ABSTRACT

Unfathomable as it may seem, the world is definitely heading to an age of Self driving cars that can replace human drivers. Imagine a world with no car crashes. Our self-driving vehicles aim to eliminate human driver error — the primary cause of 94 percent of crashes — leading to fewer injuries and fatalities. Imagine widespread use of electric-only vehicles, reducing vehicle emissions. Our self-driving vehicles will all be electric, contributing to a better environment. Imagine not sitting in traffic for what feels like half your life. And imagine a crowded city not filled with congested roads and parking lots and structures but with efficiently moving traffic and more space. Our self-driving vehicles will improve access to mobility for those who currently cannot drive due to age, disability, or otherwise. In this project, we are going to design a system that drives a car autonomously in a simulated environment using keras deep learning library. As an input, we used Udacity Self Driving Car Simulator[1] to generate input data. We use keyboard to drive around the circuit. The simulator records screen shots of the images and the decisions we made: steering angles, throttle and brake, while driving the car in simulator. The mission is to create a deep neural network to emulate the behavior of the human being, and the put this network to run the simulator autonomously.

INTRODUCTION

It is important to understand that a lot of planning has to go into designing a system that drives a car autonomously in a simulated environment. Building a complete neural network model, before implementing it with the simulator is very important in a project that involves precise integration between our model and the simulator. An equally important aspect is gaining as much background knowledge of neural networks and keras.

In order to get a good overview about keras and neural networks, my professor took an intense care on me and gave an Introductory session to me with his student, which makes me to get familiar with keras and neural networks. In order to have good understanding on keras and the concepts of neural networks, my professor encouraged me to do some basic projects. By doing multiple basic projects, it helped me gain an overall picture of what goes into designing a system that drives a car autonomously in a simulated environment from scratch.

This part of my project involved designing a system that drives a car autonomously in a simulated environment. In our model, the architecture has been inspired from paper, End to End Learning for Self-Driving Cars published[2] by NVIDIA[3]. In order to design our model to make it suitable for all driving conditions, we are going to use data augmentation technique to train our model, which produces the steering angle as output.

In order to generate input data, we used Udacity self driving car simulator. We used keyboard to drive around the circuit. The simulator records screen shots of the images and the decisions we made: steering angles, throttle and brake, while driving the car in simulator

To design our model, we used keras deep learning library to build our neural network and used OpenCV for data augmentation in python programming language.

Once the model is trained, we used flask, a web application frame work to deploy our model in Udacity self driving car simulator and to drive our car autonomously.

Problem Statement

Irrespective of what our objective is, while working on any project, we should be able to provide a fully legitimate final “soft copy” of a project. This ensures
that every single part of the project is thought through to completion and no gaps are left logistically. This particular project provides a complexity in input data and integration between our trained model and simulator.

We collected data on a bright beautiful day, what if it’s dark or raining when our autonomous car runs on a street, that wouldn’t be good because our car can be confused.

While designing our model, we should make sure that our trained model is suitable to integrate with Udacity self driving car simulator.

LITERATURE OVERVIEW

The Literature Review is split into the six components:
They are
(1) Deep Learning
(2) Computer Vision
(3) Python
(4) keras
(5) End-to-End Deep Learning for Self-Driving Cars
(6) Udacity Self Driving Car Simulator

1. Deep Learning

Deep learning is a subset of a subset of the way that Machine's think. If we want to know about Deep learning, we need to have a basic idea on Artificial Intelligence and Machine Learning. So, what is AI and Machine Learning.

Artificial intelligence

Artificial intelligence is the broadest term we use for technology that allows machines to mimic human behavior. AI can include logic, if-then rules, and more.

Machine Learning

One subset of AI technology is machine learning. Machine learning allows computers to use statistics and use the experience to improve at tasks.

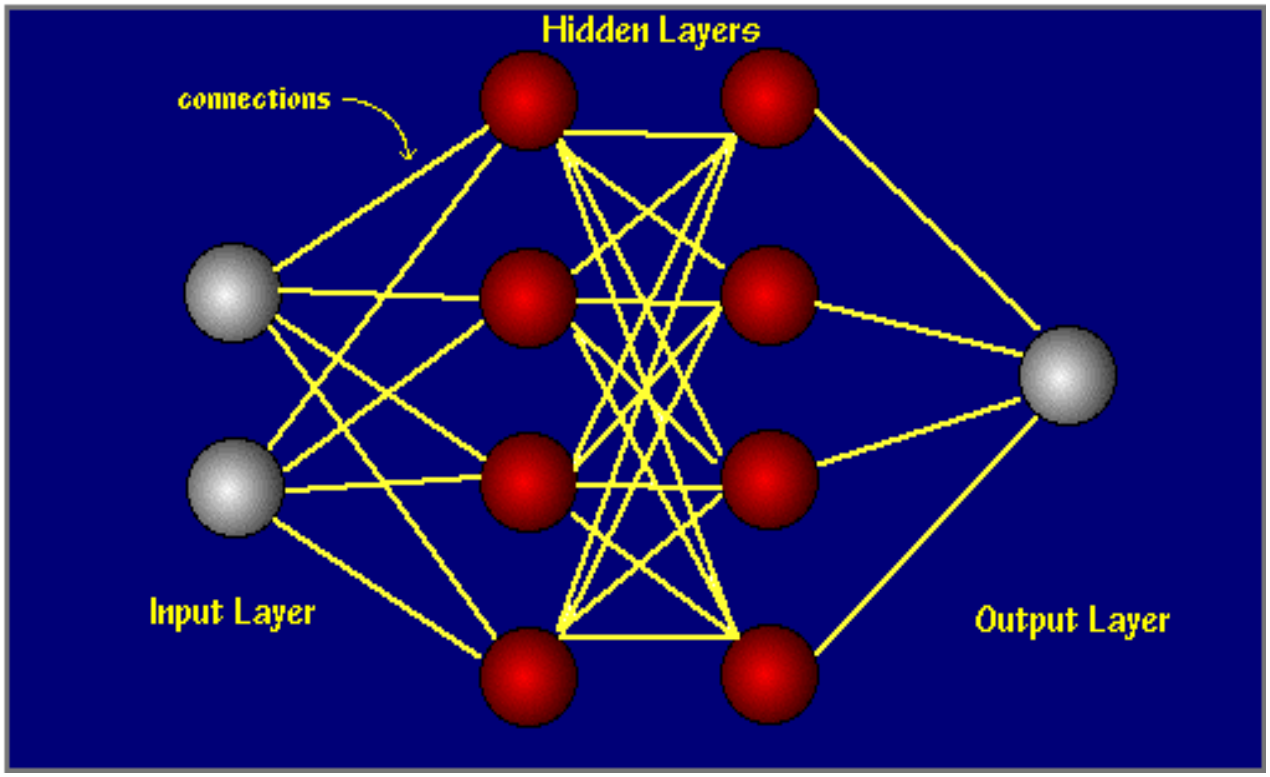
Deep Learning

A subset of machine learning is deep learning, which uses algorithms to train computers to perform tasks by exposing neural nets to huge amounts of data, allowing them to predict responses without needing to actually complete every task.

Neural Networks

Neural networks are typically organized in layers. Layers are made up of a number of interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer',

which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted 'connections'. The hidden layers then link to an 'output layer' where the answer is output as shown in the graphic below.

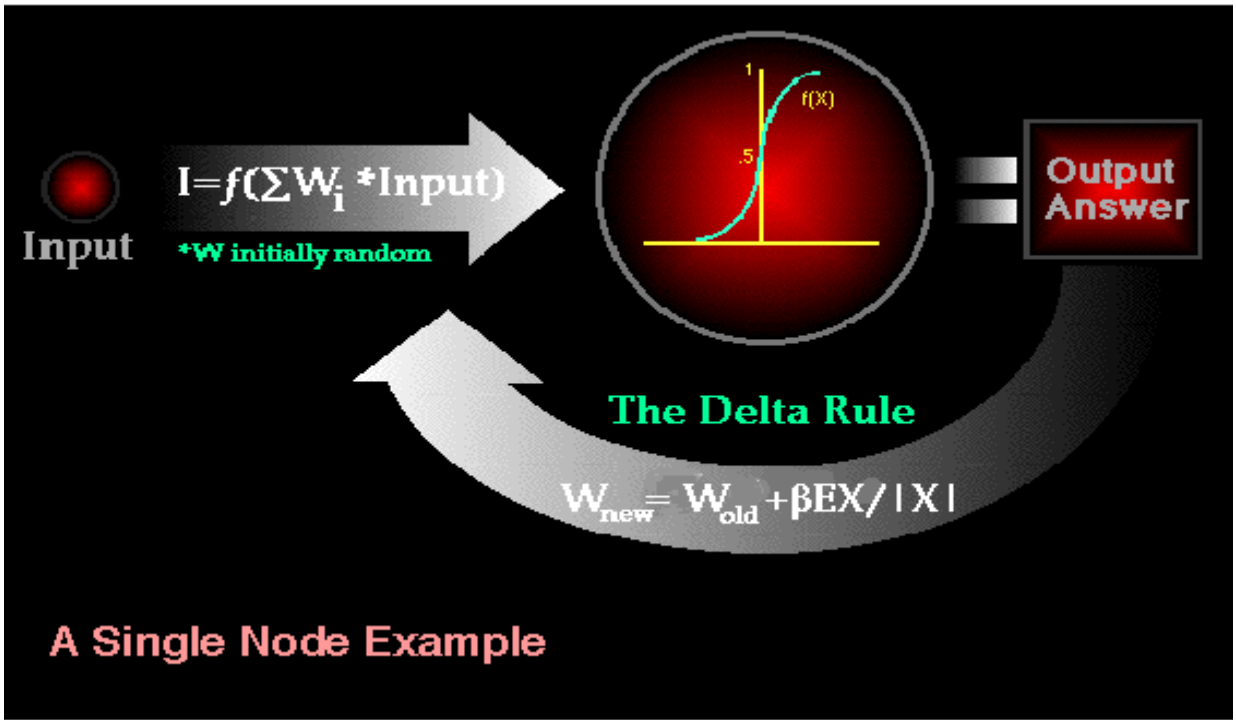


source : http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Most ANNs[1] contain some form of 'learning rule' which modifies the weights of the connections according to the input patterns that it is presented with. In a sense, ANNs learn by example as do their biological counterparts; a child learns to recognize dogs from examples of dogs. Although there are many different kinds of learning rules used by neural networks, this demonstration is concerned only with one; the delta rule. The delta rule is often utilized by the most common class of ANNs called BPNNs[2]. Back propagation is an abbreviation for the backwards propagation of error. With the delta rule, as with other types of back propagation, 'learning' is a supervised process that occurs with each

