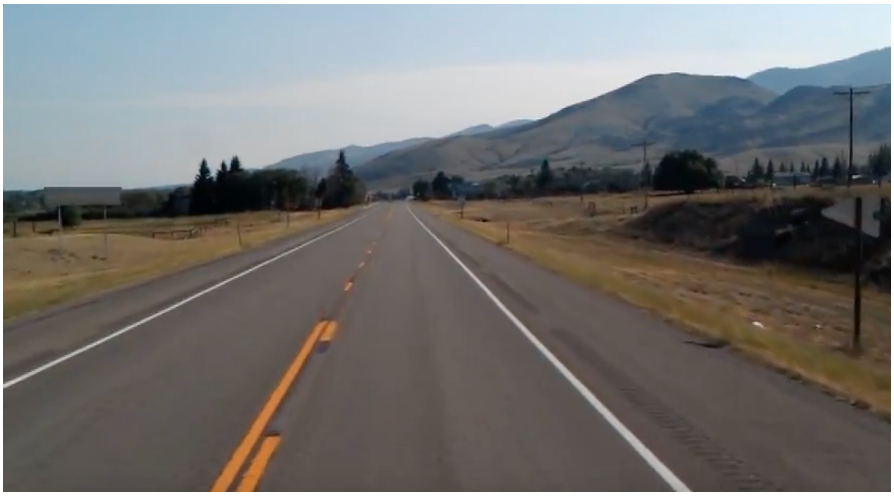

Figure 1.8: Test Image for Lane Detection

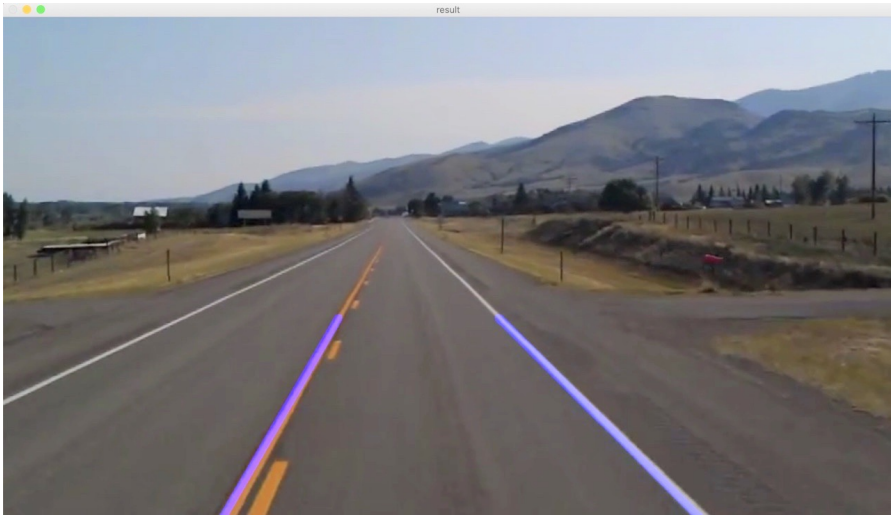

Figure 1.9: Test Image for Lane Detection

CHAPTER 2

2.1 Traffic sign classification

2.1.1 Introduction

Packages used:
NumPy, MatPlotLib, Keras(sequential, dense, Adam optimizer, To_categorical, dropout, flatten, Conv2D, Maxpulling2D), Pickle, Pandas, Random.

We have taken 43 category of traffic signs dataset from https://bitbucket.org/jadslim/german-traffic-signs [16]. Data set contains data elements of classes ranging from 180-2010 making total of 52000 data elements approximately which are labeled in sign-names.csv in the same traffic sign repository.
We then classified the data elements into three categories namely: training set, test set, and validation set of sizes 32*32 pixels. We then preprocess the dataset using cv2 and grayscale functions to transform the image from color to grayscale.

We used histogram equalization function to improve the overall contrast of our image in order to make the image clear.
Since some of the classes in our dataset consists of less number of data element we use the image data generator from keras.preprocessing.image module to randomly generate few more data elements by augmenting the existing data elements using zoom, shear, rotation , height and width shifting operations to make the data classes uniformly distributed.
We upgraded the Lenet model to modified model using sequential, conv2D, dropout layers with the relu activation function and softmax activation function and finally using Adam optimizer with a learning rate of 0.001.

We also added a dropout layer removing 40% of the redundant elements so that the model does not memorise the images rather than extracting the features. We define the loss as categorical_crossentropy and metrics as accuracy. Now we summarize the model to get the final list of parameters and trainable parameters.
We now go for traing the model using fit_generator with datagen.flow function taking batch_size = 50, steps_per_epoch = 2000, num_of_epocs = 10 and validation_set = 20% size of test data inorder to attain the best possible accuracy of 98.5% We

then plot the loss and accuracy graphs of the training and validation data. Now the algorithm is ready to test any given image and classify it.

2.1.2 Detail Discussion

Traffic signs are an integral part of our road infrastructure. They provide critical information, sometimes compelling recommendations, for road users, which in turn requires them to adjust their driving behaviour to make sure they adhere with whatever road regulation currently enforced. Without such useful signs, we would most likely be faced with more accidents, as drivers would not be given critical feedback on how fast they could safely go, or informed about road works, sharp turn, or school crossings ahead. In our modern age, around 1.3M people die on roads each year. This number would be much higher without our road signs.

Naturally, autonomous vehicles must also abide by road legislation and therefore recognize and understand traffic signs.

Traditionally, standard computer vision methods were employed to detect and classify [17] traffic signs, but these required considerable and time-consuming manual work to handcraft important features in images. Instead, by applying deep learning to this problem, we create a model that reliably classifies traffic signs, learning to identify the most appropriate features for this problem by itself. In this post, I show how we can create a deep learning architecture that can identify traffic signs with close to 98.5% accuracy on the test set.

Project Setup

The dataset is split into training, test and validation sets, with the following characteristics:
→ Images are 32 (width) x 32 (height) x 3 (RGB color channels)
→ Training set is composed of 34799 images
→ Validation set is composed of 4410 images
→ Test set is composed of 12630 images
There are 43 classes (e.g. Speed Limit 20km/h, No entry, Bumpy road, etc.)
Moreover, we used Python 3.7 with Tensorflow to develop our code.

Figure 2.1: Sample image of the data-set

We used a very popular data set from bit-bucket consisting of the above criteria (https://bitbucket.org/jadslim/german-traffic-signs). This data-set is widely used as a world-wide standard for traffic-sign classification. It was proposed as the benchmark proposal at the IEEE International Joint Conference for Neural Networks (IJCNN) 2013.

Images and Distribution

We can see below a sample of images from the data-set, with labels displayed above the row of corresponding images. Some of them are quite dark so we will look to improve contrast a bit later.

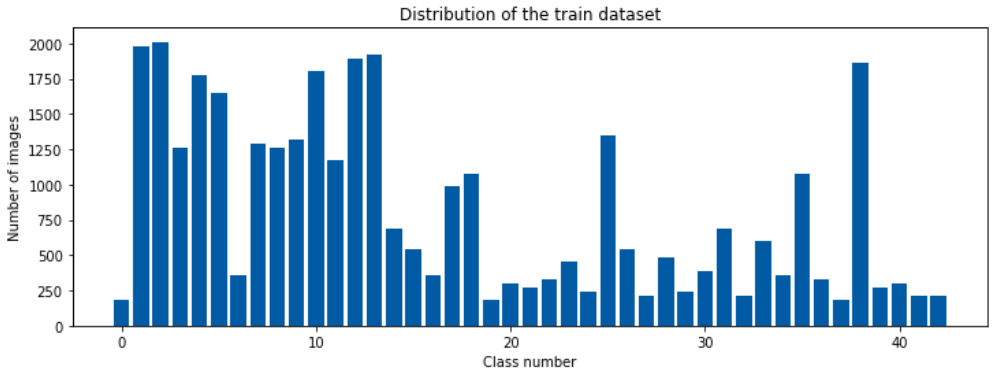

Figure 2.2: Distribution of images in the data-set

Sample of Training Set Images With Labels

There is also a significant imbalance across classes in the training set, as shown in the histogram below. Some classes have less than 200 images, while others have over 2000. This means that our model could be biased towards over-represented classes, especially when it is unsure in its predictions. We will see later how we can mitigate this discrepancy using data augmentation.

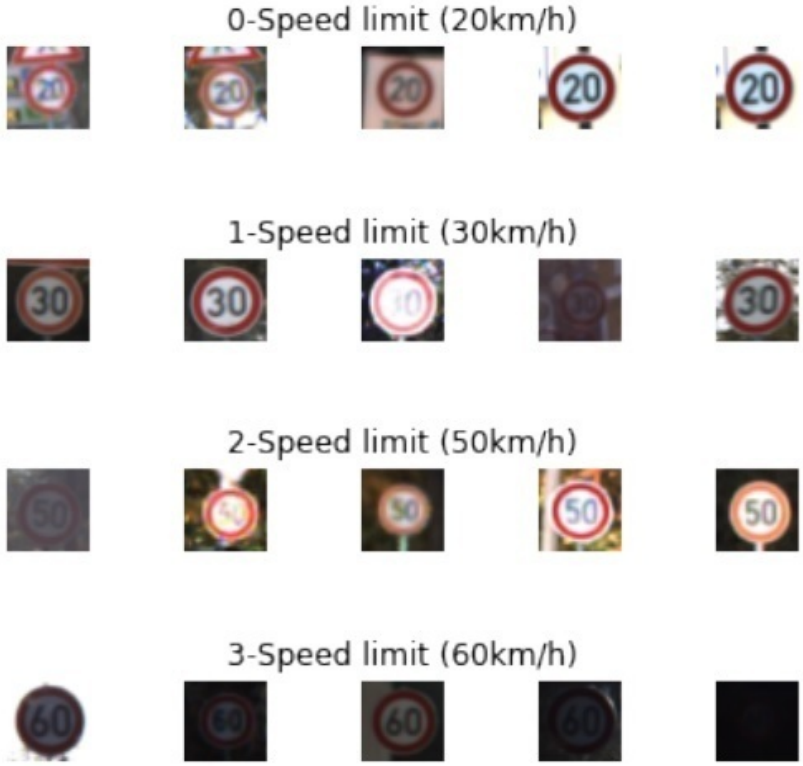

Figure 2.3: Test Image for Lane Detection

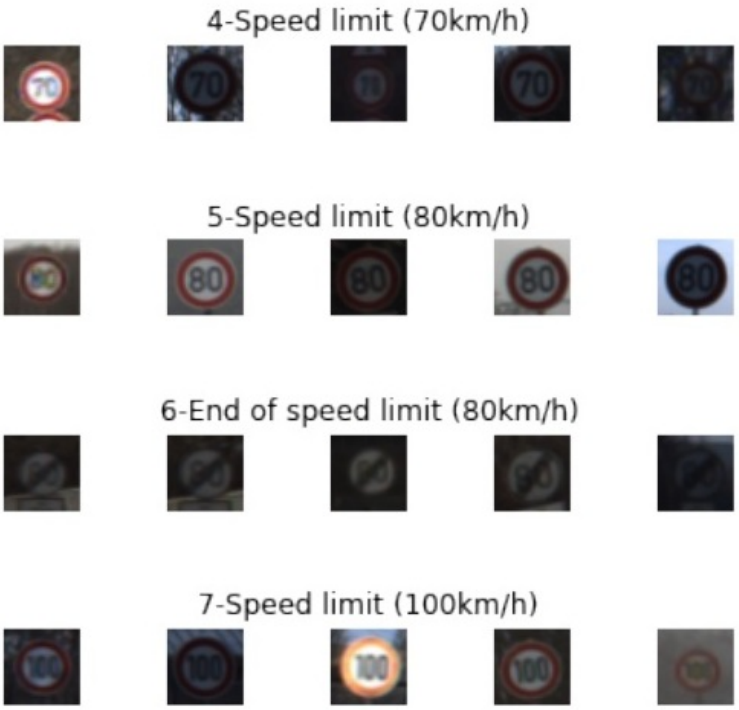

Figure 2.4: Test Image for Lane Detection

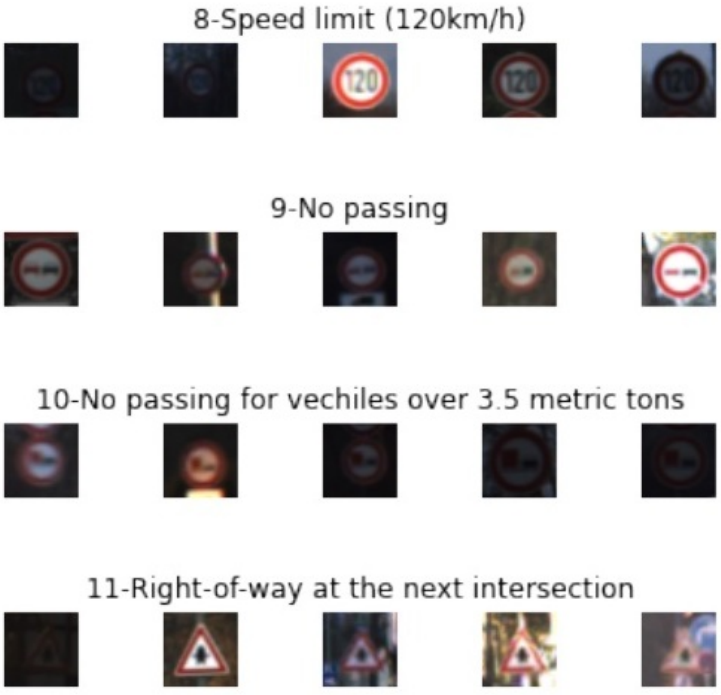

Figure 2.5: Test Image for Lane Detection

Figure 2.6: Test Image for Lane Detection

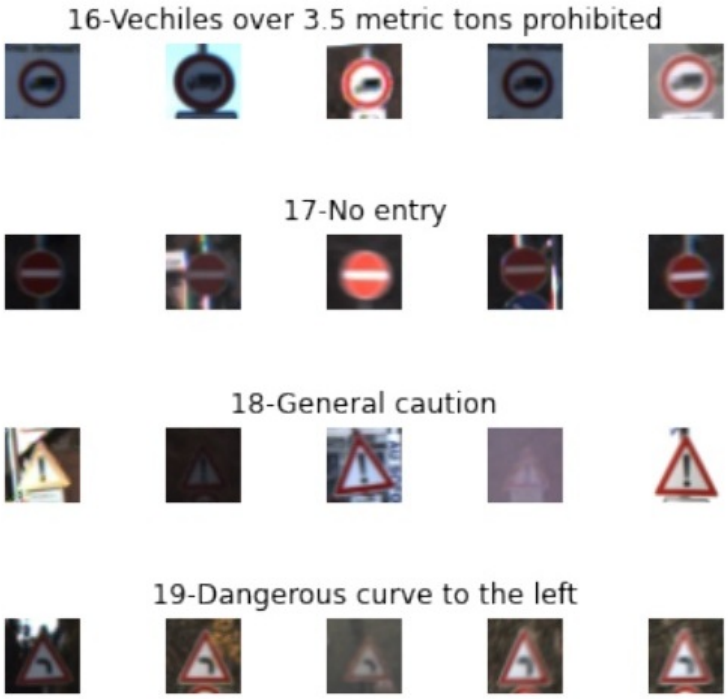

Figure 2.7: Test Image for Lane Detection

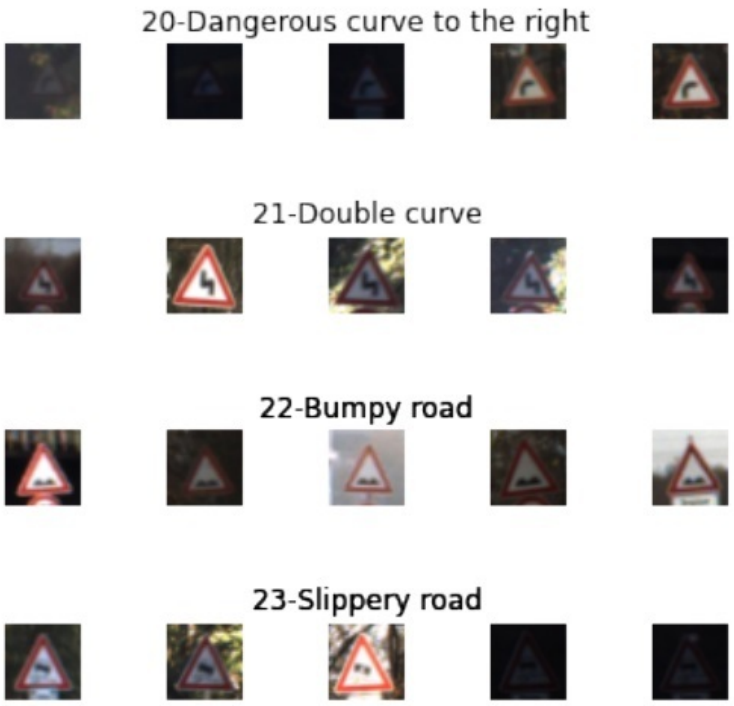

Figure 2.8: Test Image for Lane Detection

Figure 2.9: Test Image for Lane Detection

Figure 2.10: Test Image for Lane Detection

Figure 2.11: Test Image for Lane Detection

Figure 2.12: Test Image for Lane Detection

Figure 2.13: Test Image for Lane Detection

Pre-Processing Steps

We initially apply two pre-processing steps to our images:

Grayscale :

We convert our 3 channel image to a single grayscale image (we do the same thing in chapter 1 — Lane Line Detection)

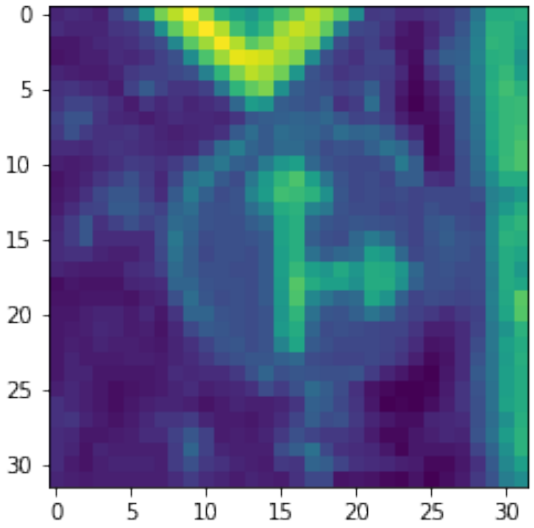

Figure 2.14: Transforming image to gray scale

.

Image Normalisation

We center the distribution of the image dataset by subtracting each image by the dataset mean and divide by its standard deviation. This helps our model treating images uniformly. The resulting images look as follows:

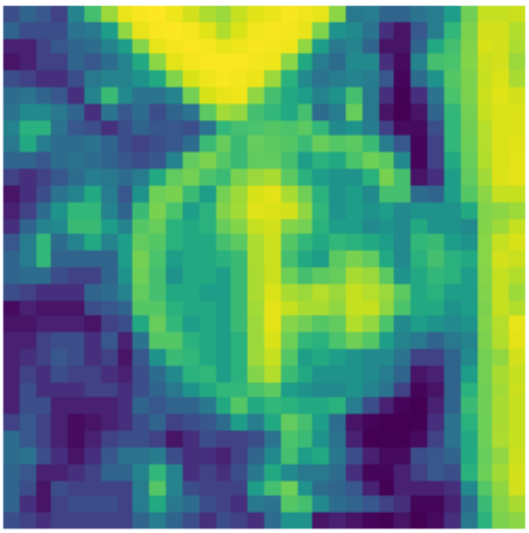

Figure 2.15: Histogram Equalization of image

Normalised images — we can see how “noise” is distributed

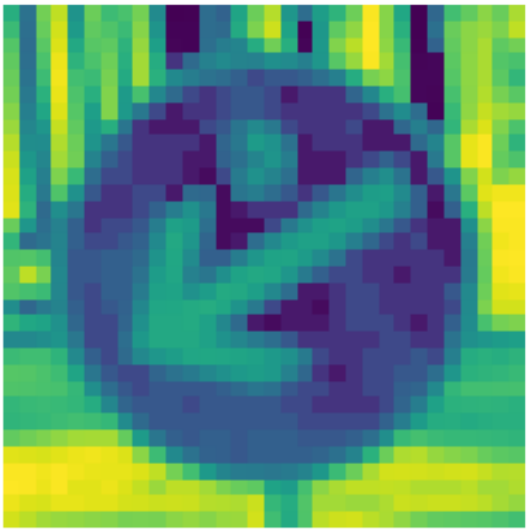

Figure 2.16: Sample of preprocessed image

Model Architecture

The architecture proposed is inspired from Yann Le Cun's paper on classification of traffic signs. We added a few tweaks and created a modular codebase which allows us to try out different filter sizes, depth, and number of convolution layers, as well as the dimensions of fully connected layers. In homage to Le Cun, and with a touch of cheekiness, we called such network LeNet. We mainly tried 5x5 and 3x3 filter (aka kernel) sizes, and start with depth of 32 for our first convolutional layer. LeNet's 3x3 architecture is shown below:

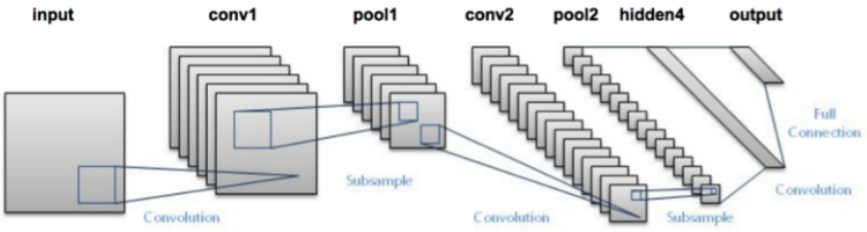

Figure 2.17: Architecture of LeNet Model

LeNet 3x3 Architecture : The network is composed of 3 convolutional layers — kernel size is 3x3, with depth doubling at next layer — using ReLU as the activation function, each followed by a 2x2 max pooling operation. The last 3 layers are fully connected, with the final layer producing 43 results (the total number of possible labels) computed using the SoftMax activation function. The network is trained using mini-batch stochastic gradient descent with the Adam optimizer. We build a highly modular coding infrastructure that enables us to dynamically create our models. We modified the existing model in order to improve the performance [18]

and accuracy of the model. We have shown the model here,

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 28, 28, 60)        1560
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 24, 24, 60)        90060
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 60)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 10, 10, 30)        16230
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 8, 8, 30)          8130
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 30)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 480)               0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               240500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_4 (Dense)              (None, 43)                21543
=================================================================
Total params: 378,023
Trainable params: 378,023
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 2.18: Modified LeNet Model Architecture

Dropout

In order to improve the model reliability, we turned to dropout, which is a form of regularisation where weights are kept with a probability p: the unkept weights are thus “dropped”. This prevents the model from overfitting.

Histogram Equalization

Histogram Equalization is a computer vision technique used to increase the contrast in images. As some of our images suffer from low contrast (blurry, dark), we will improve visibility by applying OpenCV’s Contrast Limiting Adaptive Histogram Equalization (aka CLAHE) function.

Data Augmentation

We observed earlier that the data presented glaring imbalance across the 43 classes. Yet it does not seem to be a crippling problem as we are able to reach very high accuracy despite the class imbalance. We also noticed that some images in the test set are distorted. We are therefore going to use data augmentation techniques in an attempt to:

→ Extend dataset and provide additional pictures in different lighting settings and orientations

→ Improve model's ability to become more generic

→ Improve test and validation accuracy, especially on distorted images

We use a nifty library called imgaug(i.e., ImageDataGenerator) to create our augmentations. We mainly apply affine transformations to augment the images.

While the class imbalance probably causes some bias in the model, we have decided not to address it at this stage as it would cause our dataset to swell significantly and lengthen our training time (we don't have a lot of time to spend on training at this stage). Instead, we decided to augment each class by 10%. Our new dataset looks as follows.

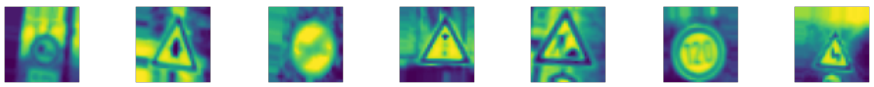

Figure 2.19: Augumented set of images

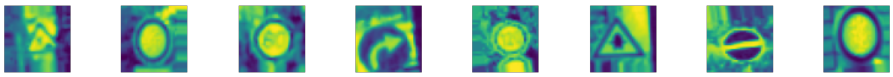

Figure 2.20: Augumented set of images

The distribution of images does not change significantly of course, but we do apply grayscale, histogram equalization and normalisation pre-processing steps to our images.

Testing On New Images

We decided to test our model on new images as well, to make sure that it's indeed generalised to more than the traffic signs in our original dataset. We therefore downloaded a new image and submitted it to our model for prediction.

Figure 2.21: Random image for testing from internet

.

The Image was chosen because of the following:

→ It represent different traffic sign that we currently classify

→ It vary in shape and color

→ It is under different lighting condition

→ It is under different orientation

→ It has different background, etc.

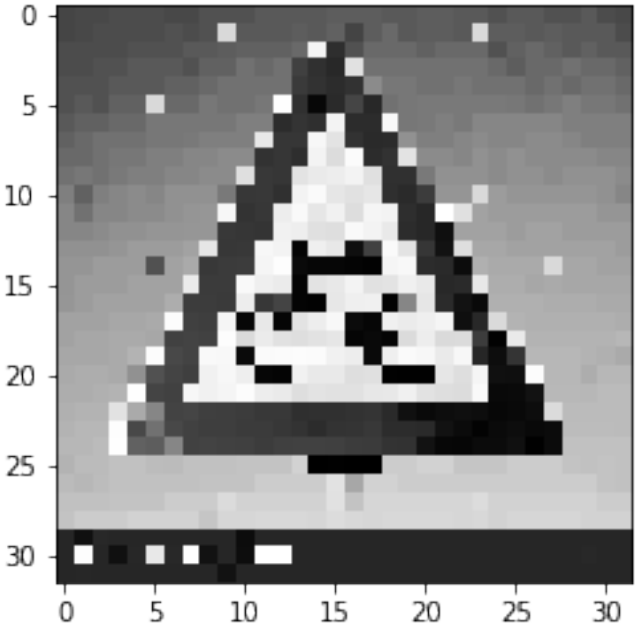

Figure 2.22: Preprocessed Test image

.

We get the result of or test image as

Predicted Sign: [29]- Bicycles crossing

with the following scores

Test score: 0.12626752376309303

Test accuracy: 0.9714172605380876

Visualizing Our Activation Maps We show below the results produced by each convolutional layer (before max pooling), resulting in 3 activation maps.

Layer 1 :

We can see that the network is focusing a lot on the edges of the circle and somehow on the truck. The background is mostly ignored.

Layer 2 :

Activation Map Of Second Convolutional Layer.

It is rather hard to determine what the network is focusing on in layer 2, but it seems to "activate" around the edges of the circle and in the middle, where the truck appears.

Layer 3 :

This activation map is also hard to decipher. But it seems the network reacts to stimuli on the edges and in the middle once again.

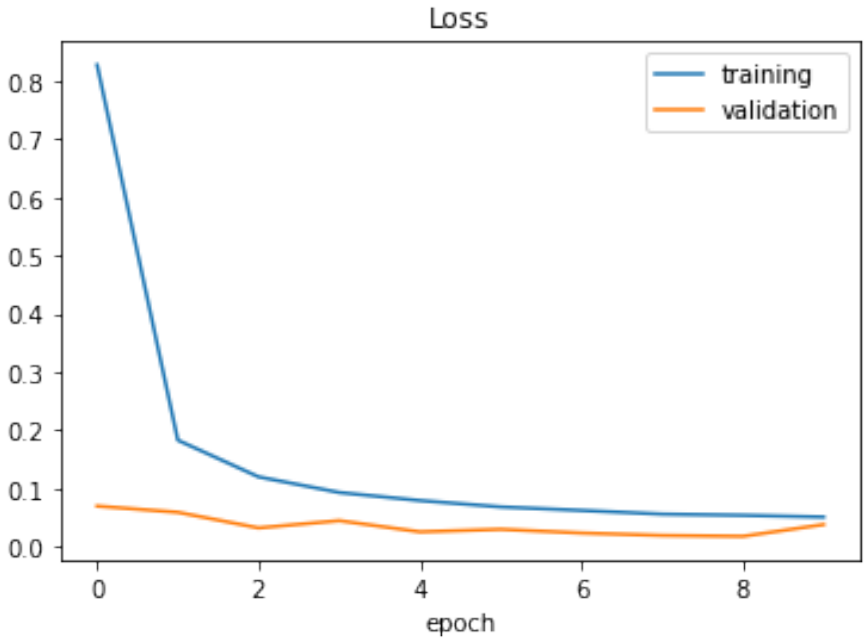

Figure 2.23: Loss Plot of our model during training.

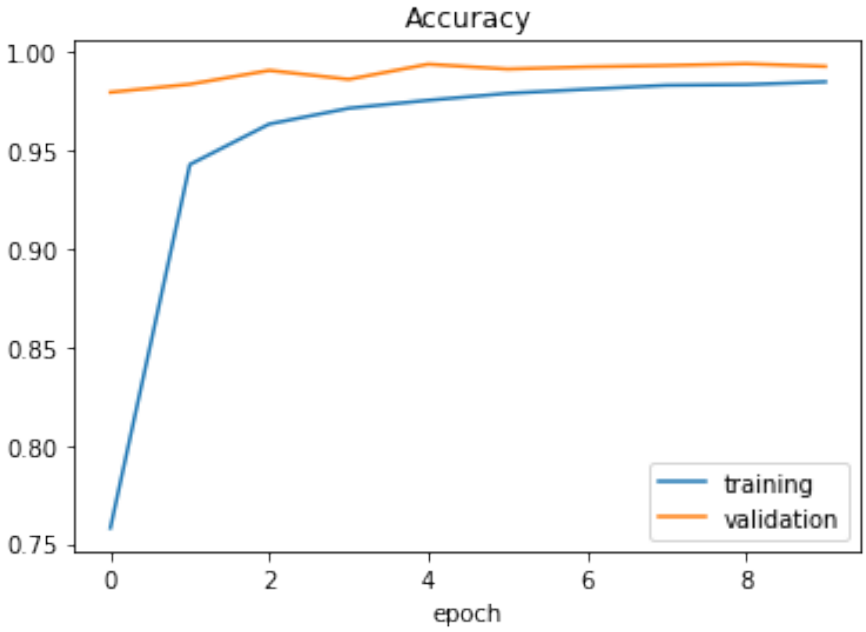

Figure 2.24: Accuracy Plot of our model during training.

2.1.3 Conclusion

We covered how deep learning can be used to classify traffic signs with high accuracy, employing a variety of pre-processing and regularization techniques (e.g. dropout), and trying different model architectures. We built highly configurable code and developed a flexible way of evaluating multiple architectures. Our model

reached close to close to 98.5% accuracy on the test set, achieving 99.27% on the validation set.

We used Tensorflow, matplotlib and investigated artificial neural network architectures.

2.1.4 Future Scope

In the future, we believe higher accuracy [19] can be achieved by applying further regularization techniques such as batch normalization and also by adopting more modern architectures such as GoogLeNet's Inception Module, ResNet, or Xception.

CHAPTER 3

Deep Learning Training

3.1 Deep Learning training using Convolution Neural Network (Nvidia model)

Packages used : OS, NumPy, MatPlotLib, Keras, sklearn, imgaug, cv2, pandas, ntpath, random, etc.

Using an open-source self driving car simulator we collected a descent size of dataset consisting of images and label csv file while training the model.