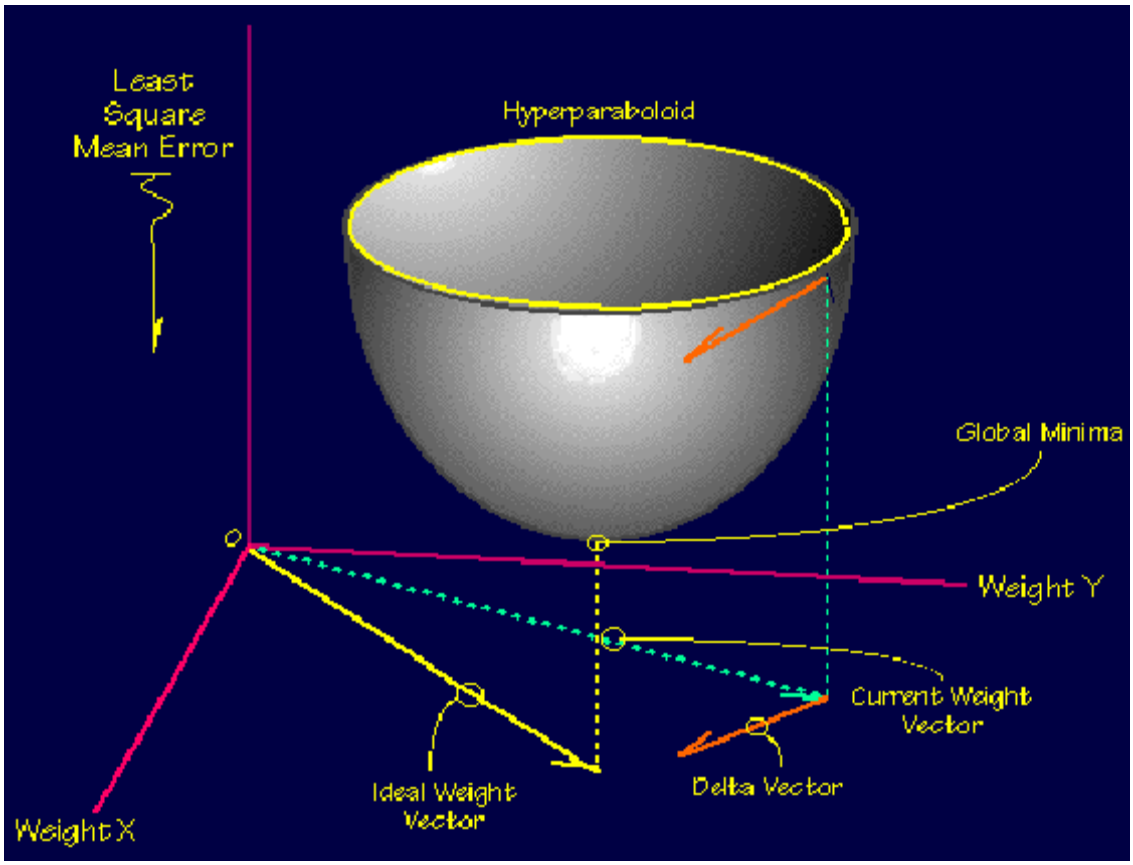
1 ANNs – Artificial Neural Network
2 BPNNs-Backpropagation Neural Network

cycle or 'epoch' (i.e. each time the network is presented with a new input pattern) through a forward activation flow of outputs, and the backwards error propagation of weight adjustments. More simply, when a neural network is initially presented with a pattern it makes a random 'guess' as to what it might be. It then sees how far its answer was from the actual one and makes an appropriate adjustment to its connection weights. More graphically, the process looks something like figure below:



source : http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Note also, that within each hidden layer node is a sigmoidal activation function which polarizes network activity and helps it to stablize. Back propagation performs a gradient descent within the solution's vector space towards a 'global minimum' along the steepest vector of the error surface. The global minimum is that theoretical solution with the lowest possible error. The error surface itself is a hyper paraboloid but is seldom 'smooth' as is depicted in the graphic below. Indeed, in most problems, the solution space is quite irregular with numerous 'pits' and 'hills' which may cause the network to settle down in a 'local minum' which is not the the best overall solution, which is in the below figure.



source : http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Since the nature of the error space can not be known a prioi, neural network analysis often requires a large number of individual runs to determine the best solution. Most learning rules have built-in mathematical terms to assist in this process which control the 'speed' (Beta-coefficient) and the 'momentum' of the learning. The speed of learning is actually the rate of convergence between the current solution and the global minimum. Momentum helps the network to overcome obstacles (local minima) in the error surface and settle down at or near the global minimum.

Once a neural network is 'trained' to a satisfactory level it may be used as an analytical tool on other data. To do this, the user no longer specifies any training runs and instead allows the network to work in forward propagation mode only. New inputs are presented to the input pattern where they filter into and are processed by the middle layers as though training were taking place, however, at this point the output is

retained and no back propagation occurs. The output of a forward propagation run is the predicted model for the data which can then be used for further analysis and interpretation. It is also possible to over-train a neural network, which means that the network has been trained exactly to respond to only one type of input; which is much like rote memorization. If this should happen then learning can no longer occur and the network is referred to as having been "grand mothered" in neural network jargon. In real
world applications this situation is not very useful since one would need a separate grand mothered network for each new kind of input. We can find more information in this page[4].

2. Computer Vision

The computer vision part of the project is accomplished using OpenCV. This is a library which contains functions that are used to achieve real-time computer vision. It was released by Intel in 1999 to simplify computations that were too intensive for a computer. It started off with the simple applications of ray tracing and 3D display walls. Documentation for OpenCV3, which is what we
will be using in this project is available in the respective page. The main focus and use of computer vision in this project to rotate, translate, flip, shift, transform to darker or to brighter. To achieve this, inspiration was taken from this post[3] by Vivek Yadav.

3. Python

Installation of Python is required for this project. Python is a programming language that is dynamic and object oriented. It emphasises on readability of the code and thereby, features indentation to delimit blocks of code. Python can be downloaded from official python website[5]. Version 3.5.2 has been used in our project.

---

3 augmentation - https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9

4. Keras

In this project, we are going to use keras deep learning library to create a model. Keras is a high-level neural networks API, written in Python and capable of running on top of Tensor Flow, CNTN or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. We can find more information in keras documentation[4]

5. End-to-End Deep Learning for Self-Driving Car

In a new automotive application, NVIDIA have used convolutional neural networks (CNNs) to map the raw pixels from a front-facing camera to the steering commands for a self-driving car. This powerful end-to-end approach means that with minimum training data from humans, the system learns to steer, with or without lane markings, on both local roads and highways. The system can also operate in areas with unclear visual guidance such as parking lots or unpaved roads.

Figure1 : NVIDIA's self-driving car in action

4 keras Documentation - https://keras.io/

They designed the end-to-end learning system using an NVIDIA DevBox running Torch 7 for training. An NVIDIA DRIVE™ PX self-driving car computer, also with Torch 7, was used to determine where to drive—while operating at 30 frames per second (FPS). The system is trained to automatically learn the internal representations of necessary processing steps, such as detecting useful road features, with only the human steering angle as the training signal. They never explicitly trained it to detect, for example, the outline of roads. In contrast to methods using explicit decomposition of the problem, such as lane marking detection, path planning, and control, our end-to-end system optimizes all processing steps simultaneously.

They believe that end-to-end learning leads to better performance and smaller systems. Better performance results 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.

Convolutional Neural Networks to Process Visual Data

CNNs[5] have revolutionized the computational pattern recognition process[6]. 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 important breakthrough of CNNs is that features are now learned automatically from training examples. The CNN approach is especially powerful when applied to image recognition tasks because the convolution operation captures the 2D nature of images. By using the convolution kernels to scan an entire

5 CNN – Convolutional Neural Network

image, relatively few parameters need to be learned compared to the total number of operations.

While CNNs with learned features have been used commercially for over twenty years[7],their adoption has exploded in recent years because of two important developments. First, large, labeled data sets such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [8] are now widely available for training and validation. Second, CNN learning algorithms are now implemented on massively parallel graphics processing units (GPUs), tremendously accelerating learning and inference ability.

The CNNs that we describe here go beyond basic pattern recognition. We developed a system that learns the entire processing pipeline needed to steer an automobile. The groundwork for this project was actually done over 10 years ago in a Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE)[9], in which a sub-scale radio control (RC) car drove through a junk-filled alley way. DAVE[6] was trained on hours of human driving in similar, but not identical, environments. The training data included video from two cameras and the steering commands sent by a human operator.

In many ways, DAVE was inspired by the pioneering work of Pomerleau[10], who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. ALVINN[7] is a precursor to DAVE, and it provided the initial proof of concept that an end-to-end trained neural network might one day be capable of steering a car on public roads. 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[11], but DAVE's performance was not sufficiently reliable to provide a full alternative to the more modular approaches to off-road driving. (DAVE's mean distance between crashes was about 20 meters in complex environments.)

---

6 DAVE – DAPRA Autonomous Vehicle

7 ALVINN – Autonomous Land Vehicle in a Neural Network

About a year ago we started a new effort to improve on the original 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. We are excited to share the preliminary results of this new effort, which is aptly named: DAVE–2.

The DAVE-2 System

Figure 2 shows a simplified block diagram of the collection system for training data of DAVE-2. Three cameras are mounted behind the windshield of the data-acquisition car, and time stamped video from the cameras is captured simultaneously with the steering angle applied by the human driver. The 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).

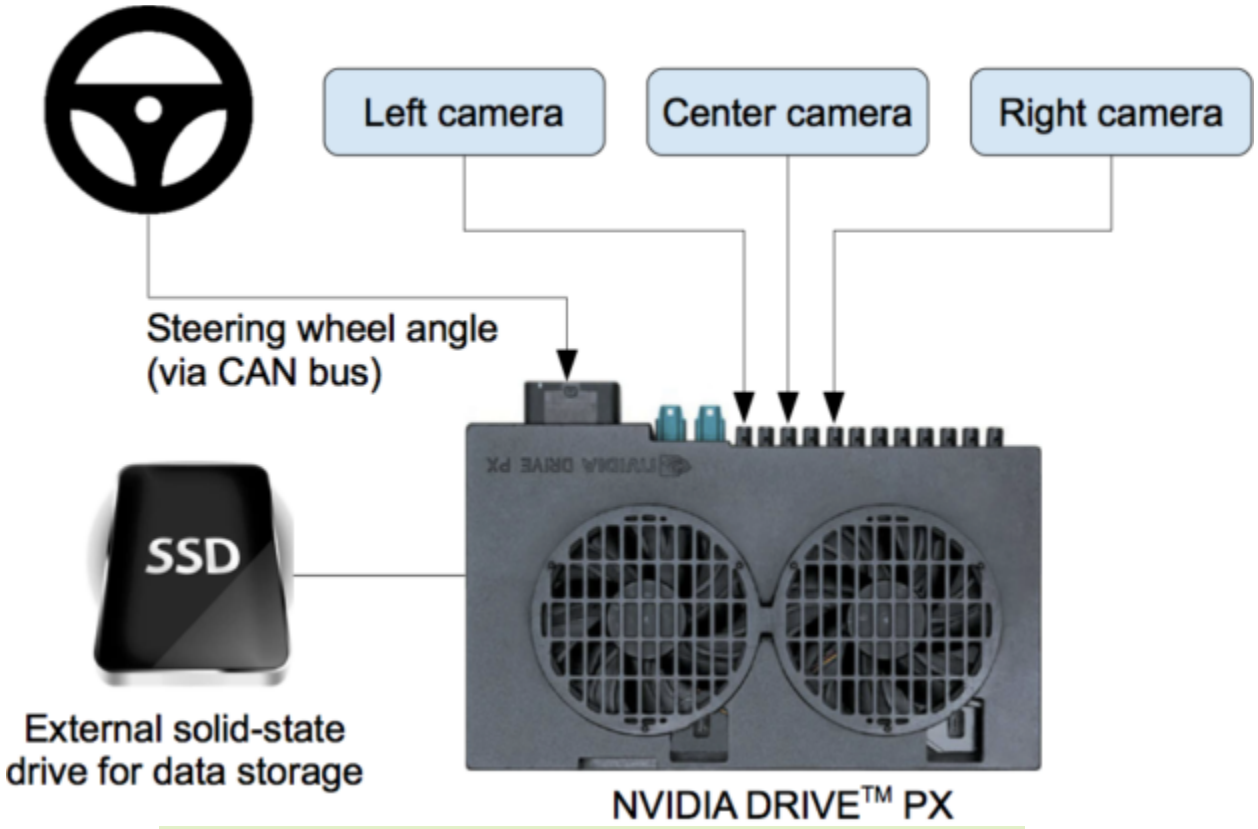


Figure 2: High-level view of the data collection system.

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 also learn how to recover from any mistakes, or 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.

The images for two specific off-center shifts can be obtained from the left and the right cameras. Additional shifts between the cameras and all rotations are simulated through viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires 3D scene knowledge which we don't have, so we 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 for a more complete rendering it

introduces distortions for objects that stick above the ground, such as cars, poles, trees, and buildings. Fortunately these distortions don't pose a significant problem for network training. The steering label for the transformed images is quickly adjusted to one that correctly steers the vehicle back to the desired location and orientation in two seconds.
Figure 3 shows a block diagram of our training system. Images are fed into a CNN that then computes a proposed steering command. The proposed command is compared to the desired command for that image, and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. The weight adjustment is accomplished using back propagation as implemented in the Torch 7 machine learning package.

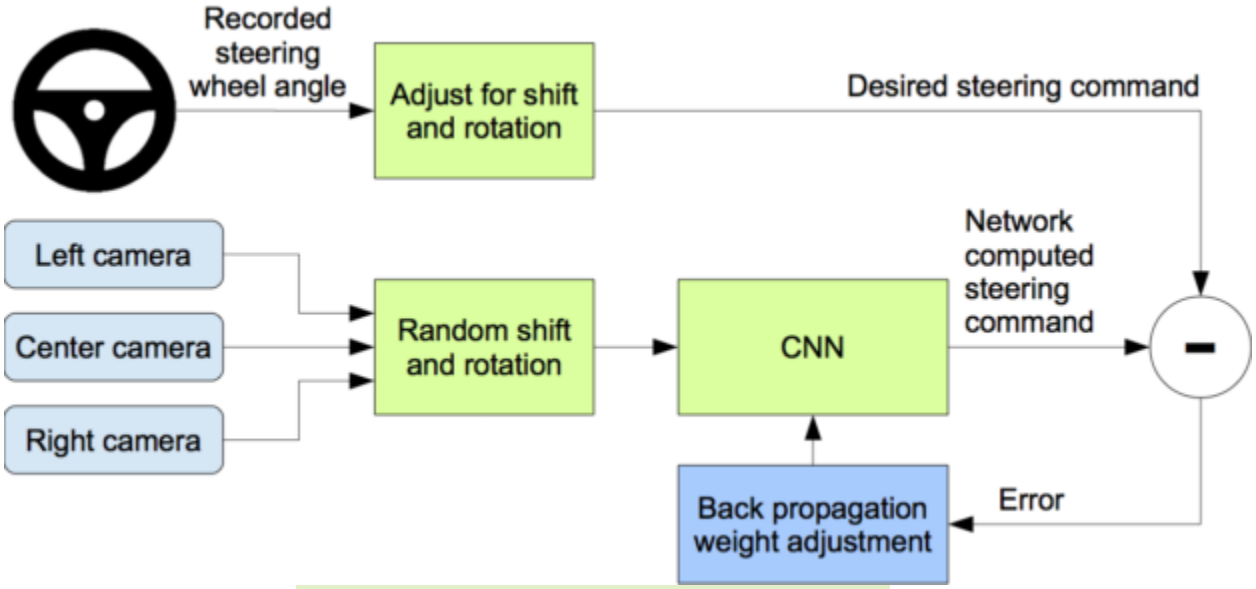


Figure 3: Training the neural network.

Once trained, the network is able to generate steering commands from the video images of a single center camera. Figure 4 shows this configuration.

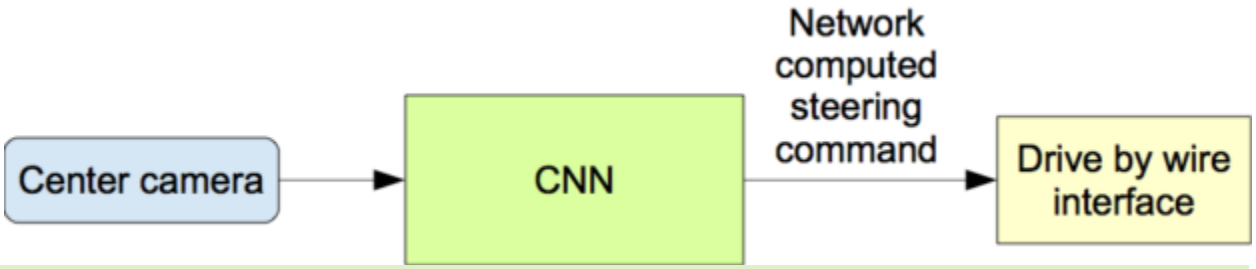


Figure 4: The trained network is used to generate steering commands from a single front-facing center camera.

Data Collection

Training data was collected by driving on a wide variety of roads and in a diverse set of lighting and weather conditions. They gathered surface street data in central New Jersey and highway data from Illinois, Michigan, Pennsylvania, and New York. Other road types include two-lane roads (with and without lane markings), residential roads with parked cars, tunnels, and unpaved roads. Data was collected in clear, cloudy, foggy, snowy, and rainy weather, both day and night. In some instances, the sun was low in the sky, resulting in glare reflecting from the road surface and scattering from the windshield.

The data was acquired using either our drive-by-wire test vehicle, which is a 2016 Lincoln MKZ, or using a 2013 Ford Focus with cameras placed in similar positions to those in the Lincoln. System has no dependencies on any particular vehicle make or model. Drivers were encouraged to maintain full attentiveness, but otherwise drive as they usually do. As of March 28, 2016, about 72 hours of driving data was collected.

Network Architecture

They train the weights of our network to minimize the mean-squared error between the steering command output by the network, and either the command of the human driver or the adjusted steering command for off-center and rotated images (see “Augmentation”, later). Figure 5 shows the network architecture, which 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.

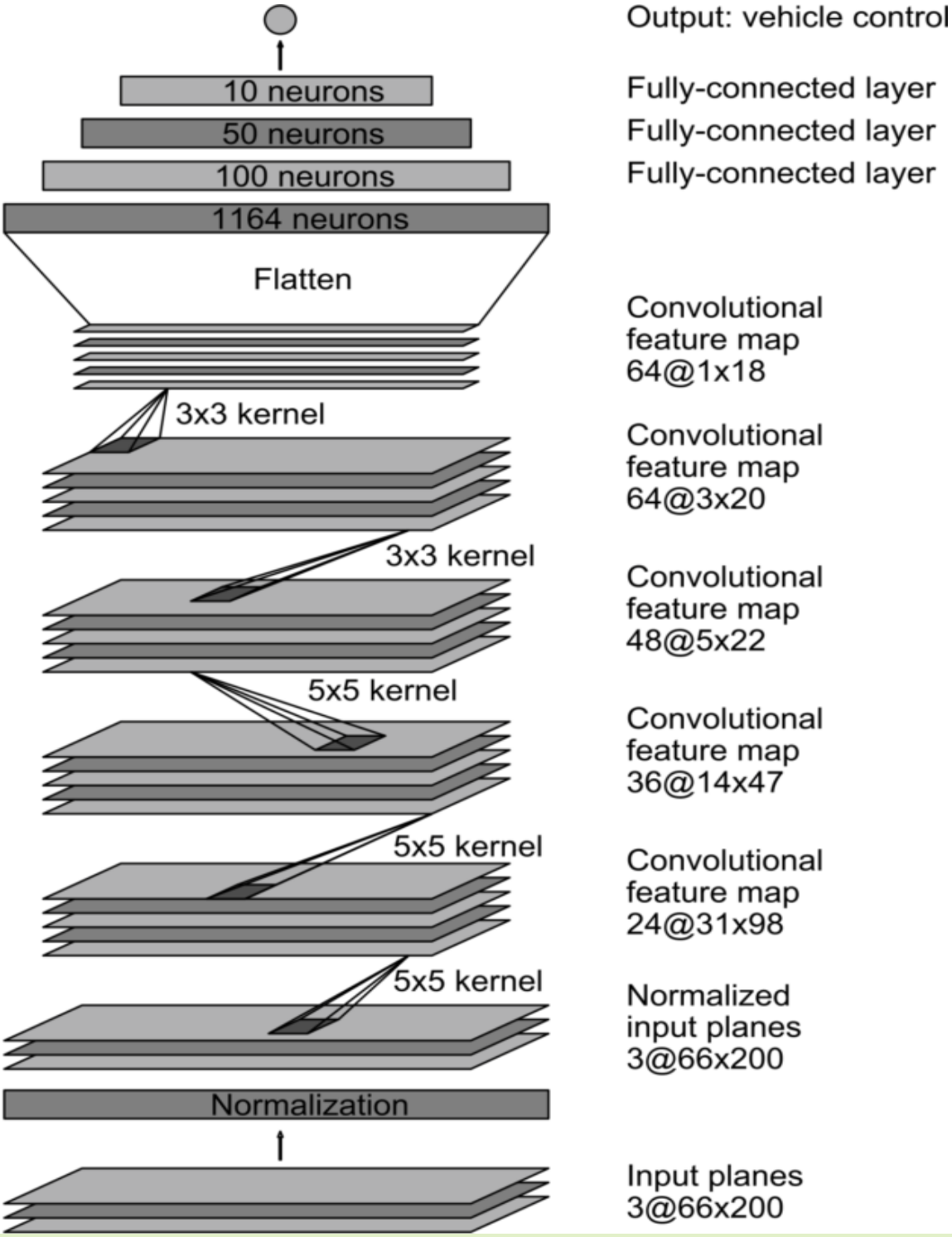


Figure 5: CNN architecture. The network has about 27 million connections and 250 thousand parameters

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 are designed to perform feature extraction, and are chosen empirically through a series of experiments that vary layer configurations. They then 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 final two convolutional layers.
They follow the five convolutional layers with three fully connected layers, leading to a final output control value which is the inverse-turning-radius. The fully connected layers are designed to function as a controller for steering, but we noted 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.

Data Selection

The first step to training a neural network is selecting the frames to use. 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, simply select data where the driver is staying in a lane, and discard the rest. They then sample that video at 10 FPS because a higher sampling rate would include images that are highly similar, and thus not provide much additional useful information. To remove a bias towards driving straight the training data includes a higher proportion of frames that represent road curves.

Augmentation

After selecting the final set of frames, They augment the data by adding artificial shifts and rotations to teach the network how to recover from

a poor position or orientation. The magnitude of these perturbations is chosen randomly from a normal distribution. The distribution has zero mean, and the standard deviation is twice the standard deviation that we measured with human drivers. Artificially augmenting the data does add undesirable artifacts as the magnitude increases (as mentioned previously).

Simulation

Before road-testing a trained CNN, They first evaluate the network's performance in simulation. Figure 6 shows a simplified block diagram of the simulation system, and Figure 7 shows a screenshot of the simulator in interactive mode.