3.1.1 Introduction to Deep Learning for Autonomous Vehicle

Network Architecture

We train the weights of our network to minimize the mean squared error(MSE) between the steering command output by the network and the command of either the simulation driver, or the adjusted steering command for off-center and rotated images. Our network architecture is shown in Figure 3.1. The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers. The input image is split into YUV planes and passed to the network. The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process. Performing normalization in the network allows the normalization scheme to be altered with the network architecture and to be accelerated via GPU processing. The convolutional layers were designed to perform feature extraction and were chosen empirically through a series of experiments that varied layer configurations. We use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel and a non-strided convolution with a 3×3 kernel size in the last two convolutional layers. We follow the five convolutional layers with three fully connected layers leading to an output control value which is the inverse turning radius. The fully connected layers are designed to function as a controller for steering, but we note that by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor and which serve as controller.

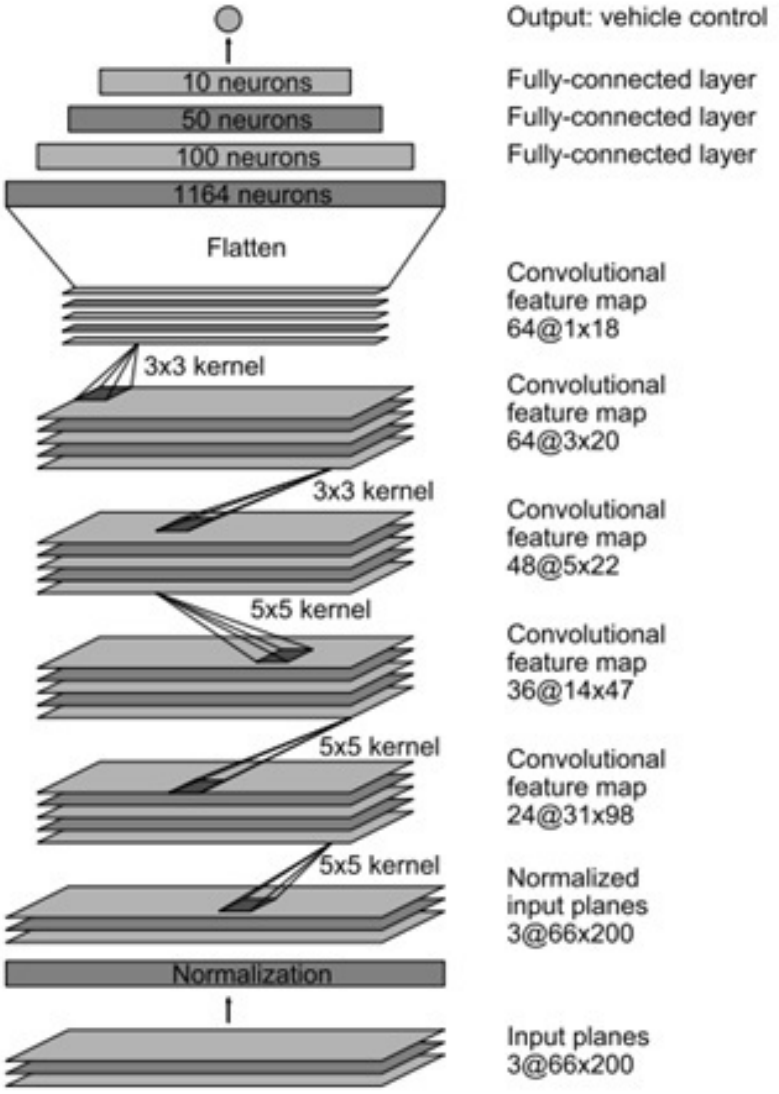

Figure 3.1: Architecture of NVIDIA Model

3.1.2 Training Details

Data Selection

The first step to training a neural network is selecting the frames to use. Our collected data is labeled with road type, weather condition, and the driver's activity (staying in a lane, switching lanes, turning, and so forth). To train a CNN to do lane following we only select data where the driver was staying in a lane and discard the rest. We then sample that video at 10 FPS. A higher sampling rate would result in including images that are highly similar and thus not provide much useful information.To remove a bias towards driving straight the training data includes a higher proportion of frames that represent road curves and we have reduced the proportion of driving straight along a road in the training data.

Algorithm

1. Backpropagation Training Algorithm

3.1.3 Backpropagation

In Machine Learning/Deep Learning, Backpropagation is a widely used algorithm in training Feed Forward Neural Networks for Supervised Learning. Generalizations of backpropagation exist for other artificial neural networks (ANNs) and for functions generally – a class of algorithms referred to generically as "backpropagation". In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes it feasible to use gradient methods for training multilayer networks, updating weights to minimize loss; gradient descent, or variants such as stochastic gradient descent, are commonly used. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this is an example of dynamic programming.

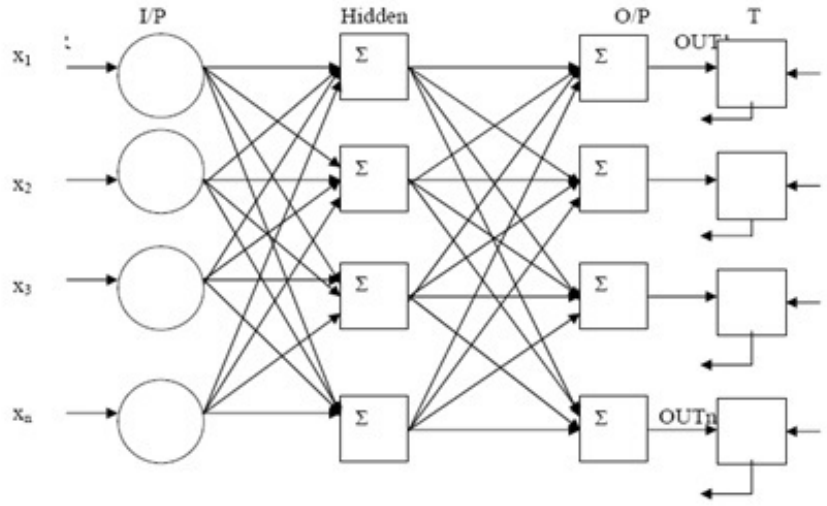

Figure 3.2: Backpropagation Training.

The term backpropagation strictly refers only to the algorithm for computing the gradient, not how the gradient is used; but the term is often used loosely to refer the entire learning algorithm, including how the gradient is used, such as by stochastic gradient descent. Backpropagation generalizes the gradient computation in the delta rule, which is the single-layer version of backpropagation, and is in turn generalized by automatic differentiation, where backpropagation is a special case of

reverse accumulation (or ”reverse mode”). The term backpropagation and its general use in neural networks was announced in Rumelhart, Hinton & Williams (1986a), then elaborated and popularized in Rumelhart, Hinton & Williams (1986b), but the technique was independently rediscovered many times, and had many predecessors dating to the 1960s.

3.1.4 Objective

The general idea here is to gather training data by driving the car in simulator, then train the Deep Neural Network with that data, and in the end let the car be driven by the model generated by the deep neural network.
We have used an open-source self driving car simulator namely “Udacity’s Self Driving Car Simulator” to train our car by gathering image data during training since it is a deep learning project and use the training data to teach our car to drive autonomously on other background environmental conditions/ roads.

Deep Learning is one of ways to make autonomous driving possible. Here we used Nvidia’s “End to End Learning for Self-Driving Cars” network. We used Udacity’s Self-Driving Car Simulator to do this. Udacity built this simulator for their Self-Driving Car Nanodegree and have now open-sourced it. It was built using Unity. Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop both three-dimensional and two-dimensional video games and simulations for computers, consoles and mobile devices.

There are 3 parts to the whole process:

- Data Generation for the Autonomous System
- Training the Autonomous System
- Testing the Autonomous System

Data Generation for the Autonomous System

This part is inspired by Nvidia’s Self-Driving Car’s data collection process. They attached 3 cameras to a car. One is placed in the center and the other 2 are placed on each side of the car. They recorded the steering wheel’s data- capturing the steering angle. So, in our process, we are going to capture:

$\rightarrow$ Images from Center, Left and Right Cameras.

Figure 3.3: Image from left dash-board camera.

Figure 3.4: Image from center dash-board camera.

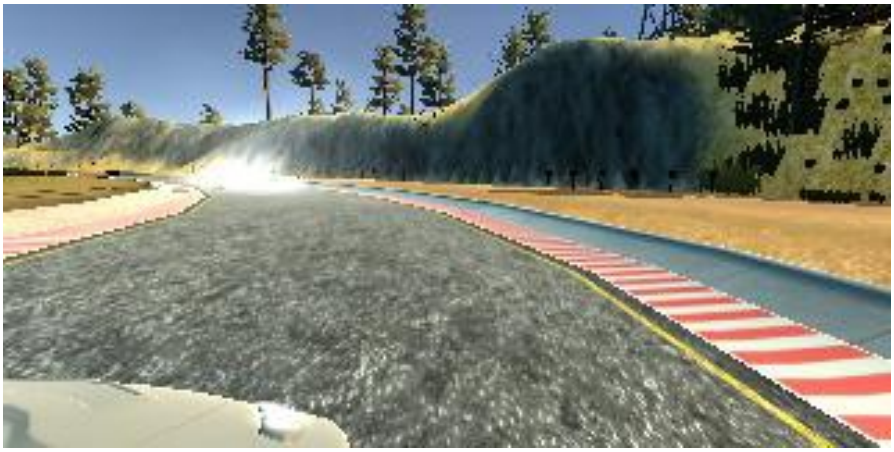

Figure 3.5: Image from right dash-board camera.

.

→ Steering Angle.

→ Speed of the car.

→ Throttle.

→ Brake.

These recorded values are stored in a CSV file.

Center Image	Left Image	Right Image	Steering Angle	Throttle	Break	Speed
IMG/center_2016_10_21_	IMG/left_2016_10_21_17_	IMG/right_2016_10_21_1	0	0.09803098	0	0
IMG/center_2016_10_21_	IMG/left_2016_10_21_17_	IMG/right_2016_10_21_1	0	0.09803098	0	0.06057268

Figure 3.6: Sample part of the driving_log.csv file

Here is a table attached :

$SlNo$	$center$	$left$	$right$	$steer$	$throttle$	$reverse$	$speed$
1	c_20_01.jpg	l_20_01.jpg	r_20_01.jpg	0.0	0.0	0	0.000078
2	c_20_02.jpg	l_20_02.jpg	r_20_02.jpg	0.0	0.0	0	0.000079
3	c_20_03.jpg	l_20_03.jpg	r_20_03.jpg	0.0	0.0	0	0.000078
4	c_20_04.jpg	l_20_04.jpg	r_20_04.jpg	0.0	0.0	0	0.000080
5	c_20_05.jpg	l_20_05.jpg	r_20_05.jpg	0.0	0.0	0	0.000078
6	c_20_06.jpg	l_20_06.jpg	r_20_06.jpg	0.0	0.0	0	0.000078
7	c_20_07.jpg	l_20_07.jpg	r_20_07.jpg	0.0	0.0	0	0.000079
8	c_20_08.jpg	l_20_08.jpg	r_20_08.jpg	0.0	0.0	0	0.000078
9	c_20_09.jpg	l_20_09.jpg	r_20_09.jpg	0.0	0.0	0	0.000080
10	c_20_10.jpg	l_20_10.jpg	r_20_10.jpg	0.0	0.0	0	0.000078
11	c_20_11.jpg	l_20_11.jpg	r_20_11.jpg	0.0	0.0	0	0.000078
12	c_20_12.jpg	l_20_12.jpg	r_20_12.jpg	0.0	0.0	0	0.000079

Table 3.1: Visual Sample of driving_log.csv Table

High-level view of the Nvidia's data collection system The steering wheel is attached to the system via the Controller Area Network(CAN) to feed the value of steering wheel, throttle, break. The cameras are also connect to the system where they are feeding in continuous stream [20] of video data. The system is called Nvidia Drive PX- Nvidia Drive is a AI platform that allows the users to build and deploy self-driving cars, trucks and shuttles. It combines deep learning, sensor fusion, and surround vision to change the driving experience. A Solid State Drive is used to store all the data that is collected. The Udacity's Self Driving Car simulator mimics the same.

We can find the simulator here.

The simulator has 2 modes -

→ Training mode,

→ and Autonomous mode.

Training mode

In the training mode, the simulator captures the images from 3 cameras, speed value, throttle value, brake, and steering angle. We have to click on the record icon on the top right of the screen to start the recording.

Figure 3.7: Training Mode on Simulator

Autonomous mode

In this mode the simulator acts as a server and the python script acts as client.

Training the Autonomous System The data collected during the Data Generation part involved collecting all the camera images, steering angle and others while a human driver was driving the car. We trained a model that clones how the human was driving the car- essentially clones the driver's behaviour to different road scenarios. This is called Behavioural Cloning. To formally define it- Behavioural cloning is a method by which human sub-cognitive skills can be captured and reproduced in a computer program.

This is further discussed in detail in Chapter 4.
The autonomous mode looks like this:

Figure 3.8: Autonomous Mode on Simulator

To formally define its Behavioural cloning, it is a method by which human sub-cognitive skills can be captured and reproduced in a computer program.

Training the Neural Network

The images captured from the 3 cameras are randomly shifted and rotated and then fed into the Neural Network. Based on these inputs, the Neural Network would output a single value- the steering angle. Essentially, based on the input images, the Neural Neural decides by what angle the car must be steered. This output value is compared with the steering data collected from human driving to compute the error in the Neural Network's decision. With this error, the model uses Backpropogation algorithm to optimize the parameter (weights) of the model to reduce this error.

We will use Nvidia's Convolutional Neural Network(CNN) architecture.(The network consists of 9 layers- a normalization layer, 5 convolutional layers and 3 fully connected layers. The input image is converted to YUV. The first layer normalises the image. The normalisation values are hardcoded and it is not trainable. The convolutional layers perform feature extraction. The 1st 3 layers use strided convolutions with a 5×5 kernel, and the last 2 convolutional layer use non-strided convolution with a 3×3 kernel size. The convolutional layers are followed by 3 fully connected layers. The fully connected layers are supposed to act as a controller for the Autonomous system. But, given the end-to-end learning of the whole network,

it is hard to say if the fully connected layers are the ones solely responsible for the controller.)

Testing the Autonomous System

The testing happens using only the center camera images. The center camera input is fed to the Neural Network, the Neural Network ouputs the steering angle value, this value is fed to the Autonomous car. The Udacity's Self Driving Car simulator follows the Server-Client architecture as follows:

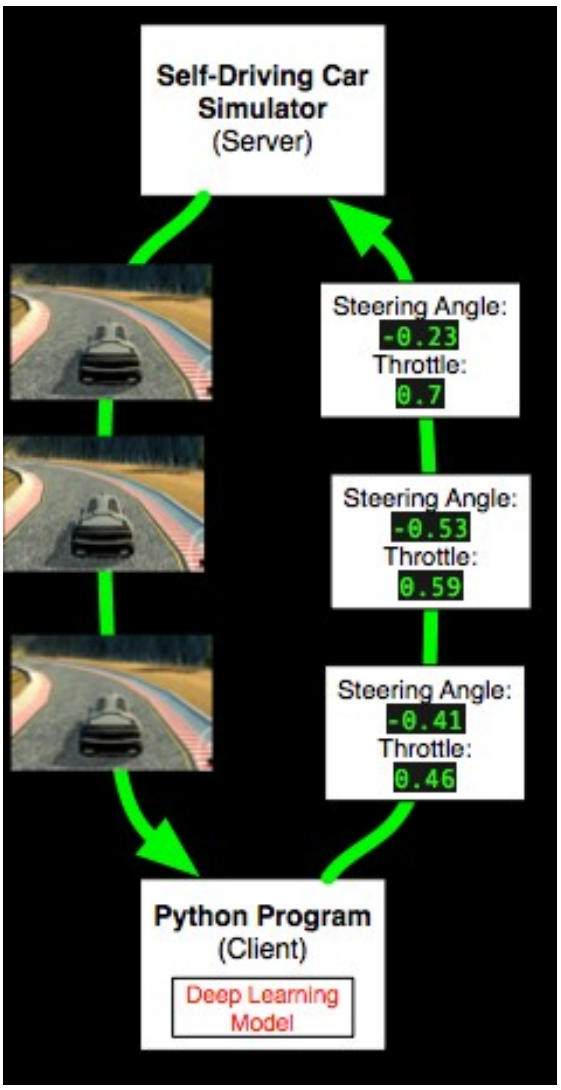

Figure 3.9: Simulator(Server) - Neural Network(Client)

The server will be the simulator and the client would be the Neural Network, or rather the Python program. This whole process is a cyclic feedback loop. The simulator outputs the images, the python program analyses it and outputs the steering angle and the throttle. The simulator receives this and turns the car accordingly. And the whole process goes on cyclically.This is further discussed in detail in chapter 5.