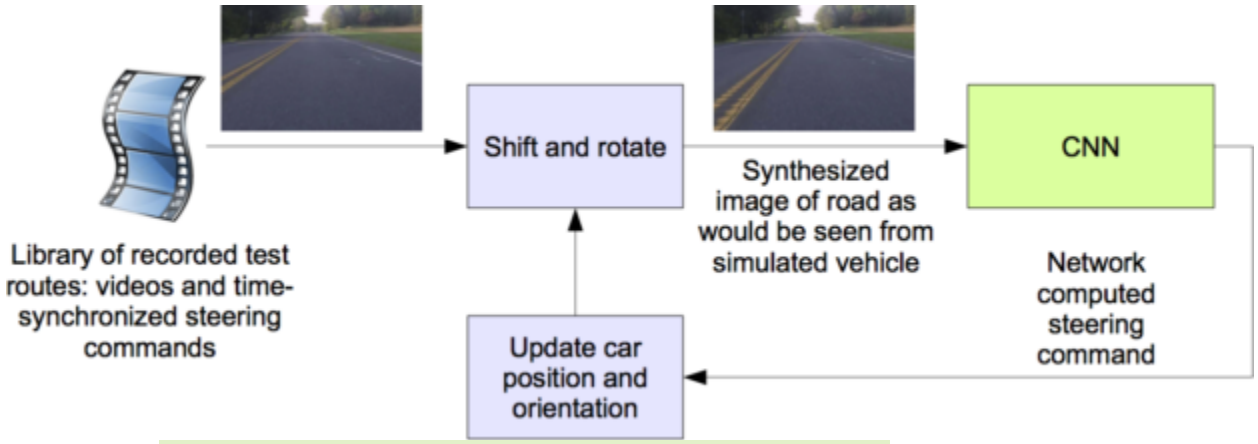


Figure 6: Block-diagram of the drive simulator.

The simulator takes prerecorded videos from a forward-facing on-board camera connected to a human-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 the recorded steering commands generated by the human driver.
Since human drivers don't drive in the center of the lane all the time, they must manually calibrate the lane's center as it is associated with each frame in the video used by the simulator. They 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 human driven path and the ground truth. The transformation is accomplished by the same methods as described previously.

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, which then returns a steering command for that frame. The CNN steering commands as well as the recorded human-driver commands are fed into the dynamic model[12] of the vehicle to update the position and orientation of the simulated vehicle.

Figure 7: Screenshot of the simulator in interactive mode. See text for explanation of the performance metrics. The green area on the left is unknown because of the viewpoint transformation. The highlighted wide rectangle below the horizon is the area which is sent to the CNN.

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 human 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.

They evaluated the networks in two steps: first in simulation, and then in on-road tests.
In simulation they have the networks provide steering commands in our simulator to an ensemble of prerecorded test routes that correspond to about a total of three hours and 100 miles of driving in Monmouth County, NJ. The test data was taken in diverse lighting and weather conditions and includes highways, local roads, and residential streets.
We estimate what percentage of the time the network could drive the car (autonomy) by counting the simulated human interventions that occur when the simulated vehicle departs from the center line by more than one meter. They assume that in real life an actual intervention would require a total of six seconds: this is the time required for a human to retake control of the vehicle, re-center it, and then restart the self-steering mode. We calculate the percentage autonomy by counting the number of interventions, multiplying by 6 seconds, dividing by the elapsed time of the simulated test, and then subtracting the result from 1:

$$\text{autonomy} = (1 - \frac{\text{\# of interventions} \cdot 6[\text{seconds}]}{\text{elapsed time}[\text{seconds}]}) \cdot 100$$

Thus, if we had 10 interventions in 600 seconds, we would have an autonomy value of

$$(1 - \frac{10 \cdot 6}{600}) \cdot 100 = 90\%$$

On-road Tests

After a trained network has demonstrated good performance in the simulator, the network is loaded on the DRIVE PX in our test car and taken out for a road test. For these tests they measure performance as the fraction of time during which the car performs autonomous steering. This time excludes lane changes and turns from one road to another. For a typical drive in Monmouth County NJ from our office in Holmdel to Atlantic Highlands, we are autonomous approximately 98% of the time. They also drove 10 miles on the Garden State Parkway (a multi-lane divided highway with on and off ramps) with zero intercepts. You can see video here[13]

6. Udacity Self Driving Car Simulator

Udacity recently made its self-driving car simulator source code available on their GitHub which was originally build to teach their self-Driving Car Engineer Nanodegree students. Now, anybody can take advantage of the useful tool to train your machine learning models to clone driving behavior.

We can manually drive a car to generate training data, or machine learning model can autonomously drive for testing.
In the main screen of the simulator, we can choose a scene and mode. Figure below is Simulator main screen.

First, you choose a scene by clicking one of the scene pictures. In above, the lake side scene (the left) is selected. Next, we choose a mode: Training Mode or Autonomous Mode. As soon as you click one of the mode buttons, a car appears at the start position.

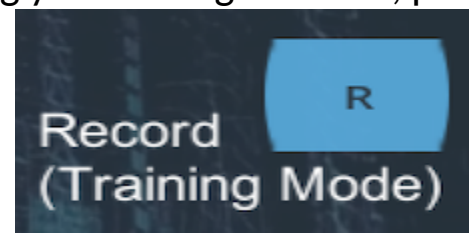
simulator train mode

Training Mode

In the training mode, you drive the car manually to record the driving behavior. You can use the recorded images to train your machine learning model. To drive the car, use the following keys:

If you like using mouse to direct the car, you can do so by dragging:

To start a recording your driving behavior, press R on your keyboard.



You can press R again to stop the recording.

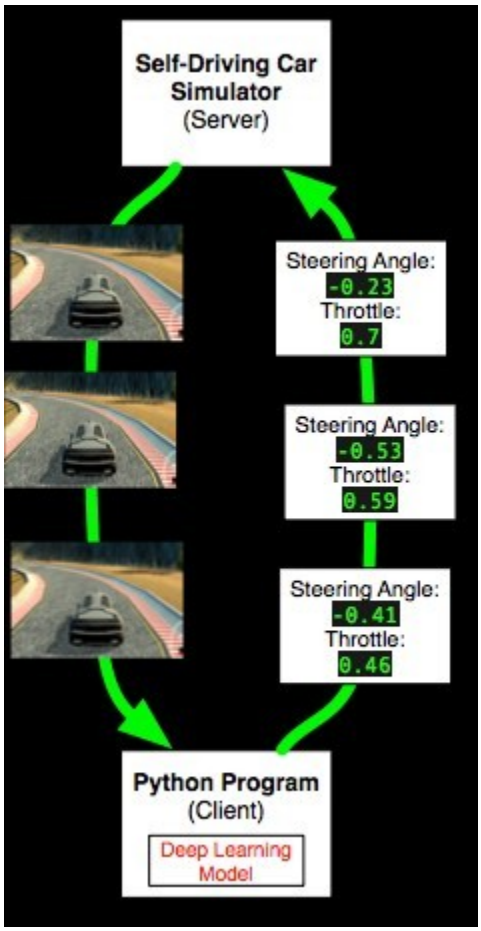
Finally, use ESC to exit the training mode.

You can see the driving instructions any time by clicking **CONTROLS** button in the top right of the main screen.

Autonomous Mode

In the autonomous mode, you are testing your machine learning model to see how well your model can drive the car without dropping off the road / falling into the lake.

Technically, the simulator is acting as a server where your program can connect to and receive a stream of image frames from.



Simulator (Server) <-> Model (Client)

For example, your Python program can use a machine learning model to process the road images to predict the best driving instructions, and send them back to the server.

Each driving instruction contains a steering angle and an acceleration throttle, which changes the car's direction and the speed (via acceleration). As this happens, your program will receive new image frames at real time.

SOFTWARE REQUIREMENTS

a. Python

Python is a widely used high-level, general-purpose,
interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.
To install python using command line. In this project we should use python version 3.5.2 only. In order to change the version of python using command line, use

$ alias python3 = python 3.5.2

For more information, you can see this documentation[14].

b. Pandas

Pandas is an open source, BSD-licensed library providing
high-performance, easy-to-use data structures and data
analysis tools for the Python programming language.
To install pandas using command line, use

$ pip3 install pandas

For more information, you can see this documentation[15].

c. Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

a powerful N-dimensional array object, sophisticated (broadcasting) functions and tools for integrating C/C++ and Fortran code
To install Numpy using command line, use

$ pip3 install numpy

For more information, you can see this documentation[16].

d. Matplotlib

Matplotlib is a Python 2D plotting library which produces
publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and Ipython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.
To install matplotlib using command line, use

$ pip3 install matplotlib

For more information, you can see this documentation[17].

f. Scikit - learn

Scikit - Learn It is used for machine learning in python. It is simple and efficient tools for data mining and data analysis. It is accessible to everybody, and reusable in various contexts. It can be built on NumPy, SciPy, and matplotlib
To install scikit – learn using command line, use

$ pip3 install scikit-learn

for more information, you can see this documentation[18].

g. OpenCV

OpenCV (Open Source Computer Vision Library) is released under a BSD license and hence it's free for both academic and commercial use. It has C++, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. Written in optimized C/C++, the library can take advantage of multi-core processing. Enabled with OpenCL, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform.

Adopted all around the world, OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 14 million. Usage ranges from interactive art, to mines inspection, stitching maps on the web or through advanced robotics.

To install opencv using command line, use

$ sudo apt-get install python-opencv

For detailed installation steps, see this guide lines[19]
For more information, you can see this documentation[20].

h. Pillow

The **Python Imaging Library** adds image processing capabilities to your Python interpreter. This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities. The core image library is designed for fast access to data stored in a few basic pixel formats. It should provide a solid foundation for a general image processing tool.

To install pillow using command line argument, use

$ pip3 install pillow

For more information, you can see this documentation[21].

i. Scikit-image

Scikit-image is a collection of algorithms for Image processing. It is available free of charge and free of restriction. It is high quality, per reviewed code, written by an active community of volunteers.
To install scikit-image using command line arguments, use

$ sudo apt-get install python-skimage

For more information, you can see this documentation[22].

j. Scipy

SciPy (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, some of the core packages are Numpy, SciPy library, Matplotlib, IPython, Sympy, pandas.
To install scipy using command line arguments, use

$ sudo apt-get install python-numpy python-scipy python-matplotlib ipython ipython-notebook python-pandas python-sympy python-nose

For more information, you can see this documentation[23].

k. h5py

The h5py package is a Pythonic interface to the HDF5 binary data format. It lets you store huge amounts of numerical data, and easily manipulate that data from NumPy. For example, you can slice into multi-terabyte datasets stored on disk, as if they were real NumPy

arrays. Thousands of datasets can be stored in a single file, categorized and tagged however you want.
To install h5py using command line argument, use

$ pip3 install h5py

For more information, you can see this documentation[24].

l. Eventlet

Eventlet is a concurrent networking library for Python that allows you to change how you run your code, not how you write it. It uses epoll or kqueue or libevent for highly scalable non-blocking I/O. Coroutines ensure that the developer uses a blocking style of programming that is similar to threading, but provide the benefits of non-blocking I/O. The event dispatch is implicit, which means you can easily use Eventlet from the Python interpreter, or as a small part of a larger application.
To install Eventlet using command line arguments, use

$ pip3 install eventlet

For more information, you can see this documentation[25].

m. Flask-socketio

Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application can use any of the SocketIO official clients libraries in Javascript, C++, Java and Swift, or any compatible client to establish a permanent connection to the server.
To install it using command line arguments, use

$ pip3 install flask-socketio

For more information, you can see this documentation[26].

n. Seaborn

Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphic.
To install it using command line arguments, use

$ pip3 install seaborn

For more information, you can see this documentation[27].

o. Imageio

Imageio is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats. It is cross-platform, runs on Python 2.7 and 3.4+, and is easy to install.
To install it using command line arguments, use

$ pip3 install imageio

For more information, you can see this documentation[28].

p. Moviepy

MoviePy is a Python library for video editing: cutting, concatenations, title insertions, video compositing (a.k.a. non-linear editing), video processing, and creation of custom effects. MoviePy can read and write all the most common audio and video formats, including GIF, and runs on Windows/Mac/Linux, with Python 2.7+ and 3.