The project goals and pipeline

The main task of the project is to build a Convolutional Neural Network using Keras library in Python. A very important step is to properly collect data in simulator by recording correct driving and recovery driving. After preprocessing the data and training the model, we should check if we are able to use it to drive a car without leaving the road.

Steps of this project:
1. Collect video data of driving behavior using simulator
2. Preprocess the data
3. Choose a CNN model and create it in Keras
4. Train and validate the model
5. Verify if the model predicts correct angles during autonomous driving.

Data collection

Below, there is a short excerpt from my data collection process. According to recommendations from the Udacity course and other participants, it consists of two modes of driving:

- Good driving - teaching the model how to drive properly around the track
- Recovery driving - teaching the model how to recover from situations when the car is on a side of the road

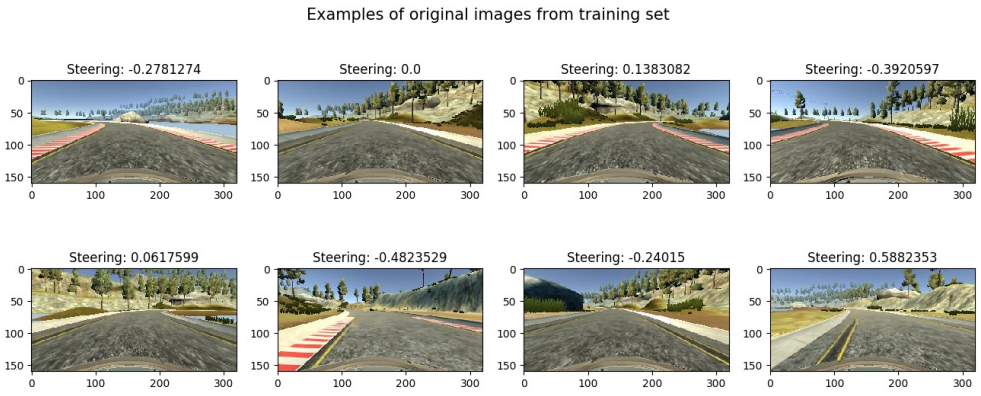

Figure 3.10: Sample of original images from data set with steering angles

The recorded data, which will be used to feed the neural network, is composed of: set of video frames and corresponding steering angles

After training, the model will be able to predict the angle having the image on it's input.

It turned out that the data obtained from the recovery mode is even more important for the Behavioral Cloning project. From the ”good driving” dataset most of the samples contain steering angle equal to 0. Having collected the ”recovery driving” we gain valuable information about non-zero steering angles. Thanks to such data, the model would learn how to quickly apply bigger steering angle when vehicle is suddenly off the track. Even if the model is not perfectly tuned and the car will wobble right and left in autonomous mode, there is a higher chance that we can successfully pass challenging turns on track.
Below, there are random frames from training set with corresponding steering angles. Applied steering is normalized to (-1, 1) range.

Additionally, in the Udacity simulator, it was possible to use images from three cameras. They are ”mounted” in a row, with some distance from each other. So, images from them differed slightly. Below, you can see frames from left and right camera taken at the same moment.
The idea behind this is to collect more samples and diversify them.
During autonomous mode, images fetched into model input will come from the central camera only. That means that we should add a certain angle shift to frames from left and right camera.

Figure 3.11: Images from left and right dash-board cameras.

In the shown example, we can see that a vehicle is turning left. For the angle corresponding to image from the right camera, we should subtract some value. It’s because when we will see such image in auto mode (using central camera), we should expect an answer from the model: Turn a bit more to the left. It’s working the opposite in the left camera images.

From each sample from data collection, we picked randomly one of three images from available cameras - applying a relevant angle shift if needed. In total, 3 laps were used for further processing.

Training data analysis

Udacity has already offered some pre-recorded laps, but we have decided to play around with simulator ourselves. So we have recorded 3 laps for each direction of the track. Both directions are needed in order to avoid the bias of the deep neural net towards turning to the left side of the road.
The recording resulted in 14919 captured images. Images contain captured data from three cameras on the car: left, center and right.
Captured images from left, center and right camera from the car.
For the training purposes we have used only center camera which proved to be enough for getting very decent end-results. In order to make the model more general, it is advised to use all three cameras for car to be able to handle scenarios for getting back to the center of the track better.

The below image shows the steering angle distribution :

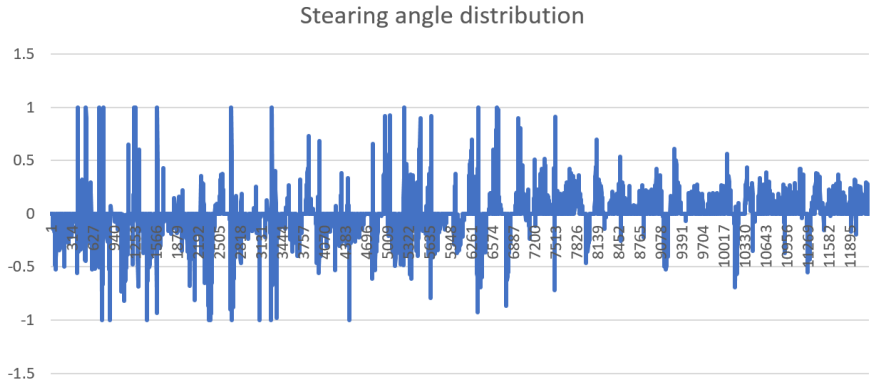

Figure 3.12: Steering Angle Distribution.

When using multiple cameras, it is important to remember that steering angle needs to be adjusted properly with a constant for both left and right camera.

Multiple camera capturing

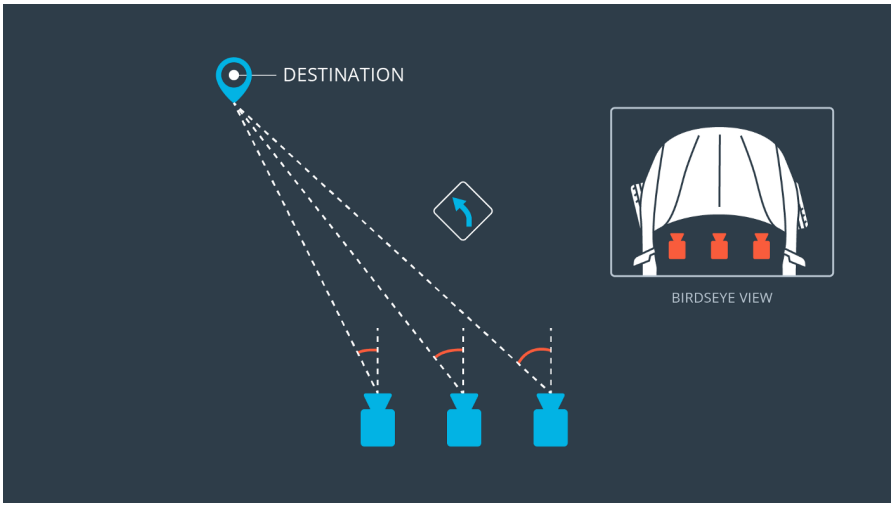

Figure 3.13: Multiple camera setup on dashboard to collect data

The training data also contains CSV file which contains time stamped image captures together with steering angle, throttle, break and speed.

CSV file contains the metadata from the training data Since the training data contains laps in both track directions, the steering angle is not biased:

80% of the data is intended to be used for training and 20% for validation. All the training data can be found here.

Data pre processing

As in any Machine Learning project it's crucial to preprocess our data, e.g. by data augmentation, cleaning, transformation or dimensionality reduction. It can significantly help to train the model. Let's visualize some properties of our dataset, clean it a bit and then produce more samples of desired properties. Generating more samples from existing ones can be helpful if we don't have large enough dataset. But it also can prevent overfitting because during generation of new samples we can add image shifts, translations, shadows etc. The data is more diverse and could be potentially used for more "unseen" situations.

Data cleaning

After drawing a histogram of steering angles for all collected samples, one conclusion comes straightaway. Although we did a "recovery driving" there is still a huge number of samples with angles close to 0. This could bias the model towards predicting 0 angle. To improve this situation we rejected about 85% of samples whose steering angle is really close to 0. Below, there are histograms of training samples before and after this operation:

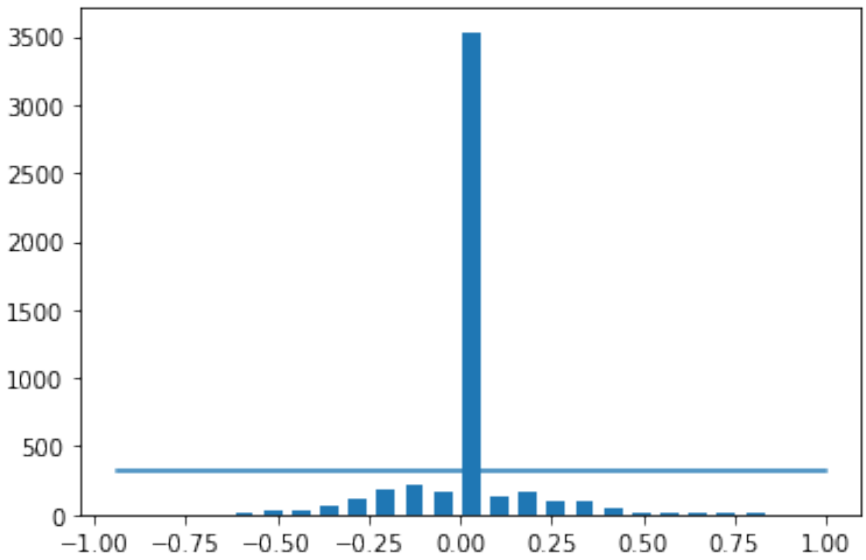

Figure 3.14: Histogram of original data-set

We can see that there are still many 0 angles among the samples but it's significantly lower value than before.

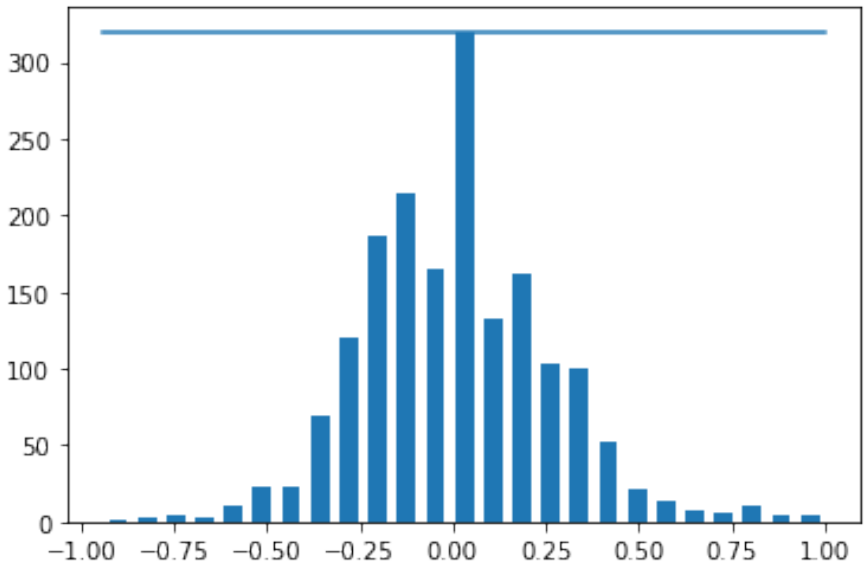

Figure 3.15: Histogram of data-set after reducing 0 angle steering images

Data augmentation

On the latter histogram we can now observe that for larger steering angles (let's say, angle $>$ 0.25), most of them are on the left side of the chart. It means that, during samples collection, vehicle turned left more than turned right. It's justified as the circuit in the simulator is anti-clockwise. Again, we don't want to bias the model towards negative angles in this case. The remedy is to flip some of the samples.

It means flipping an image horizontally and changing the sign of corresponding steering angle. We did it for randomly chosen 50% of samples.

Correction Factor:

Infact we need to introduce a correction factor that we will add in the images taken from left dashboard camera and a correction factor that we will subtract in the images taken from right dashboard camera in order to keep our vehicle in the center. So, lets suppose our steering correction angle is 0.2. So in case of left images
steering angle = steering angle + 0.2
and in case of right images
steering angle = steering angle - 0.2

Flipping:

That was the first step of data pre processing. Moving on, what else can be done to generate more data. We can flip images horizontally and negate the corresponding steering angle to get an altogether different record.So that means by 1 record in csv we are able to generate 6 different records. One for center, one for left , one for right, one for center flipped, one for left flipped, one for right flipped.
In the simulator there are a lot of left turns, so we don't want that our model to know only how to turn to left so we are flipping to generalize so that it can make a right turn too.

The final histogram depicts samples and their angles after the flipping operation:

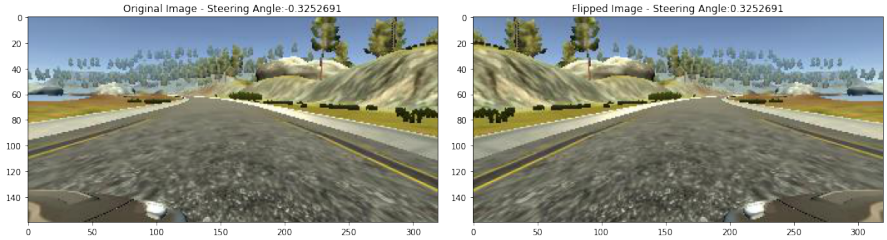

Figure 3.16: Flipping original image

Normalizing:

we need to Normalize the model. Well it is a perfect step to normalize the data before processing so we normalize all the images mean centered. Since we have a lot of images so it is not a good idea to apply it to all images in one go, since you may go out of memory we will later add this step in keras model and will give keras the opportunity to do it for us.This process is also known as panned process.

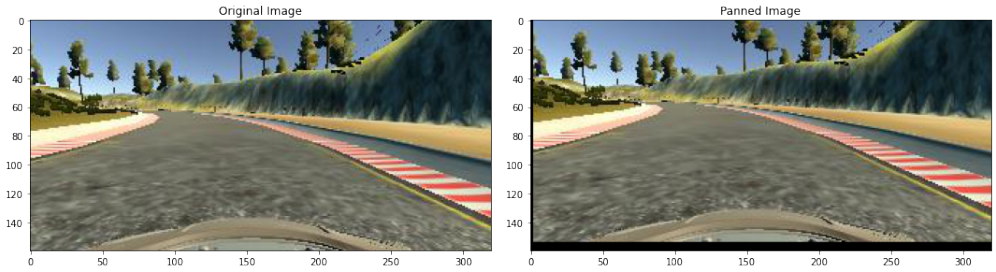

Figure 3.17: Panned original image

Cropping:

If we observe the image, the trees and sky do not play any important role in our image, instead they are more likely to distract our model. So we don't need them. Also if we observe, car dash is visible in the bottom of the image so we crop that too. We avoid cropping anything horizontally because we do not want to loose information of the curvature of the road. For cropping images initially, we use keras to crop.