To install it using command line arguments, use

$ pip3 install moviepy

For more information, you can see this documentation[29].

q. Tensorflow

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization[30], it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

To install it using command line arguments, use

$ pip3 install tensorflow-gpu

For complete installation guide, follow the steps in this website[8].

For more information, you can see this documentation[31]

r. Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of Tensor Flow, CNTN or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. Keras allows us : For easy and fast prototyping (through user friendliness, modularity, and extensibility). Supports both convolutional

8 Tensor flow installation - https://www.tensorflow.org/tutorials/

networks and recurrent networks, as well as combinations of the two. To Runs seamlessly on CPU and GPU.

To install it using command line arguments, use

$ sudo pip3 install keras

For more information, you can see this documentation[32].

s. Udacity Self Driving Car Simulator

This simulator was built for Udacity's Self-Driving Car Nanodegree, to teach students how to train cars how to navigate road courses using deep learning. It is very easy to install. Please go to this page[9], follow the instructions and download these files[10].

9 Udacity Simulator Installation - https://github.com/udacity/self-driving-car-sim
10 Download Files for simulator - https://github.com/udacity/self-driving-car-sim.git

Methodology and Approach

Data Generation

As an input, we used Udacity Self Driving Car Simulator to generate input data. We use keyboard to drive around the circuit. The simulator records screen shots of the images and the decisions we made: steering angles, throttle and brake, while driving the car in simulator.

The data comprises of images as seen by the car at each point of time while navigating around the track, along with a CSV file, holding the entire data of the car (speed, break, throttle, images) for the entire navigation period. So the columns in the CSV are:

Centre camera image path	Left camera image path	Right camera image path	Steering angle	Speed	Throttle	Break

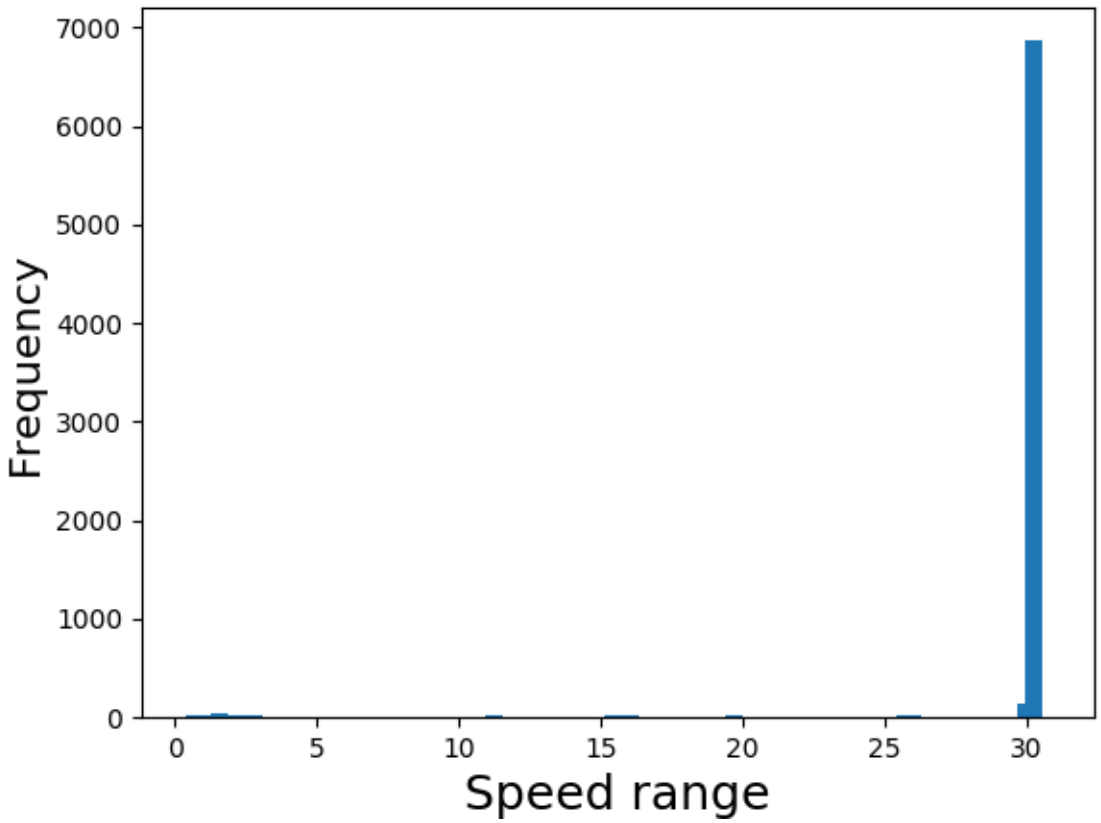
First three columns are just image paths to the images as seen by the car during navigation from three cameras.

Steering angle is the values of steering wheel, lies between -1 and 1.
Speed the speed of the car for that moment.

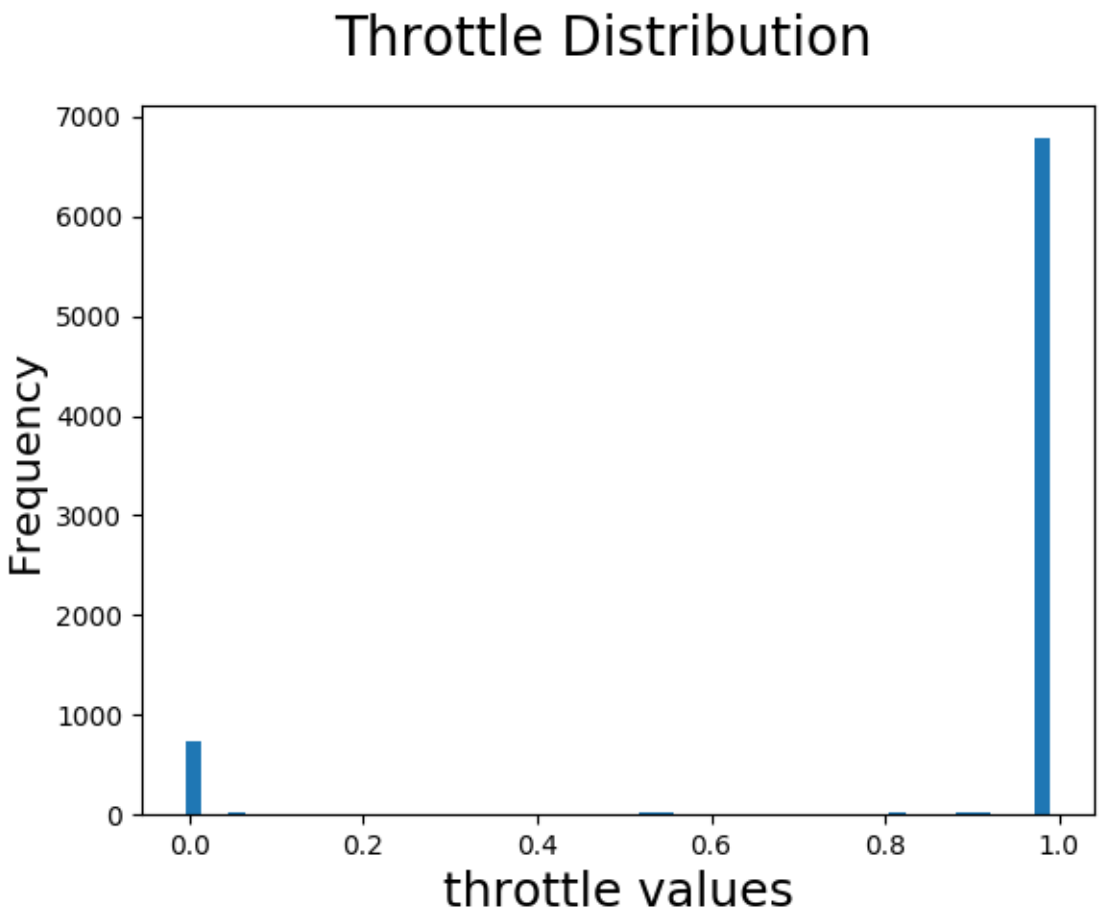
Data Distribution

By now, we are all too familiar with the tricks data can play. So, let's peek inside a bit more as to how our data looks like.

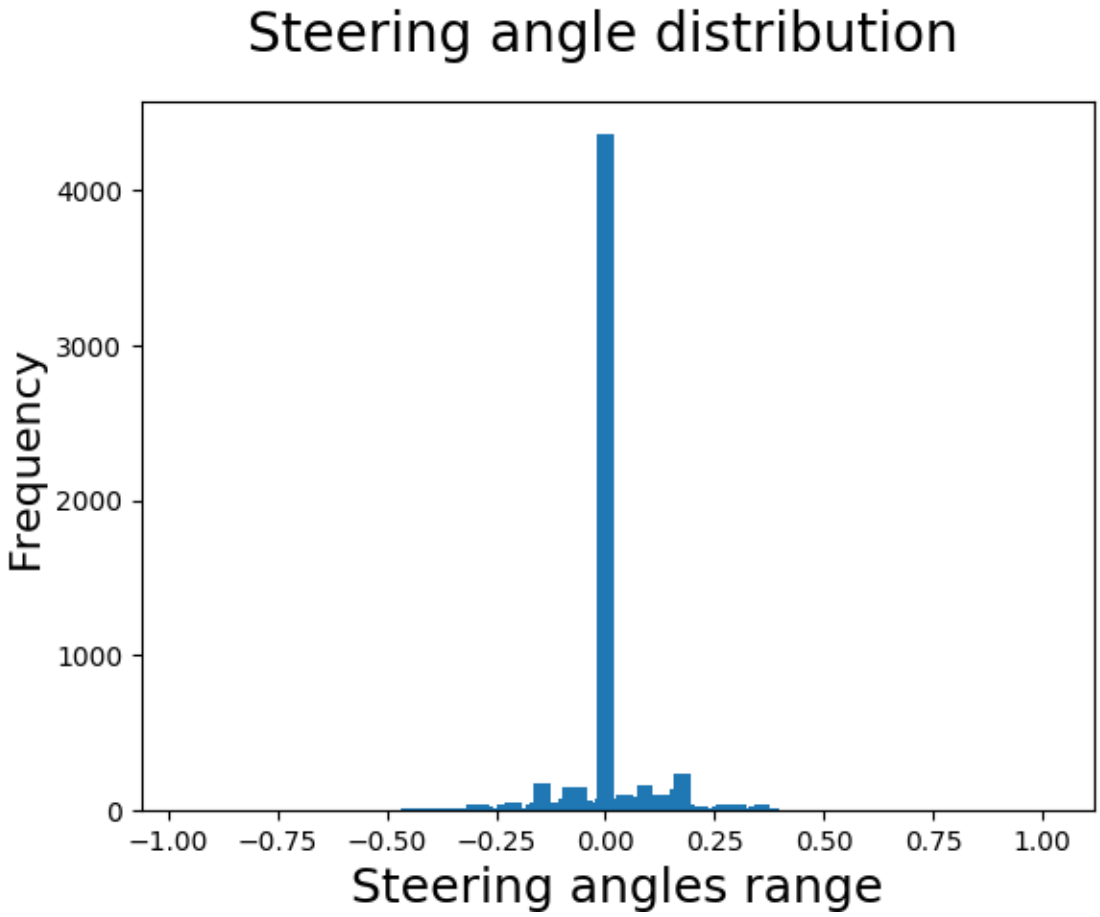
Let's look at the distribution of speed in our input data.


Speed Distribution
Frequency
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
30
Speed range

Let’s look at the distribution of throttle



We are primarily concern with the steering angle and images. Let's look at the distribution of the steering angle



The data is highly skewed and unbalanced.

The main task is to drive a car around in a simulator on a race track, and then use deep learning to mimic the behavior of human. This is a very interesting problem because it is not possible to drive under all possible scenarios on the track, so the deep learning algorithm will have to learn general rules for driving. We must be very careful while using deep learning models, because they have a tendency to overfit the data. Overfitting refers to the condition where the model is very sensitive to the training data itself, and the model's behavior does not generalize to new/unseen data. One way to avoid overfitting is to collect a lot of data.

A typical convolutional neural network can have up to a million parameters, and tuning these parameters requires millions of training instances of uncorrelated data, which may not always be possible and in some cases cost prohibitive. For our car example, this will require us to drive the car under different weather, lighting, traffic and road conditions. One way to avoid overfitting is to use augmentation.

Augmentation

Augmentation refers to the process of generating new training data from a smaller data set such that the new data set represents the real world data one may see in practice. As we are generating thousands of new training instances from each image, it is not possible to generate and store all these data on the disk. We will therefore utilize keras generators to read data from the file, augment on the fly and use it to train the model. We will utilize images from the left and right cameras so we can generate additional training data to simulate recovery. Keras generator is set up such that in the initial phases of learning, the model drops data with lower steering angles with higher probability. This removes any potential for bias towards driving at zero angle. After setting up the image augmentation pipeline, we can proceed to train the model. The training was performed using simple adam learning algorithm with learning rate of 0.0001. After this training, the model was able to drive the car by itself on the first track for hours and generalized to the second track.

All the training was based on driving data of about 4 laps using key board on track 1 in one direction alone. The model never saw track 2 in training, but with image augmentation (flipping, darkening, shifting, etc) and using data from all the cameras (left, right and center) the model

was able to learn general rules of driving that helped translate this learning to a different track.

Augmentation helps us extract as much information from data as possible. We will generate additional data using the following data augmentation techniques. Augmentation is a technique of manipulating the incoming training data to generate more instances of training data. This technique has been used to develop powerful classifiers with little data. However, augmentation is very specific to the objective of the neural network.

Brightness augmentation

Changing brightness to simulate day and night conditions. We will generate images with different brightness by first converting images to HSV, scaling up or down the V channel and converting back to the RGB channel

You can observe the images below for brightness augmentation.

Above Image is after applying random brightness

Above Image is after applying random brightness

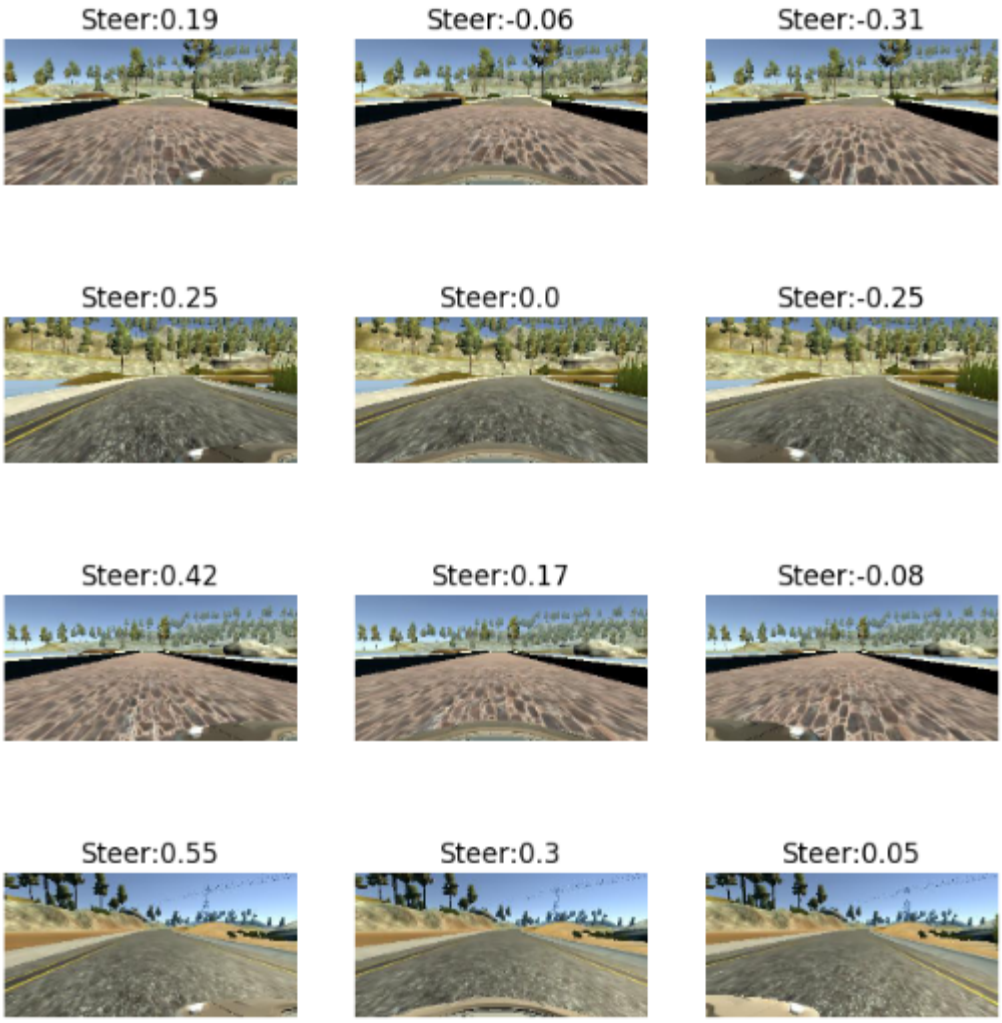
Above Image is after applying random brightness

Above Image is after applying random brightness

Using left and right camera images

Using left and right camera images to simulate the effect of car wandering off to the side, and recovering. We will add a small angle .25 to the left camera and subtract a small angle of 0.25 from the right

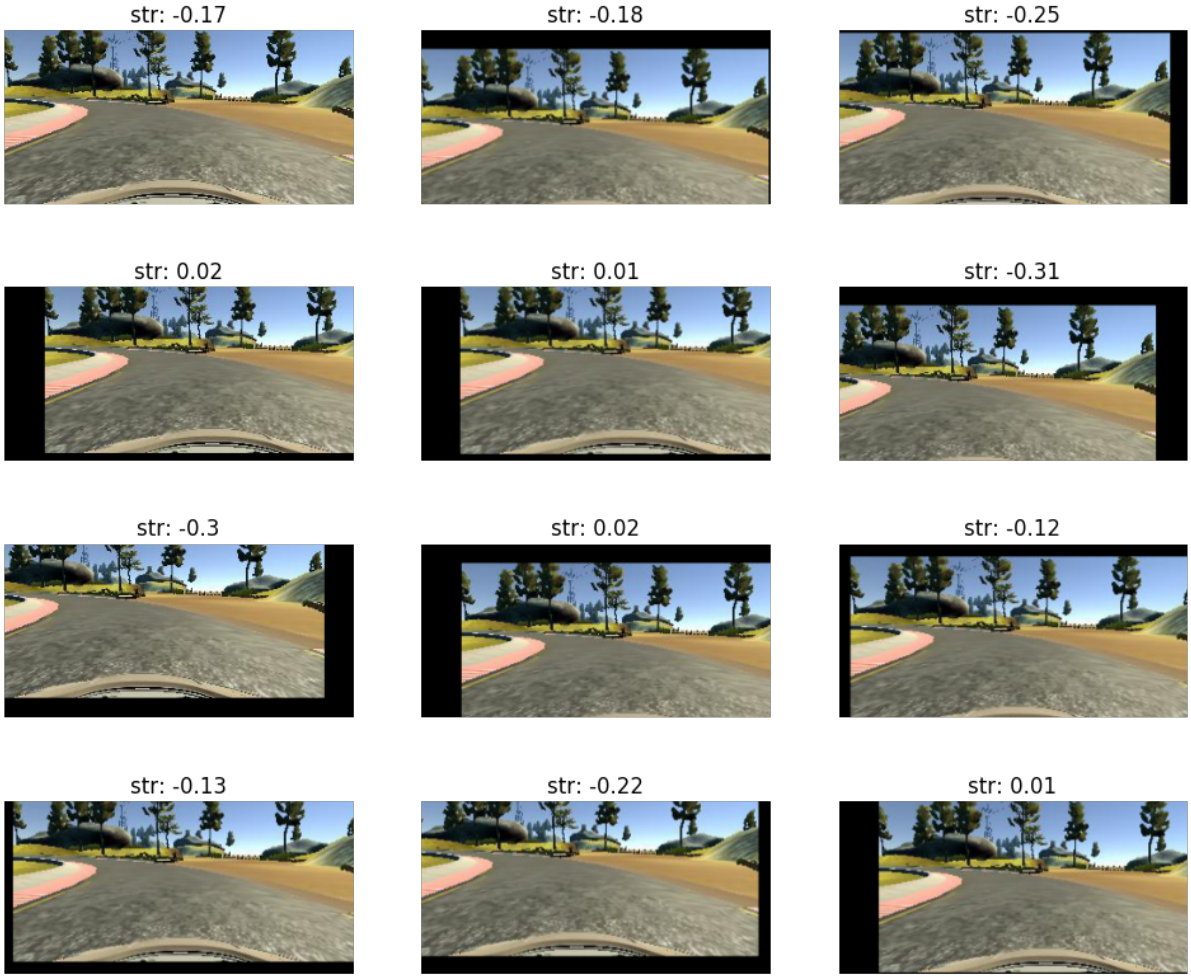
camera. The main idea being the left camera has to move right to get to center, and right camera has to move left. See the images below for better understanding.