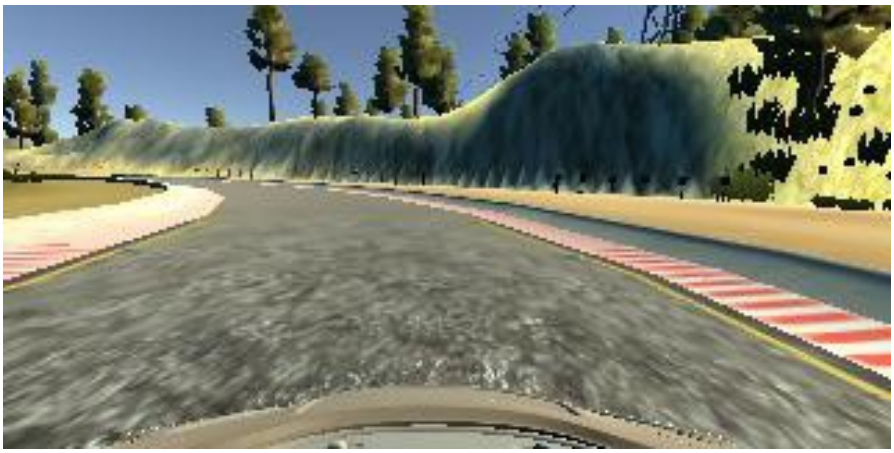

Figure 3.18: Original image before cropping.

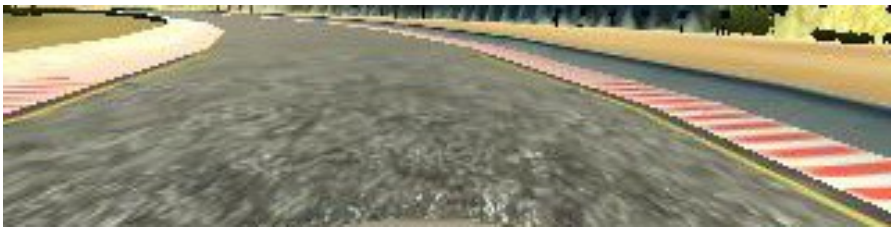

Figure 3.19: Original image after cropping.

Figure 3.20: Blurred image after cropping.

Zooming:

We can zoom the data set so that the order in which these images are processed by the CNN does not matters.

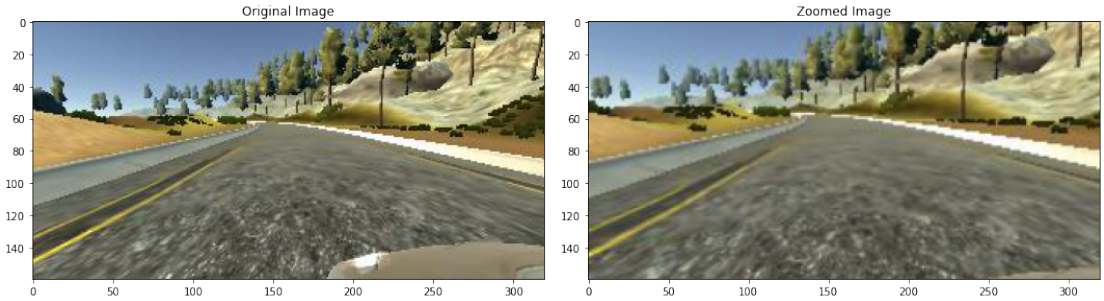

Figure 3.21: Zoomed original image

There are many more Augmentation techniques that can be applied like varying lighting conditions and brightness, a little bit of transformations as well but let's be happy with the current data set and feed this into our model and see what happens. Some of them are shown below after applying multiple augmentation techniques on them :

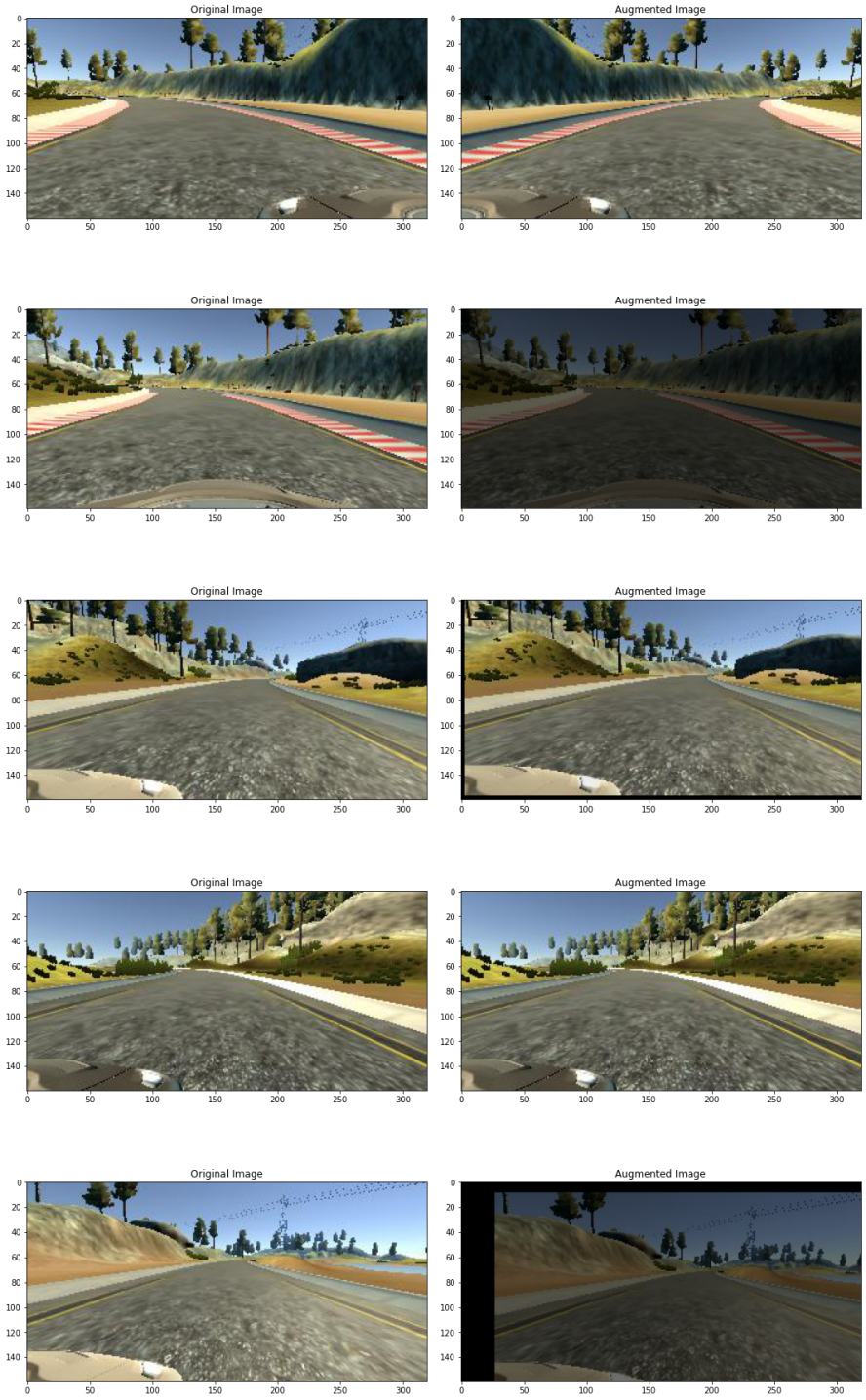

Figure 3.22: Augmented original image 1

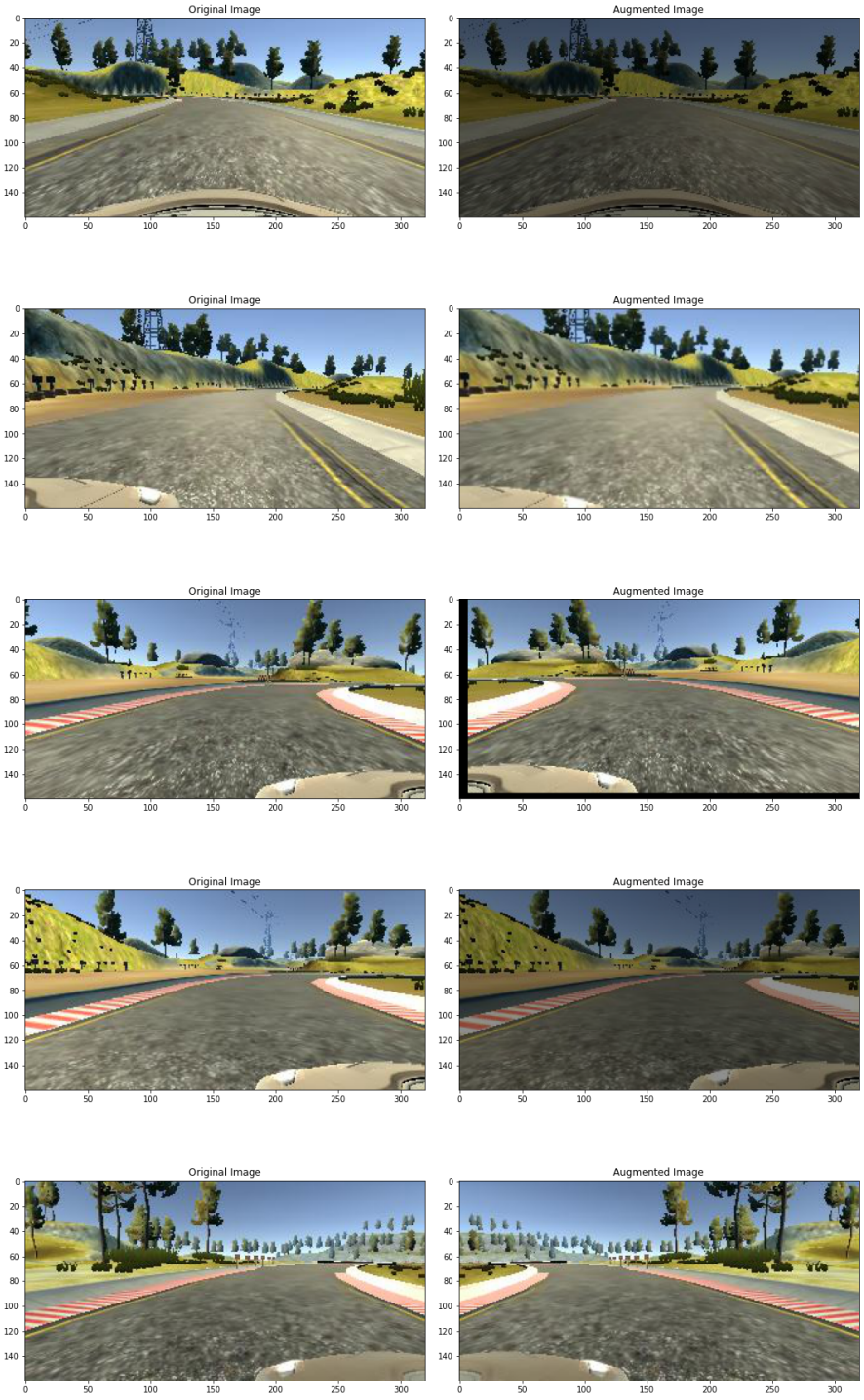

Figure 3.23: Augmented original image 2

We finally rescaled all images from dataset from 320x160 pixels to 200x66. Then, using Keras Cropping2D layer I cropped 80 rows from the top and 40 rows from the bottom of each image. This removed unnecessary information about sky, trees and the car hood at the bottom. The final image size was 200x66 which was intentional as it's regarded easier for CNN to operate when the input image is as per the requirement of our model.

The NVIDIA Model works great and extract details efficiently with YUV coloured image rather than RGB image, so we change the color of our data-set images into YUV-format. It looks like the below figure :

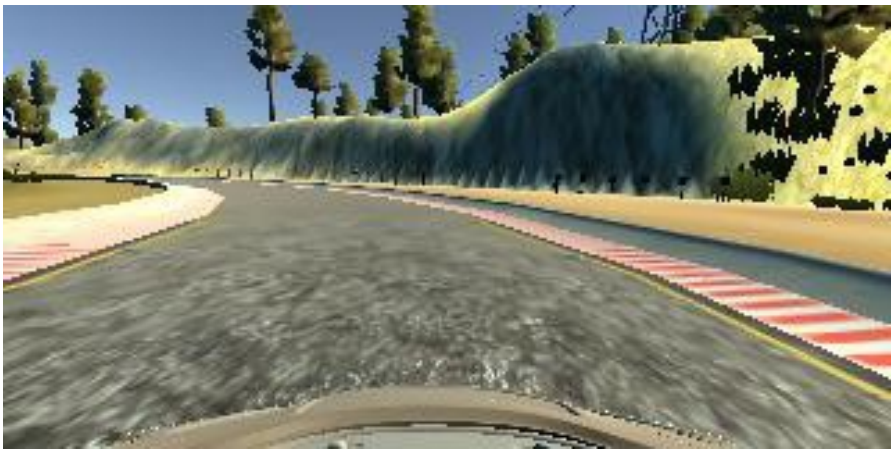

Figure 3.24: Original Image in RGB color.

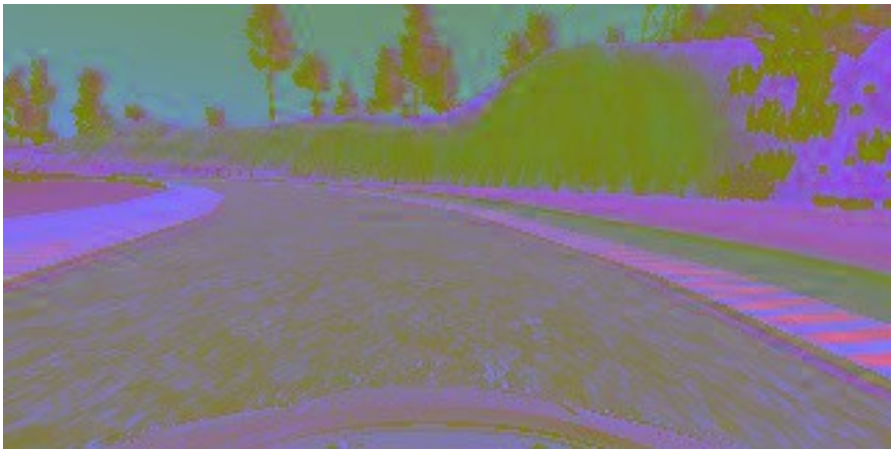

Figure 3.25: Augmented Image in YUV format.

The final pre processed data looks like this :

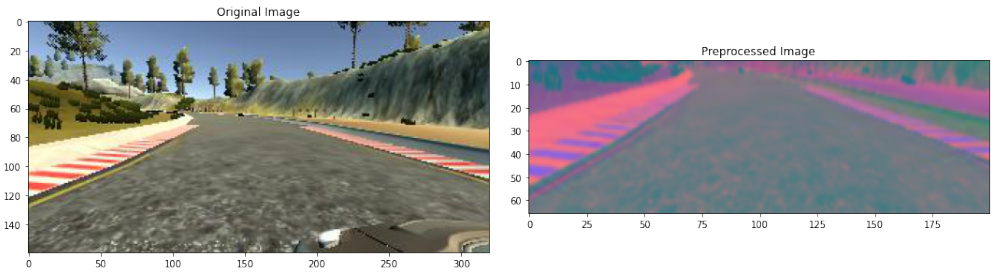

Figure 3.26: Pre processed original image

Then the training and validation images can be described as the below figure :

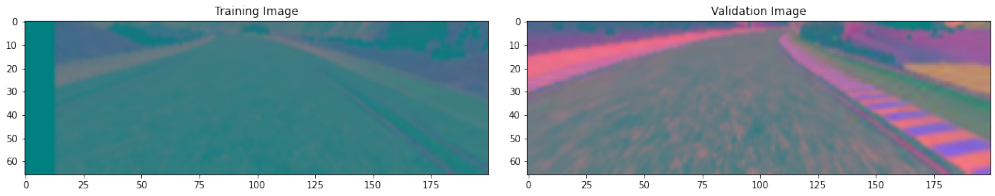

Figure 3.27: Training image and Validation image sample

3.1.5 Model Architecture

Throughout the Behavioral Cloning project we tested 3 models: LeNet, model proposed by Comma.ai and model reported in Nvidia paper for end-to-end learning for self-driving cars. Our first step was to use a convolution neural network model similar to the LeNet model. WE thought this model might be appropriate because it worked well in previous projects utilizing CNN for e.g. handwritten digits recognition. After testing these models, adding/removing some layers we observed that the Nvidia model works the best in general. It's a well known CNN architecture recently, it has about 27 million connections and 250 thousand parameters approx. Ref. Nvidia fig. 3.1

Generators

Important Note- If we have 10,000 samples in data.csv then we will have 10,000 * 6 images in our dataset and it is not feasible to load these many images in one shot, since it will consume all the memory. So we have to take the help of generators that generate data in batches and use "yield" keyword, which keeps on appending images one after the other. Also, it is recommended to perform data augmentation inside generators.