Horizontal and vertical shifts

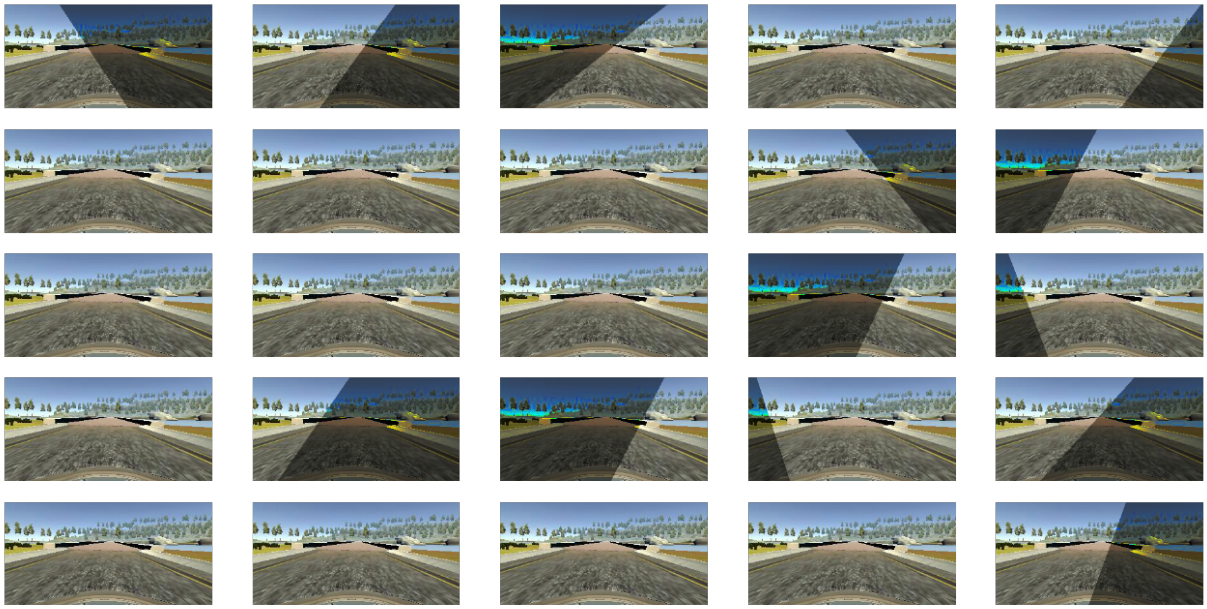
We will shift the camera images horizontally to simulate the effect of car being at different positions on the road, and add an offset corresponding to the shift to the steering angle. We added 0.004 steering angle units per pixel shift to the right, and subtracted 0.004 steering angle units per pixel shift to the left. We will also shift the

images vertically by a random number to simulate the effect of driving up or down the slop. See the images below for better understanding.



Shadow augmentation

The next augmentation we will add is shadow augmentation where random shadows are cast across the image. This is implemented by choosing random points and shading all points on one side (chosen randomly) of the image. The results for this augmentation is presented below.

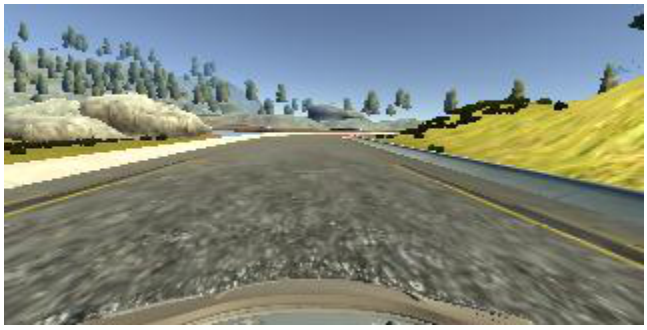


Flipping

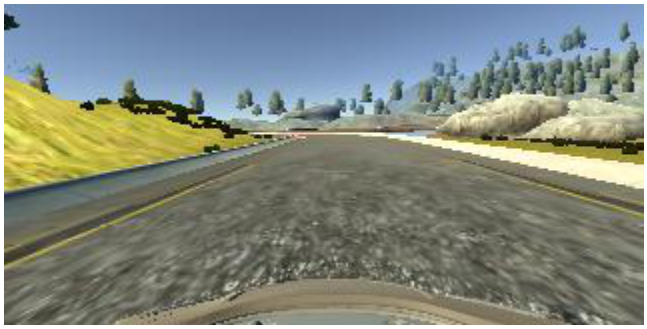
In addition to the transformations above, we will also flip images at random and change the sign of the predicted angle to simulate driving in the opposite direction.

We can understand the flipping operation by observing the original image and flipped image.

Original image



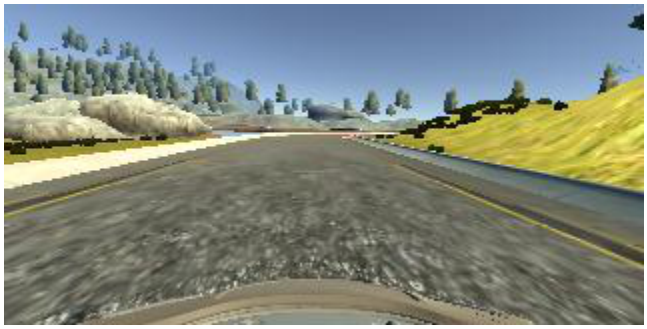
Flipped image



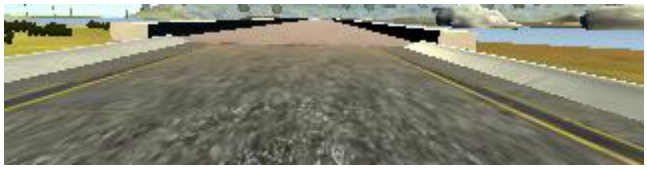
Preprocessing

After augmenting the image as above, we will crop the top 1/5 of the image to remove the horizon and the bottom 25 pixels to remove the car's hood. Originally 1/3 of the top of car image was removed, but later it was changed to 1/5 to include images for cases when the car may be driving up or down a slope. We will next rescale the image to a 64X64 square image. After augmentation, the augmented images looks as follows. These images are generated using keras generator, and unlimited number of images can be generated from one image. I used Lambda layer in keras to normalize intensities between -.5 and .5.

Orginal image

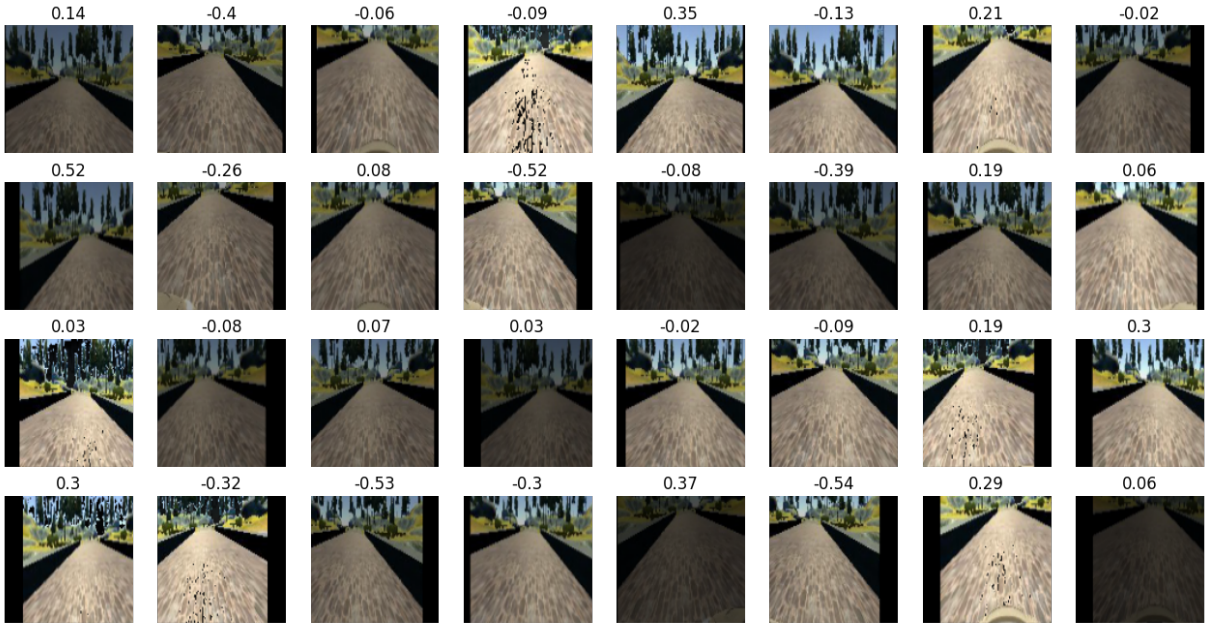


Cropped image



Keras generator for subsampling

As there was limited data and we are generating thousands of training examples from the same image, it is not possible to store all the images apriori into memory. We will utilize keras generator function to sample images such that images with lower angles have lower probability of getting represented in the data set. This alleviates any problems we may ecounter due to model having a bias towards driving straight. Panel below shows multiple training samples generated from one image.



The Model

The model architecture has been inspired from this post[11] and from End to End Deep Learning for Self Driving Car paper by NVIDIA.

There are two interesting bits about this architecture:

1.Its a simpleified version of the architcture mentioned in the paper published by NVIDIA.

2.The 1x1 convolutions at the beginning of this architecture.

To quote the author of this post[12],

> *"The first layer is 3 1X1 filters, this has the effect of transforming the color space of the images. Research has shown that different color spaces are better suited for different applications. As we do not know the best color space apriori, using 3 1X1 filters allows the model to choose its best color space"*

The 1x1 convolutions are followed by 3 sets of 3x3 convolution layers, of filter sizes 16, 32 and 64. Each set is accompanied with an max pooling layer and a drop out layer. These sets are followed by 3 fully connected layers. ELU has been chosen as the activation function.

---

11 Architecture post - https://github.com/vxy10/P3-BehaviorCloning

12 Author Architecture post - https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9

Model parameters:

1.Training/Test split = 0.8

2. Batch Size = 32

3. Number of epochs = 10

4. Learning rate = 1.0e-4

5. Samples per epoch = 20000

6. Optimizer : Adam

And now it's time to train.

Training

Having divided the dataset into train and test sets, generator is used to generate data on the fly for each epoch of training. The generator creates a batch of 32, where for each batch:

An image is chosen from amongst the left, center and right camera images.

The steering correction is set to be 0.25.

The image is then augmented by using aforementioned techniques.

The batches are then fed to the model.

Test

Using the Udacity provided simulator and drive.py file[13], the car can be driven autonomously around the track by executing

From the results , we observed that the steering angles are somewhat correct, but not completely accurate. Hence, the car turns, but not quite. And on sharp turns, the *not quite* factor plays a big role in taking the car out of the track. We can improve that in 2 ways:

> Retrain the model by tweaking parts of training configuations and settings such as data, model architecture etc.
>
> Put a leash on the car and control it using that leash.

Resolve the Data distribution

A quick look into the data reveals that there are 6148 number of rows in our data that have steering values less than 0.01. Which justifies our model's tendency to move straight and not being able to make sharp turns. If we removed around 70% of those, we would be left with 3732 data points in total, which would be **comparatively** balanced.

Since we know that when the car turns, the steering angles do not go up to the required values, lets inflate through steering values received from the model and give our car a help around the turns. This would require modifying the drive.py file.

13 drive.py file - https://d17h27t6h515a5.cloudfront.net/topher/2016/December/585b1df0_drive/drive.py

```
if np.absolute(steering_angle)>0.07:
    print('super inflating angles')
    steering_angle*=2.5
    throttle = 0.001
elif np.absolute(steering_angle)>0.04:
    print('inflating angles')
    steering_angle*=1.5
    throttle = 0.1
```

Now Test Again

It works. we observed , after removing the data points for straight path navigation, there's a swivel in the movement of the car, which means, the learning could be improved a bit here. Also, at the sharp turns, it no longer turns at the last moment, relying on the control code to get it through, signifying improved learning for turns.