To test our model how it is performing, lets divide our dataset into two parts, training set and validation set. This is also helpful to judge the performance as well to combact overfitting.

3.1.6 Model Tuning

In order to gauge how well the model was working, We split our image and steering angle data into a training and validation set. We found that sometimes our model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting. To combat this, we modified the model so that after each convolutional layer there is a dropout layer.
Also, to increase nonlinearity, "ELU" activation layers were placed after each layer

of the model. The model used an Adam optimizer, so the learning rate was automated during the training process. We tuned a bit the batch size which finally equaled 100. Also, we were experimenting with the number of epochs between 5 and 15. Finally, it was sufficient to run the training on 10 epochs.

3.1.7 Final Model Architecture

The final model architecture consisted of a convolution neural network with the following layers and layer sizes. Total number of parameters (all trainable) equaled 252219.

3.1.8 Training and the final outcome

Training was done on Google Colab instance which was equipped with standard Tesla GPUs. After the training was done, the car was able to complete the track by itself.

The graph of the Loss plot is shown below :

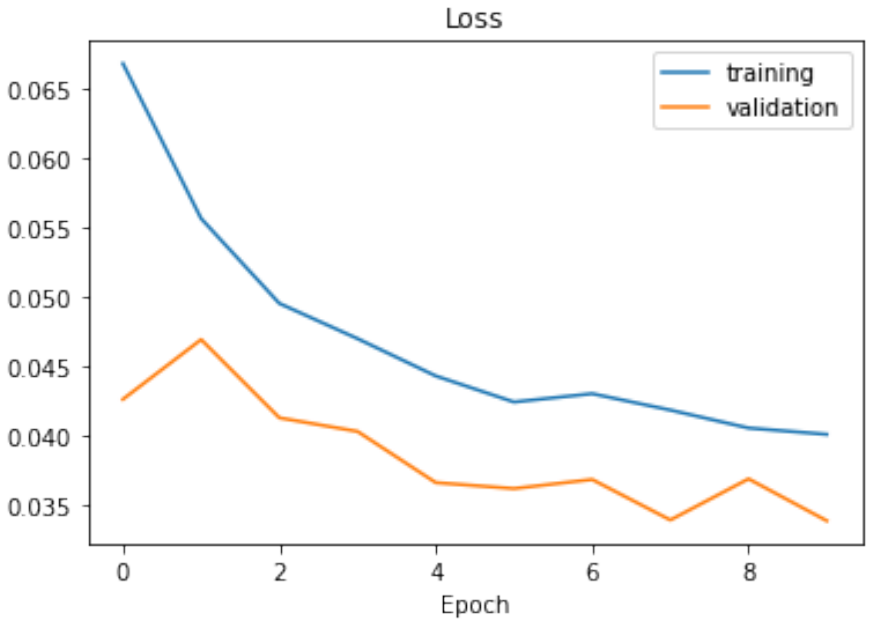

Figure 3.28: Loss plot of our model

But the most important thing learnt is the Importance of GOOD training data [21]. Things at a point do depend on the model architecture but the training data is the ultimate decider.

Simulation

While testing a trained CNN, we first evaluate the networks performance in simulation. A simplified block diagram of the simulation system is shown in Figure 1.2. The simulator takes pre-recorded frames of videos from a forward-facing on-board camera on a 3D Game-driven data-collection vehicle and generates images that approximate what would appear if the CNN were, instead, steering the vehicle. These test videos are time-synchronized with recorded steering commands generated by the player(training) driver.

Since the game drivers might not be driving in a particular lane all the time, we manually calibrate the lane associated with each frame in the video used by the simulator. We call this position the "ground truth". The simulator transforms the original images to account for departures from the ground truth.

This transformation also includes any discrepancy between the game driven path and the ground truth. The simulator accesses the recorded test video along with the synchronized steering commands that occurred when the video was captured. The simulator sends the first frame of the chosen test video, adjusted for any departures from the ground truth, to the input of the trained CNN. The CNN then returns a steering command for that frame. The CNN steering commands as well as the recorded training-driver commands are fed into the dynamic model of the vehicle to update the position and orientation of the simulated vehicle. The simulator then modifies the next frame in the test video so that the image appears as if the vehicle were at the position that resulted by following steering commands from the CNN. This new image is then fed to the CNN and the process repeats. The simulator records the off-center distance (distance from the car to the lane center), the yaw, and the distance traveled by the virtual car. When the off-center distance exceeds one meter, a virtual training intervention is triggered, and the virtual vehicle position and orientation is reset to match the ground truth of the corresponding frame of the original test video.

CHAPTER 4

Vehicle behavioral cloning

4.1 Introduction

After playing with lane lines detection and traffic signs classifcation, it's time to feed a Deep Neural Network with a video recorded during human driving. The model will be trained afterwards to act "humanlike" and predict correct steering angles while driving autonomously. Such procedure is also called Behavioral Cloning [22] [23]. For simplicity (and public safety) We collected video data and tested the resulting DNN in a game-like simulator developed by Udacity.

4.1.1 Behavioral Cloning

This method is widely used among the companies in the industry. Google's Waymo reports to collect 4 million miles of data by Nov 2017. It's a huge, still growing database which enables to produce high-quality models like Deep Neural Networks. Their goal is to generalize and behave appropriately in any situation on a road.

Even in previously unseen situations. Some companies, like drive.ai, almost entirely rely on such holistic deep-learning approach. This is in contrast to a traditional robotics approach or when smaller DNNs are used for different system components. Of course, processing of such enormous amount of data is a great challenge.

This stimulates the need of faster GPU processors or new DNN structures. Here , you can find an informative analysis describing what effort it takes to train such data in real production environment. It's worth noticing that recorded data usually doesn't come only from cameras. There are also LiDARs or radars which fused together create a complete machine vision system.

The main objective of the behavioural cloning is to understand the behaviour and state of mind of the driver while driving inorder to adapt the safety [24] and stability of the vehicle and passengers on-board. In other terms it is the process of cloning/ replicating the behaviour of the driver while collecting the data for training the model.

So the better the performance and stability of the driver while training the better will be the accuracy of the autonomous vehicle and the more the training data more will be the precision to deal with certain new conditions and environmental factors.

CHAPTER 5

Simulation

5.1 Simulation on Open source self driving car simulator

While testing a trained CNN, we first evaluate the networks performance in simulation. A simplified block diagram of the simulation system is shown in Figure 1.2. The simulator [25] takes pre-recorded frames of videos from a forward-facing on-board camera on a 3D Game-driven data-collection vehicle and generates images that approximate what would appear if the CNN were, instead, steering the vehicle. These test videos are time-synchronized with recorded steering commands generated by the player(training) driver.

Since the game drivers might not be driving in a particular lane all the time, we manually calibrate the lane associated with each frame in the video used by the simulator. We call this position the "ground truth". The simulator transforms the original images to account for departures from the ground truth.

Note that this transformation also includes any discrepancy between the game driven path and the ground truth. The simulator accesses the recorded test video along with the synchronized steering commands that occurred when the video was captured. The simulator sends the first frame of the chosen test video, adjusted for any departures from the ground truth, to the input of the trained CNN. The CNN then returns a steering command for that frame. The CNN steering commands as well as the recorded training-driver commands are fed into the dynamic model [8] of the vehicle to update the position and orientation of the simulated vehicle. The simulator then modifies the next frame in the test video so that the image appears as if the vehicle were at the position that resulted by following steering commands from the CNN. This new image is then fed to the CNN and the process repeats. The simulator records the off-center distance (distance from the car to the lane center), the yaw, and the distance traveled by the virtual car. When the off-center distance exceeds one meter, a virtual training intervention is triggered, and the virtual vehicle position and orientation is reset to match the ground truth of the corresponding frame of the original test video.

5.1.1 Simulation process

Once training is complete, we save the model to use it for driving the car autonomously in our simulator. We make a state dictionary and save the model with .h5 format using model.save('model.h5').
Before we begin testing our model, we need a file that will load our model, get the frames of the track from the simulator to process through our model and send the steering prediction back to the simulator. We have made a drive.py file which basically a PyTorch version of Udacity's drive.py which is a client-server program to establish a connection between our trained state dictionary model.h5 file and the simulation application to perform the autonomous driving of car in a completely new track.

The final step was to run the Udacity simulator in autonomous mode which used our CNN model. There were a few spots where the vehicle frequently fell off the track, especially during the turn after crossing the bridge. It sometimes ended in spectacular crashes or even in vehicle sinks. To improve the driving behavior in these cases, We tried to tune the model for better generalization, as discussed above. Sometimes we also added new samples covering just turns to emphasize these difficult situations in the dataset. But when it started working and the vehicle was able to drive autonomously around the track without leaving the road, our satisfaction was really big!

The server will be the simulator and the client would be the Neural Network, or rather the Python program. This whole process is a cyclic feedback loop. The simulator outputs the images, the python program analyses it and outputs the steering angle and the throttle. The simulator receives this and turns the car accordingly. And the whole process goes on cyclically.

Open the simulator again and now choose the autonomous mode. The car should drive on its own.

The CNN had never seen this track. The performance on the training track was a little off but fine as it shows that the car was not merely memorizing the track. It recovered successfully from a few critical situations, even though none of those maneuvers had been performed during training.

On the upper left corner a predicted steering angle is displayed.

5.1.2 Conclusions

Of course, a lot of upgrades can be done to this model. The main task in Behavioral Cloning project was to drive autonomously on the same track where the data was collected. But we had also an access to another, more challenging track - mountainous and with a lot of shadows on roads. . It would be closer to the real-world application requirements where vehicles need to handle new and very different situations during driving. We could further augment our data by introducing artificial shadows or by shifting the image in some directions. Also, better CNN architecture could be probably chosen, with more optimal parameters for this task.

5.1.3 Possible improvements

In 10 years most of us probably won't own a car. We'll have some subscription with some company like an Uber... and we would pay Rs. 10,000 approx. a month and every morning we wake up with a car in our driveway that will take us to work, if the law of the land [26] permits.

CHAPTER 6

Screenshots

6.1 Working and Results of self driving car simulator

.

We have used Mac OS Catalina Version 10.15.3 with the following configurations :

→ 16 GB DDR3 RAM

→ Graphic Card : NVIDIA GeForce GT 650M + integrated graphics of the system

→ Intel i7 Quad-core CPU

Application Softwares :

→ Anaconda Navigator 1.9.12 with Python 3.7.7

→ Atom 1.45.0 Editor

→ PyTorch Framework

→ Google Colab to run our project as our hardware was not sufficient(Less GPU performance).

Let us Create a folder and name it 'Behavioural Cloning' on the Desktop and Save our 'drive.py' and 'model.h5' files here

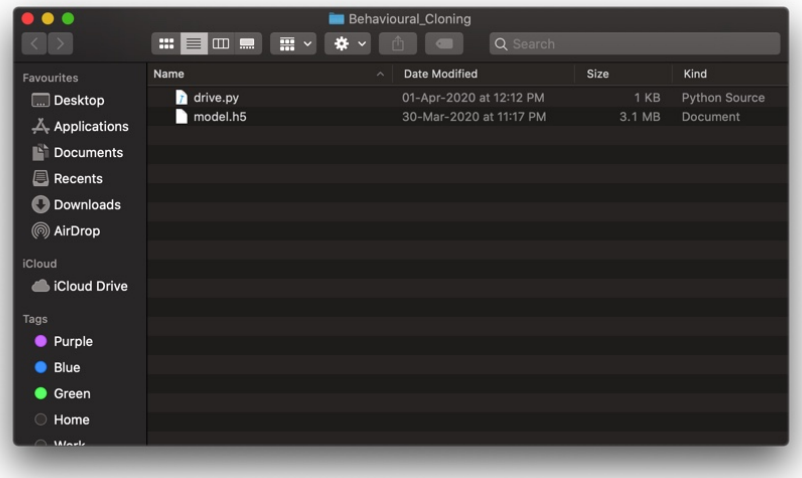

Figure 6.1: Files in Behavioural Cloning

Now we open the terminal and go to our Behavioural Cloning Path and run

our environment 'myenv' in which we have installed all the required modules from Python library(i.e., Keras, Numpy, os, sklearn, cv2, pandas, ntpath, random, matplotlib, imaug, etc).

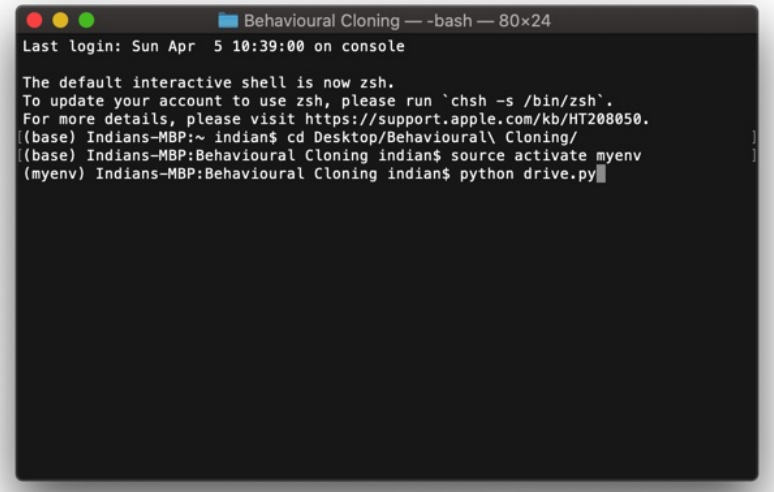

Figure 6.2: Running the program in our system and connecting it to the simulation application.

Our program runs successfully and loads the keras library module and gets connected to the Simulation Application.

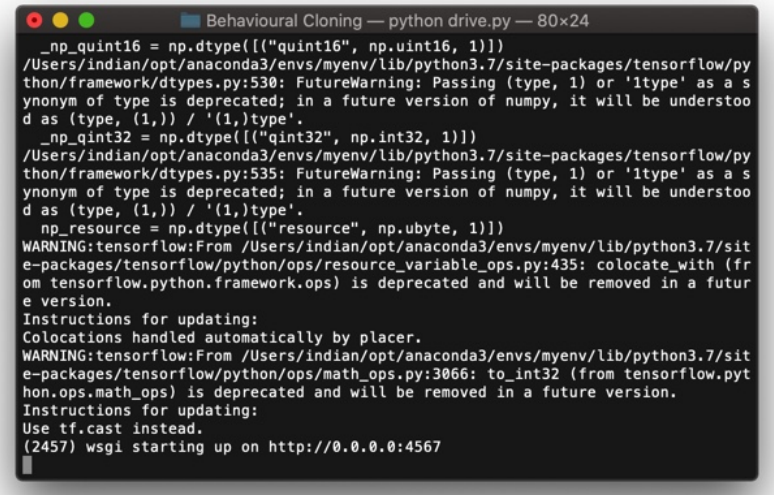

Figure 6.3: Successful running of the program

This is how the Opening window of the Udacity open-source Self-Driving Car

Simulator looks like :

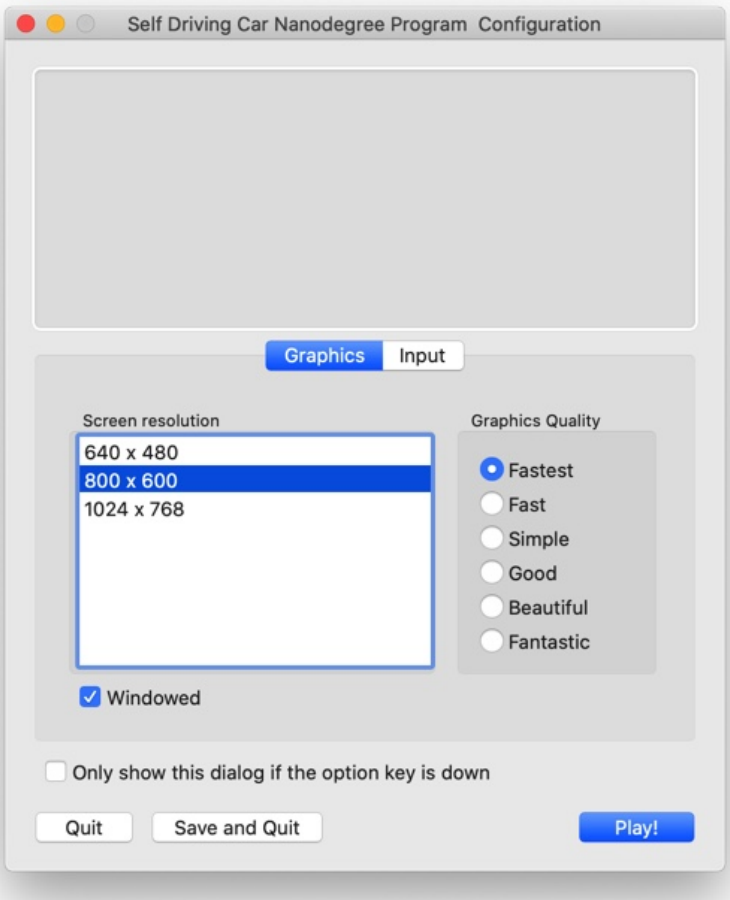

Figure 6.4: Opening Window of Simulator

.

We then press on the 'Play!' button.

This is how the next window looks like :

Figure 6.5: Next Window

We then go for Training and Collecting the image data by Clicking on the 'TRAINING MODE' first.

Figure 6.6: Training Mode

We have trained our model by completing 3 laps in each direction, making a total of 6 laps, driving at the center of the lane since it is a one-way single lane. The below two images are of the car driving autonomously in the same environment/

track where it was trained.

Figure 6.7: Autonomous mode 1

Figure 6.8: Autonomous mode 2

The below two images are of the car driving autonomously in a different environment/ track where it was never trained(in a new hilly road).

Figure 6.9: Autonomous Mode 3

Figure 6.10: Autonomous Mode 4

The below image is of the terminal running in the background indicating 3 columns consisting of Left Turning Angle, Right Turning Angle and the speed of the car in miles per hour for every frame by several calculations inorder to maintain the stability and safety of the car.

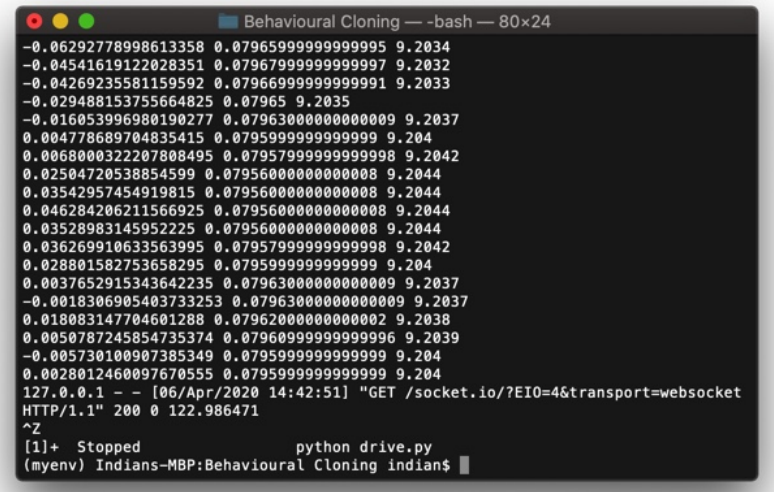

Figure 6.11: Terminal Catalog.

APPENDIX A

CODE ATTACHMENTS

A.1 Lane Detection

```
# Python program to find the edges of the lanes for settimg the
    boundary limit

import cv2
import numpy as np

def make_coordinates(image, line_parameters):
        slope, intercept = line_parameters
        y1 = image.shape[0]
        y2 = int(y1*(3/5))
        x1 = int((y1 - intercept)/slope)
        x2 = int((y2 - intercept)/slope)
        return np.array([x1, y1, x2, y2])

def average_slope_intercept(image, lines):
        left_fit = []
        right_fit = []
        for line in lines:
                x1, y1, x2, y2 = line.reshape(4)
                parametres = np.polyfit((x1, x2), (y1, y2), 1)
                slope = parametres[0]
                intercept = parametres[1]
                if slope < 0:
                        left_fit.append((slope, intercept))
                else:
                        right_fit.append((slope, intercept))
        left_fit_average = np.average(left_fit, axis=0)
        right_fit_average = np.average(right_fit, axis=0)
        left_line = make_coordinates(image, left_fit_average)
        right_line = make_coordinates(image, right_fit_average)
        return np.array([left_line, right_line])

def canny(image):
        gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
        blur = cv2.GaussianBlur(gray, (5,5), 0)
        canny = cv2.Canny(blur, 50, 150)
        return canny

def display_lines(image, lines):
        line_image = np.zeros_like(image)
        if lines is not None:
                for x1, y1, x2, y2 in lines:
                        cv2.line(line_image, (x1, y1), (x2, y2),
                            (255, 0, 0), 10)
        return line_image
```

```
44
45 def region_of_interest(image):
46         height = image.shape[0]
47         polygons = np.array([[(200, height), (1100, height), (550,
               250)]])
48         mask = np.zeros_like(image)
49         cv2.fillPoly(mask, polygons, 255)
50         masked\_image = cv2.bitwise_and(image, mask)
51         return masked_image
52
53 #image = cv2.imread('/Users/indian/Desktop/finding-lanes/test_image.
       jpg')
54 #lane_image = np.copy(image)
55 #canny_image = canny(lane_image)
56 #cropped_image = region_of_interest(canny_image)
57 #lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100, np.array
       ([]), minLineLength=40, maxLineGap=5)
58 #averaged_lines = average_slope_intercept(lane_image, lines)
59 #line_image = display_lines(lane_image, averaged_lines)
60 #combo_image = cv2.addWeighted(lane_image, 0.8, line_image, 1, 1)
61 #cv2.imshow("result", combo_image)
62 #cv2.waitKey(0)
63
64 cap = cv2.VideoCapture("test2.mp4")
65 while(cap.isOpened()):
66         _, frame = cap.read()
67         canny_image = canny(frame)
68         cropped_image = region_of_interest(canny_image)
69         lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100, np.
               array([]), minLineLength=40, maxLineGap=5)
70         averaged_lines = average_slope_intercept(frame, lines)
71         line_image = display_lines(frame, averaged_lines)
72         combo_image = cv2.addWeighted(frame, 0.8, line_image, 1, 1)
73         cv2.imshow("result", combo_image)
74         if cv2.waitKey(1) & 0xFF == ord('q'):
75                 break
76 cap.release()
77 cv2.destroyAllWindows()
```

A.2 Traffic-Sign Classification

```
1 #Traffic Sign Classification Python Program
2 !git clone https://bitbucket.org/jadslim/german-traffic-signs
3
4 !ls german-traffic-signs
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import keras
9 from keras.models import Sequential
10 from keras.layers import Dense
11 from keras.optimizers import Adam
12 from keras.utils.np_utils import to\_categorical
13 from keras.layers import Dropout, Flatten
```

```
14 from keras.layers.convolutional import Conv2D, MaxPooling2D
15 import pickle
16 import pandas as pd
17 import random
18
19 from keras.callbacks import LearningRateScheduler, ModelCheckpoint
20
21 %matplotlib inline
22
23 np.random.seed(0)
24
25 # TODO: Implement load the data here.
26 with open('german-traffic-signs/train.p', 'rb') as f:
27         train_data = pickle.load(f)
28 with open('german-traffic-signs/valid.p', 'rb') as f:
29         val_data = pickle.load(f)
30 # TODO: Load test data
31 with open('german-traffic-signs/test.p', 'rb') as f:
32         test_data = pickle.load(f)
33
34 # Split out features and labels
35 X_train, y_train = train_data['features'], train_data['labels']
36 X_val, y_val = val_data['features'], val_data['labels']
37 X_test, y_test = test_data['features'], test_data['labels']
38
39 #already 4 dimensional
40 print(X_train.shape)
41 print(X_test.shape)
42 print(X_val.shape)
43
44 # STOP: Do not change the tests below. Your implementation should
       pass these tests.
45 assert(X_train.shape[0] == y_train.shape[0]), "The number of images
       is not equal to the number of labels"
46 assert(X_val.shape[0] == y_val.shape[0]), "The number of images is
       not equal to the number of labels"
47 assert(X_test.shape[0] == y_test.shape[0]), "The number of images is
       not equal to the number of labels"
48 assert(X_train.shape[1:] == (32,32,3)), "The dimensions of the images
       are not 32 x 32 x 3"
49 assert(X_val.shape[1:] == (32,32,3)), "The dimensions of the images
       are not 32 x 32 x 3"
50 assert(X_test.shape[1:] == (32,32,3)), "The dimensions of the images
       are not 32 x 32 x 3"
51
52 data = pd.read_csv('german-traffic-signs/signnames.csv')
53 num_of_samples=[]
54 cols = 5
55 num_classes = 43
56 fig, axs = plt.subplots(nrows=num_classes, ncols=cols, figsize=(5,50)
       )
57 fig.tight_layout()
58 for i in range(cols):
59         for j, row in data.iterrows():
60                 x_selected = X_train[y_train == j]
61                 axs[j][i].imshow(x_selected[random.randint(0,(len(
```

```
                    x_selected) - 1)), :, :], cmap=plt.get_cmap('gray'
                    ))
                axs[j][i].axis('off')
                if i == 2:
                    axs[j][i].set_title(str(j) + "-" + row["
                        SignName"])
                    num_of_samples.append(len(x_selected))

print(num_of_samples)
plt.figure(figsize=(12, 4))
plt.bar(range(0, num_classes), num_of_samples)
plt.title("Distribution of the train dataset")
plt.xlabel("Class number")
plt.ylabel("Number of images")
plt.show()

import cv2
plt.imshow(X_train[1000])
plt.axis('off')
print(X_train[1000].shape)
print(y_train[1000])

def grayscale(img):
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        return img
img = grayscale(X_train[1000])
plt.imshow(img)
plt.axis('off')
print(img.shape)

def equalize(img):
        img = cv2.equalizeHist(img)
        return img
img = equalize(img)
plt.imshow(img)
plt.axis('off')
print(img.shape)

def preprocessing(img):
        img = grayscale(img)
        img = equalize(img)
        img = img/255
        return img
X_train = np.array(list(map(preprocessing, X_train)))
X_test = np.array(list(map(preprocessing, X_test)))
X_val = np.array(list(map(preprocessing, X_val)))

plt.imshow(X_train[random.randint(0, len(X_train) - 1)])
plt.axis('off')
print(X_train.shape)

X_train = X_train.reshape(34799, 32, 32, 1)
X_test = X_test.reshape(12630, 32, 32, 1)
X_val = X_val.reshape(4410, 32, 32, 1)

from keras.preprocessing.image import ImageDataGenerator
```

```
115
116 datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range
    =0.1, zoom_range=0.2, shear_range=0.1, rotation_range=10.)
117
118 datagen.fit(X_train)
119
120 batches = datagen.flow(X_train, y_train, batch_size = 15)
121 X_batch, y_batch = next(batches)
122
123 fig, axs = plt.subplots(1, 15, figsize=(20, 5))
124 fig.tight_layout()
125
126 for i in range(15):
127         axs[i].imshow(X_batch[i].reshape(32, 32))
128         axs[i].axis('off')
129 #print(X_batch.shape)
130
131 print(X_train.shape)
132 print(X_test.shape)
133 print(X_val.shape)
134
135 y_train = to_categorical(y_train, 43)
136 y_test = to_categorical(y_test, 43)
137 y_val = to_categorical(y_val, 43)
138
139 # create model
140 def modified_model():
141         model = Sequential()
142         model.add(Conv2D(60, (5, 5), input_shape=(32, 32, 1),
            activation='relu'))
143         model.add(Conv2D(60, (5, 5), activation='relu'))
144         model.add(MaxPooling2D(pool_size=(2, 2)))
145         model.add(Conv2D(30, (3, 3), activation='relu'))
146         model.add(Conv2D(30, (3, 3), activation='relu'))
147         model.add(MaxPooling2D(pool_size=(2, 2)))
148         #model.add(Dropout(0.5))
149         model.add(Flatten())
150         model.add(Dense(500, activation='relu'))
151         model.add(Dropout(0.4))
152         model.add(Dense(num_classes, activation='softmax'))
153         #compile model
154         model.compile(Adam(lr = 0.001), loss='
            categorical_crossentropy', metrics=['accuracy'])
155         return model
156
157 model = modified_model()
158 print(model.summary())
159
160 history = model.fit_generator(datagen.flow(X_train, y_train,
    batch_size=50), steps_per_epoch=2000, epochs=10, validation_data=(
    X_val, y_val), shuffle = 1)
161
162 plt.plot(history.history['loss'])
163 plt.plot(history.history['val_loss'])
164 plt.legend(['training','validation'])
165 plt.title('Loss')
```

```
plt.xlabel('epoch')

plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.legend(['training','validation'])
plt.title('Accuracy')
plt.xlabel('epoch')

# TODO: Evaluate model on test data
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

import requests
from PIL import Image
url = 'https://c8.alamy.com/comp/J2MRAJ/german-road-sign-bicycles-
    crossing-J2MRAJ.jpg'
r = requests.get(url, stream=True)
img = Image.open(r.raw)
plt.imshow(img, cmap=plt.get_cmap('gray'))

img = np.asarray(img)
img = cv2.resize(img, (32, 32))
img = preprocessing(img)
plt.imshow(img, cmap = plt.get_cmap('gray'))
print(img.shape)

img = img.reshape(1, 32, 32, 1)

print("predicted sign: "+ str(model.predict\_classes(img)))
```

A.3 Behavioural Cloning

```
#Cloning the behaviour of a car using deep-learning and training the
    vehicle to run autonomously on a track.

%tensorflow_version 1.x%tensorflow_version 1.x

import tensorflow
print(tensorflow.__version__)

!git clone https://github.com/Shubhashish-Jena/imgdata/

!ls imgdata

!pip3 install imgaug

import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import keras
from keras.models import Sequential
from keras.optimizers import Adam
```

```
21 from keras.layers import Convolution2D, MaxPooling2D, Dropout,
       Flatten, Dense
22 from sklearn.utils import shuffle
23 from sklearn.model_selection import train_test_split
24 from imgaug import augmenters as iaa
25 import cv2
26 import pandas as pd
27 import ntpath
28 import random
29
30 datadir = 'imgdata'
31 columns = ['center', 'left', 'right', 'steering', 'throttle', '
       reverse', 'speed']
32 data = pd.read_csv(os.path.join(datadir, 'driving_log.csv'), names =
       columns)
33 pd.set_option('display.max_colwidth', -1)
34 data.head()
35
36 def path_leaf(path):
37         head, tail = ntpath.split(path)
38         return tail
39
40 data['center'] = data['center'].apply(path_leaf)
41 data['left'] = data['left'].apply(path_leaf)
42 data['right'] = data['right'].apply(path_leaf)
43 data.head()
44
45 num_bins = 25
46 samples_per_bin = 320
47 hist, bins = np.histogram(data['steering'], num_bins)
48 center = (bins[:-1]+ bins[1:]) * 0.5
49 plt.bar(center, hist, width=0.05)
50 plt.plot((np.min(data['steering']), np.max(data['steering'])), (
       samples_per_bin, samples_per_bin))
51
52 print('total data:', len(data))
53 remove_list = []
54 for j in range(num_bins):
55         list_ = []
56         for i in range(len(data['steering'])):
57                 if data['steering'][i] >= bins[j] and data['steering'
                       ][i] <= bins[j+1]:
58                         list_.append(i)
59         list_ = shuffle(list_)
60         list_ = list_[samples_per_bin:]
61         remove_list.extend(list_)
62 print('removed:', len(remove_list))
63 data.drop(data.index[remove_list], inplace=True)
64 print('remaining:', len(data))
65
66 hist, _ = np.histogram(data['steering'], (num_bins))
67 plt.bar(center, hist, width=0.05)
68 plt.plot((np.min(data['steering']), np.max(data['steering'])), (
       samples_per_bin, samples_per_bin))
69
70 print(data.iloc[1])
```

```
71 def load_img_steering(datadir, df):
72         image_path = []
73         steering = []
74         for i in range(len(data)):
75                 indexed_data = data.iloc[i]
76                 center, left, right = indexed_data[0], indexed_data
                       [1], indexed_data[2]
77                 image_path.append(os.path.join(datadir, center.strip
                       ()))
78                 steering.append(float(indexed_data[3]))
79 
80                 # left image append
81                 image_path.append(os.path.join(datadir,left.strip()))
82                 steering.append(float(indexed_data[3])+0.15)
83 
84                 # right image append
85                 image_path.append(os.path.join(datadir,right.strip())
                       )
86                 steering.append(float(indexed_data[3])-0.15)
87         image_paths = np.asarray(image_path)
88         steerings = np.asarray(steering)
89         return image_paths, steerings
90 
91 image_paths, steerings = load_img_steering(datadir + '/IMG', data)
92 
93 X_train, X_valid, y_train, y_valid = train_test_split(image_paths,
       steerings, test_size=0.2, random_state=6)
94 print('Training Samples: {}\nValid Samples: {}'.format(len(X_train),
       len(X_valid)))
95 
96 
97 fig, axes = plt.subplots(1, 2, figsize=(12, 4))
98 axes[0].hist(y_train, bins=num_bins, width=0.05, color='blue')
99 axes[0].set_title('Training set')
100 axes[1].hist(y_valid, bins=num_bins, width=0.05, color='red')
101 axes[1].set_title('Validation set')
102 
103 def zoom(image):
104         zoom = iaa.Affine(scale=(1, 1.3))
105         image = zoom.augment_image(image)
106         return image
107 
108 image = image_paths[random.randint(0, 1000)]
109 original_image = mpimg.imread(image)
110 zoomed_image = zoom(original_image)
111 
112 fig, axs = plt.subplots(1, 2, figsize=(15, 10))
113 fig.tight_layout()
114 
115 axs[0].imshow(original_image)
116 axs[0].set_title('Original Image')
117 axs[1].imshow(zoomed_image)
118 axs[1].set_title('Zoomed Image')
119 
120 def pan(image):
121         pan = iaa.Affine(translate_percent= {"x" : (-0.1, 0.1), "y":
```

```
            (-0.1, 0.1)})
122         image = pan.augment_image(image)
123         return image
124
125
126 image = image_paths[random.randint(0, 1000)]
127 original_image = mpimg.imread(image)
128 panned_image = pan(original_image)
129
130 fig, axs = plt.subplots(1, 2, figsize=(15, 10))
131 fig.tight_layout()
132
133 axs[0].imshow(original_image)
134 axs[0].set_title('Original Image')
135
136 axs[1].imshow(panned_image)
137 axs[1].set_title('Panned Image')
138
139
140 def img_random_brightness(image):
141         brightness = iaa.Multiply((0.2, 1.2))
142         image = brightness.augment_image(image)
143         return image
144
145 image = image_paths[random.randint(0, 1000)]
146 original_image = mpimg.imread(image)
147 brightness_altered_image = img_random_brightness(original_image)
148
149 fig, axs = plt.subplots(1, 2, figsize=(15, 10))
150 fig.tight_layout()
151
152 axs[0].imshow(original_image)
153 axs[0].set_title('Original Image')
154
155 axs[1].imshow(brightness_altered_image)
156 axs[1].set_title('Brightness altered image ')
157
158 def img_random_flip(image, steering_angle):
159         image = cv2.flip(image,1)
160         steering_angle = -steering_angle
161         return image, steering_angle
162
163 random_index = random.randint(0, 1000)
164 image = image_paths[random_index]
165 steering_angle = steerings[random_index]
166
167 original_image = mpimg.imread(image)
168 flipped_image, flipped_steering_angle = img_random_flip(
        original_image, steering_angle)
169
170 fig, axs = plt.subplots(1, 2, figsize=(15, 10))
171 fig.tight_layout()
172
173 axs[0].imshow(original_image)
174 axs[0].set_title('Original Image - ' + 'Steering Angle:' + str(
        steering_angle))
```

```
175
176 axs[1].imshow(flipped_image)
177 axs[1].set_title('Flipped Image - ' + 'Steering Angle:' + str(
        flipped_steering_angle))
178
179
180 def random_augment(image, steering_angle):
181         image = mpimg.imread(image)
182         if np.random.rand() < 0.5:
183                 image = pan(image)
184         if np.random.rand() < 0.5:
185                 image = zoom(image)
186         if np.random.rand() < 0.5:
187                 image = img_random_brightness(image)
188         if np.random.rand() < 0.5:
189                 image, steering_angle = img_random_flip(image,
                        steering_angle)
190
191         return image, steering_angle
192
193 ncol = 2
194 nrow = 10
195
196 fig, axs = plt.subplots(nrow, ncol, figsize=(15, 50))
197 fig.tight_layout()
198
199 for i in range(10):
200         randnum = random.randint(0, len(image_paths) - 1)
201         random_image = image_paths[randnum]
202         random_steering = steerings[randnum]
203
204         original_image = mpimg.imread(random_image)
205         augmented_image, steering = random_augment(random_image,
                random_steering)
206
207         axs[i][0].imshow(original_image)
208         axs[i][0].set_title("Original Image")
209
210         axs[i][1].imshow(augmented_image)
211         axs[i][1].set_title("Augmented Image")
212
213 def img_preprocess(img):
214         img = img[60:135,:,:]
215         img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
216         img = cv2.GaussianBlur(img, (3, 3), 0)
217         img = cv2.resize(img, (200, 66))
218         img = img/255
219         return img
220
221 image = image_paths[100]
222 original_image = mpimg.imread(image)
223 preprocessed_image = img_preprocess(original_image)
224
225 fig, axs = plt.subplots(1, 2, figsize=(15, 10))
226 fig.tight_layout()
227 axs[0].imshow(original_image)
```

```
228 axs[0].set_title('Original Image')
229 axs[1].imshow(preprocessed_image)
230 axs[1].set_title('Preprocessed Image')
231
232
233 def batch_generator(image_paths, steering_ang, batch_size, istraining
    ):
234         while True:
235                 batch_img = []
236                 batch_steering = []
237                 for i in range(batch_size):
238                         random_index = random.randint(0, len(
                            image_paths) - 1)
239                         if istraining:
240                                 im, steering = random_augment(
                                    image_paths[random_index],
                                    steering_ang[random_index])
241                         else:
242                                 im = mpimg.imread(image_paths[
                                    random_index])
243                                 steering = steering_ang[random_index]
244
245                         im = img_preprocess(im)
246                         batch_img.append(im)
247                         batch_steering.append(steering)
248                 yield (np.asarray(batch_img), np.asarray(
                    batch_steering))
249
250
251 x_train_gen, y_train_gen = next(batch_generator(X_train, y_train, 1,
    1))
252 x_valid_gen, y_valid_gen = next(batch_generator(X_valid, y_valid, 1,
    0))
253
254 fig, axs = plt.subplots(1, 2, figsize=(15, 10))
255 fig.tight_layout()
256
257 axs[0].imshow(x_train_gen[0])
258 axs[0].set_title('Training Image')
259
260 axs[1].imshow(x_valid_gen[0])
261 axs[1].set_title('Validation Image')
262
263 def nvidia_model():
264         model = Sequential()
265
266         model.add(Convolution2D(24, 5, 5, subsample=(2, 2),
            input_shape=(66, 200, 3), activation='elu'))
267         model.add(Convolution2D(36, 5, 5, subsample=(2, 2),
            activation='elu'))
268         model.add(Convolution2D(48, 5, 5, subsample=(2, 2),
            activation='elu'))
269         model.add(Convolution2D(64, 3, 3, activation='elu'))
270         model.add(Convolution2D(64, 3, 3, activation='elu'))
271
272         #model.add(Dropout(0.5))
```

```
273             model.add(Flatten())
274             model.add(Dense(100, activation = 'elu'))
275 
276             #model.add(Dropout(0.5))
277             model.add(Dense(50, activation = 'elu'))
278 
279             #model.add(Dropout(0.5))
280             model.add(Dense(10, activation = 'elu'))
281 
282             #model.add(Dropout(0.5))
283             model.add(Dense(1))
284 
285             optimizer = Adam(lr=1e-4)
286             model.compile(loss='mse', optimizer=optimizer)
287             return model
288 
289 
290 print(model.summary())
291 model = nvidia_model()
292 
293 history = model.fit_generator(batch_generator(X_train, y_train, 100,
        1), steps_per_epoch=300, epochs=10, validation_data=
        batch_generator(X_valid, y_valid, 100, 0), validation_steps=200,
        verbose=1, shuffle = 1)
294 
295 plt.plot(history.history['loss'])
296 plt.plot(history.history['val_loss'])
297 plt.legend(['training', 'validation'])
298 plt.title('Loss')
299 plt.xlabel('Epoch')
300 
301 
302 model.save('model.h5')
303 
304 from google.colab import files
305 files.download('model.h5')
```

A.4 Simulation Application Connection

```
1  #Establishing a Server-Client Connection between the simulation
       application and our trained Deep-Learning Model(model.h5)
2  import socketio
3  import eventlet
4  import numpy as np
5  from flask import Flask
6  from keras.models import load_model
7  import base64
8  from io import BytesIO
9  from PIL import Image
10 import cv2
11 
12 sio = socketio.Server()
13 
14 app = Flask(__name__) #'__main__'
```

```
15 speed_limit = 10
16
17 def img_preprocess(img):
18         img = img[60:135,:,:]
19         img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
20         img = cv2.GaussianBlur(img,  (3, 3), 0)
21         img = cv2.resize(img, (200, 66))
22         img = img/255
23         return img
24
25
26 @sio.on('telemetry')
27 def telemetry(sid, data):
28         speed = float(data['speed'])
29         image = Image.open(BytesIO(base64.b64decode(data['image'])))
30         image = np.asarray(image)
31         image = img_preprocess(image)
32         image = np.array([image])
33         steering_angle = float(model.predict(image))
34         throttle = 1.0 - speed/speed_limit
35         print('{} {} {}'.format(steering_angle, throttle, speed))
36         send_control(steering_angle, throttle)
37
38 @sio.on('connect')
39 def connect(sid, environ):
40         print('Connected')
41         send_control(0, 0)
42
43 def send_control(steering_angle, throttle):
44         sio.emit('steer', data = {'steering_angle': steering_angle.
               __str__(),'throttle': throttle.__str__()})
45
46
47 if __name__ == '__main__':
48         model = load_model('model.h5')
49         app = socketio.Middleware(sio, app)
50         eventlet.wsgi.server(eventlet.listen(('', 4567)), app)
```

REFERENCES

[1] Dan Neil. “Who’s Behind the Wheel? Nobody. The driverless car is coming. And we all should be glad it is”. In: *Wall Street Journal (Online)[New York, NY] Sep* 24 (2012).

[2] D Liden. “What Is a Driverless Car?[on-line]”. In: *Elérhetőség: http://www. wisegeek. com/what-is-a-driverless-car. htm[Letöltve: 2018.04. 02.]* (2017).

[3] Péter Szikora and Nikolett Madarász. “Self-driving cars—The human side”. In: *2017 IEEE 14th International Scientific Conference on Informatics*. IEEE. 2017, pp. 383–387.

[4] Choton Basu, Manu Madan, and Balaji Sankaranarayanan. “ARE DRIVERLESS CARS TRULY A REALITY? A TECHNICAL, SOCIAL AND ETHICAL ANALYSIS.” In: *Issues in Information Systems* 20.4 (2019).

[5] DP Howley. “The Race to Build Self-Driving Cars”. In: *Forrás: Laptop Magazine: http://blog. laptopmag. com/high-tech-cars-go-mainstream-self-driving-in-car-radar-more* (2012).

[6] Brandon Schoettle and Michael Sivak. *Motorists’ preferences for different levels of vehicle automation*. Tech. rep. University of Michigan, Ann Arbor, Transportation Research Institute, 2015.

[7] Loz Blain. “Self-driving vehicles: What are the six levels of autonomy?” In: *Forrás: Newatlas. com: http://newatlas. com/sae-autonomous-levels-definition-self-driving/49947* (2017).

[8] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[10] Lawrence D Jackel, David Sharman, Charles E Stenard, B Ivan Strom, and Derek Zuckert. “Optical character recognition for self-service banking”. In: *AT&T technical journal* 74.4 (1995), pp. 16–24.

[11] Alex Berg, Jia Deng, and L Fei-Fei. “Large scale visual recognition challenge (ILSVRC), 2010”. In: *URL http://www. image-net. org/challenges/LSVRC* 3 (2010).

[12] Y Lecun, E Cosatto, J Ben, U Muller, and B Flepp. “Dave: Autonomous off-road vehicle control using end-to-end learning”. In: *Courant Institute/CBLL, http://www. cs. nyu. edu/yann/research/dave/index. html, Tech. Rep. DARPA-IPTO Final Report* (2004).

[13] Dean A Pomerleau. “Alvinn: An autonomous land vehicle in a neural network”. In: *Advances in neural information processing systems*. 1989, pp. 305–313.

[14] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. “End to end learning for self-driving cars”. In: *arXiv preprint arXiv:1604.07316* (2016).

[15] Danwei Wang and Feng Qi. “Trajectory planning for a four-wheel-steering vehicle”. In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*. Vol. 4. IEEE. 2001, pp. 3320–3325.

[16] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. “The German Traffic Sign Recognition Benchmark: A multi-class classification competition”. In: *The 2011 International Joint Conference on Neural Networks*. 2011, pp. 1453–1460.

[17] A. de la Escalera, L. E. Moreno, M. A. Salichs, and J. M. Armingol. “Road traffic sign detection and classification”. In: *IEEE Transactions on Industrial Electronics* 44.6 (1997), pp. 848–859.

[18] James Hedlund. “Autonomous vehicles meet human drivers: Traffic safety issues for states”. In: (2017).

[19] L. Chen, G. Zhao, J. Zhou, and L. Kuang. “Real-Time Traffic Sign Classification Using Combined Convolutional Neural Networks”. In: *2017 4th IAPR Asian Conference on Pattern Recognition (ACPR)*. 2017, pp. 399–404.

[20] Aleksandar Angelov, Andrew Robertson, Roderick Murray-Smith, and Francesco Fioranelli. “Practical classification of different moving targets using automotive radar and deep neural networks”. In: *IET Radar, Sonar & Navigation* 12.10 (2018), pp. 1082–1089.

[21] Zhenji Lu, Xander Coster, and Joost de Winter. “How much time do drivers need to obtain situation awareness? A laboratory-based study of automated driving”. In: *Applied ergonomics* 60 (2017), pp. 293–304.

[22] J. Kocić, N. Jovičić, and V. Drndarević. “Driver behavioral cloning using deep learning”. In: *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*. 2018, pp. 1–5.

[23] Wael Farag and Zakaria Saleh. “Behavior Cloning for Autonomous Driving using Convolutional Neural Networks”. In: *2018 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*. IEEE. 2018, pp. 1–7.

[24] R. Kulic and Z. Vukic. “Autonomous Vehicle Obstacle Avoiding and Goal Position Reaching by Behavioral Cloning”. In: *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*. 2006, pp. 3939–3944.

[25] Alexander Buyval, Aidar Gabdullin, Ruslan Mustafin, and Ilya Shimchik. “Realtime vehicle and pedestrian tracking for didi udacity self-driving car challenge”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 2064–2069.

[26] Elizabeth Arentz. “Driving Miss Lazy: Autonomous Vehicles, Industry and the Law”. In: *Ohio St. Bus. LJ* 12 (2017), p. 221.