CODE IMPLEMENTATION

```
import pandas as pf
from keras.optimizers import Adam
from keras.callbacks import TensorBoard
from keras.callbacks import ModelCheckpoint, Callback, EarlyStopping
from keras.layers.convolutional import Conv2D, Cropping2D
from keras.layers.pooling import MaxPooling2D
from keras.layers import Flatten, Dense, Lambda, ELU, Dropout
from keras.layers.advanced_activations import LeakyReLU, PReLU
from keras.models import Sequential
import numpy as np
from keras.preprocessing.image import img_to_array, load_img
from keras.layers.normalization import BatchNormalization
import cv2
DATA_DIRECTORY = 'data/'
TRAINING_SPLIT = 0.8
BATCH_SIZE = 32
LEARNING_RATE = 1.0e-4
EPOCHS = 15
MODEL_NAME =
        'model_hsv_70_elu_data_proper_enhancement_extreme_bn.h5'
MODEL_ROW_SIZE = 64
MODEL_COL_SIZE = 64
NO_OF_CHANNELS = 3
MODEL_INPUT_SHAPE =
        (MODEL_ROW_SIZE,
        MODEL_COL_SIZE,
NO_OF_CHANNELS)
TARGET_SIZE = (MODEL_ROW_SIZE, MODEL_COL_SIZE)
STEERING_CORRECTION = 0.25
STEERING_THRESHOLD = 0.15
STEERING_KEEP_PROBABILITY_THRESHOLD = 1
IMAGE_WIDTH = 320
IMAGE_HEIGHT = 160
#ACTIVATION = 'LeakyReLU'
ACTIVATION = 'elu'
```

```
def convert_to_YUV(image):
"converts the image from RGB space to YUV space"'
image = cv2.cvtColor(image, cv2.COLOR_RGB2YUV)
return image
def convert_to_HSV(image):
"'converts the image from RGB space to YUV space"'
image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
return image
def resize_image(image):
"'resize the image to 64 by 64 size"
'return cv2.resize(image, TARGET_SIZE)

def crop_image(image):

'''
:param image: The input image of dimensions 160x320x3
:crops out the sky and bumper portion of the car form the image
'''
cropped_image = image[55:135, :, :]
return cropped_image
def translate_image(image, steering_angle, range_x=100, range_y=10):
"""
Randomly shift the image virtially and horizontally (translation).
"""
trans_x = range_x * (np.random.rand() - 0.5)
trans_y = range_y * (np.random.rand() - 0.5)
steering_angle += trans_x * 0.002
trans_m = np.float32([[1, 0, trans_x], [0, 1, trans_y]])
height, width = image.shape[:2]
image = cv2.warpAffine(image, trans_m, (width, height))
return image, steering_angle
'''
def shift_horizon(img):
h, w, _ = img.shape
horizon = 2 * h / 5
v_shift = np.random.randint(-h/8,h/8)
pts1 = np.float32([[0,horizon],[w,horizon],[0,h],[w,h]])pts2 =
np.float32([[0,horizon+v_shift],[w,horizon+v_shift],[0,h],[w,h]])
M = cv2.getPerspectiveTransform(pts1,pts2)
```

```
return cv2.warpPerspective(img,M,(w,h),
borderMode=cv2.BORDER_REPLICATE)'''
def preprocess_image(image):
'''does the crop, resize and conversion to YUV space of the image'''
image = crop_image(image)
image = resize_image(image)
#image = convert_to_YUV(image)
image = convert_to_HSV(image)
image = np.array(image)
return image
def choose_image(row_data):
'''Selects an image from amongst the center camera image, left camra image
and the rigt camera image '''
toss = np.random.randint(3)
img_path = ''
steering = 0.0
if toss == 0:
#choose center image
img_path = row_data.iloc[0]
steering = row_data.iloc[3]
elif toss==1:
#choose left image
img_path = row_data.iloc[1]
steering = row_data.iloc[3] + STEERING_CORRECTION
elif toss==2:
#choose right image
img_path = row_data.iloc[2]
steering = row_data.iloc[3] - STEERING_CORRECTION
return img_path, steering
def load_image(img_path):
'''loads an image by reading from a file path'''
img_path = img_path.strip()
#print('image path-->', DATA_DIRECTORY + img_path)
image = cv2.imread(DATA_DIRECTORY + img_path)
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
return image
def bright_augment_image(img):
'''Adds rando mbrightness to the image'''
img1 = cv2.cvtColor(img,cv2.COLOR_RGB2HSV)
random_bright = .25 + np.random.uniform()
```

```
img1[:,:,2] = img1[:,:,2]*random_brightimg1 =
cv2.cvtColor(img1,cv2.COLOR_HSV2RGB)
return img1
def random_shadow(image):
"""
Generates and adds random shadow
"""
# (x1, y1) and (x2, y2) forms a line
# xm, ym gives all the locations of the image
x1, y1 = IMAGE_WIDTH * np.random.rand(), 0
x2, y2 = IMAGE_WIDTH * np.random.rand(), IMAGE_HEIGHT
xm, ym = np.mgrid[0:IMAGE_HEIGHT, 0:IMAGE_WIDTH]
# mathematically speaking, we want to set 1 below the line and zero otherwise
# Our coordinate is up side down. So, the above the line:
# (ym-y1)/(xm-x1) > (y2-y1)/(x2-x1)
# as x2 == x1 causes zero-division problem, we'll write it in the below form:
# (ym-y1)*(x2-x1) - (y2-y1)*(xm-x1) > 0
mask = np.zeros_like(image[:, :, 1])
mask[(ym - y1) * (x2 - x1) - (y2 - y1) * (xm - x1) > 0] = 1
# choose which side should have shadow and adjust saturation
cond = mask == np.random.randint(2)
s_ratio = np.random.uniform(low=0.2, high=0.5)
# adjust Saturation in HLS(Hue, Light, Saturation)
hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
hls[:, :, 1][cond] = hls[:, :, 1][cond] * s_ratio
return cv2.cvtColor(hls, cv2.COLOR_HLS2RGB)
return img1
def random_shadow(image):
"""
Generates and adds random shadow
"""
# (x1, y1) and (x2, y2) forms a line
# xm, ym gives all the locations of the image
x1, y1 = IMAGE_WIDTH * np.random.rand(), 0
x2, y2 = IMAGE_WIDTH * np.random.rand(), IMAGE_HEIGHT
xm, ym = np.mgrid[0:IMAGE_HEIGHT, 0:IMAGE_WIDTH]
# mathematically speaking, we want to set 1 below the line and zero otherwise
# Our coordinate is up side down. So, the above the line:
# (ym-y1)/(xm-x1) > (y2-y1)/(x2-x1)
# as x2 == x1 causes zero-division problem, we'll write it in the below form:
```

```
# (ym-y1)*(x2-x1) - (y2-y1)*(xm-x1) > 0
mask = np.zeros_like(image[:, :, 1])
mask[(ym - y1) * (x2 - x1) - (y2 - y1) * (xm - x1) > 0] = 1
# choose which side should have shadow and adjust saturation
cond = mask == np.random.randint(2)
s_ratio = np.random.uniform(low=0.2, high=0.5)
# adjust Saturation in HLS(Hue, Light, Saturation)
hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
hls[:, :, 1][cond] = hls[:, :, 1][cond] * s_ratio
return cv2.cvtColor(hls, cv2.COLOR_HLS2RGB)
def augment_image(row_data):
'''auments the input image by translating, adding random shadow, augmenting
brightness and preprocessing (crop , resize and conversion to YUV space)
the image'''
img_path, steering = choose_image(row_data)
image = load_image(img_path)
#translate image
image, steering = translate_image(image, steering)
#augment brightness
image = bright_augment_image(image)
#image = shift_horizon(image)
#flip image
# This is done to reduce the bias for turning left that is present in the training data
flip_prob = np.random.random()
if flip_prob > 0.5:
# flip the image and reverse the steering angle
steering = -1*steering
image = cv2.flip(image, 1)
#random shadow
image = random_shadow(image)
#preprocess image
image = preprocess_image(image)
return image, steering
def get_data_generator(data_frame):
'''generator to generate data for the model on the fly.'''
batch_size = BATCH_SIZE
print('data generator called')
N = data_frame.shape[0]
print('N -->', N)
batches_per_epoch = N // batch_size
```

```
print('batches per epoch-->', batches_per_epoch)
i = 0
while(True):
start = i*batch_size
end = start+batch_size - 1
X_batch = np.zeros((batch_size, 64, 64, 3), dtype=np.float32)
y_batch = np.zeros((batch_size,), dtype=np.float32)
j = 0
# slice a `batch_size` sized chunk from the dataframe
# and generate augmented data for each row in the chunk on the fly
for index, row in data_frame.loc[start:end].iterrows():
image, steering = augment_image(row)
X_batch[j] = image
y_batch[j] = steering
j += 1
i += 1
if i == batches_per_epoch - 1:
# reset the index so that we can cycle over the data_frame again
i = 0
yield X_batch, y_batch
def main():
'''main function from where the code flow starts'''
print('Behavioral Cloning')
org_data_frame = load_data()
print('Data loaded')
“get_data_stats(org_data_frame)train_data, valid_data, new_data_frame =
filter_dataset(org_data_frame)”
print('Data filtered')
#get_data_stats(new_data_frame)
org_data_frame = None
print('calling generators')
training_generator = get_data_generator(train_data)
validation_data_generator = get_data_generator(valid_data)
model = get_model()
adam = Adam(lr=LEARNING_RATE, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
decay=0.0)
model.compile(optimizer = 'adam', loss = 'mse')
early_stopping = EarlyStopping(monitor='val_loss', patience=8, verbose=1,
mode='auto')
```

```
save_weights = ModelCheckpoint('model.h5', monitor='val_loss',
save_best_only=True)
tensorboard = TensorBoard(log_dir='lossandvalloss/', histogram_freq=0,
                  write_graph=True, write_images=False)
model.fit_generator(training_generator,
validation_data=validation_data_generator, nb_epoch =
EPOCHS,
callbacks=[tensorboard], samples_per_epoch = 20000, nb_val_samples =
len(valid_data))
“model.fit(np.array([[1]]), np.array([[2]]), batch_size=BATCH_SIZE, shuffle =
True, nb_epoch =
EPOCHS, callbacks=[save_weights, early_stopping])”
#save the model
model.save(MODEL_NAME)
print("Model Saved!")
def get_model():
model = Sequential()
#model.add(Lambda(lambda x: x / 255., input_shape=MODEL_INPUT_SHAPE,
name='normalizer'))
'"model.add(Cropping2D(cropping=crop,
input_shape=(row, col, ch), name='cropping'))"'
model.add(Conv2D(3, 1, 1, subsample=(2, 2),
input_shape=MODEL_INPUT_SHAPE,activation=ACTIVATION, name='cv0'))
model.add(BatchNormalization())
model.add(Conv2D(16, 3, 3, activation=ACTIVATION, name='cv1'))
model.add(BatchNormalization())
model.add(MaxPooling2D(name='maxPool_cv1'))
model.add(Dropout(0.5, name='dropout_cv1'))
model.add(Conv2D(32, 3, 3, activation=ACTIVATION, name='cv2'))
model.add(BatchNormalization())
model.add(MaxPooling2D(name='maxPool_cv2'))
model.add(Dropout(0.5, name='dropout_cv2'))
model.add(Conv2D(64, 3, 3, activation=ACTIVATION, name='cv3'))
model.add(BatchNormalization())model.add(MaxPooling2D(name='maxPool_cv3'
))
model.add(Dropout(0.5, name='dropout_cv3'))
model.add(Flatten())
model.add(Dense(1000, activation=ACTIVATION, name='fc1'))
model.add(BatchNormalization())
model.add(Dropout(0.5, name='dropout_fc1'))
```

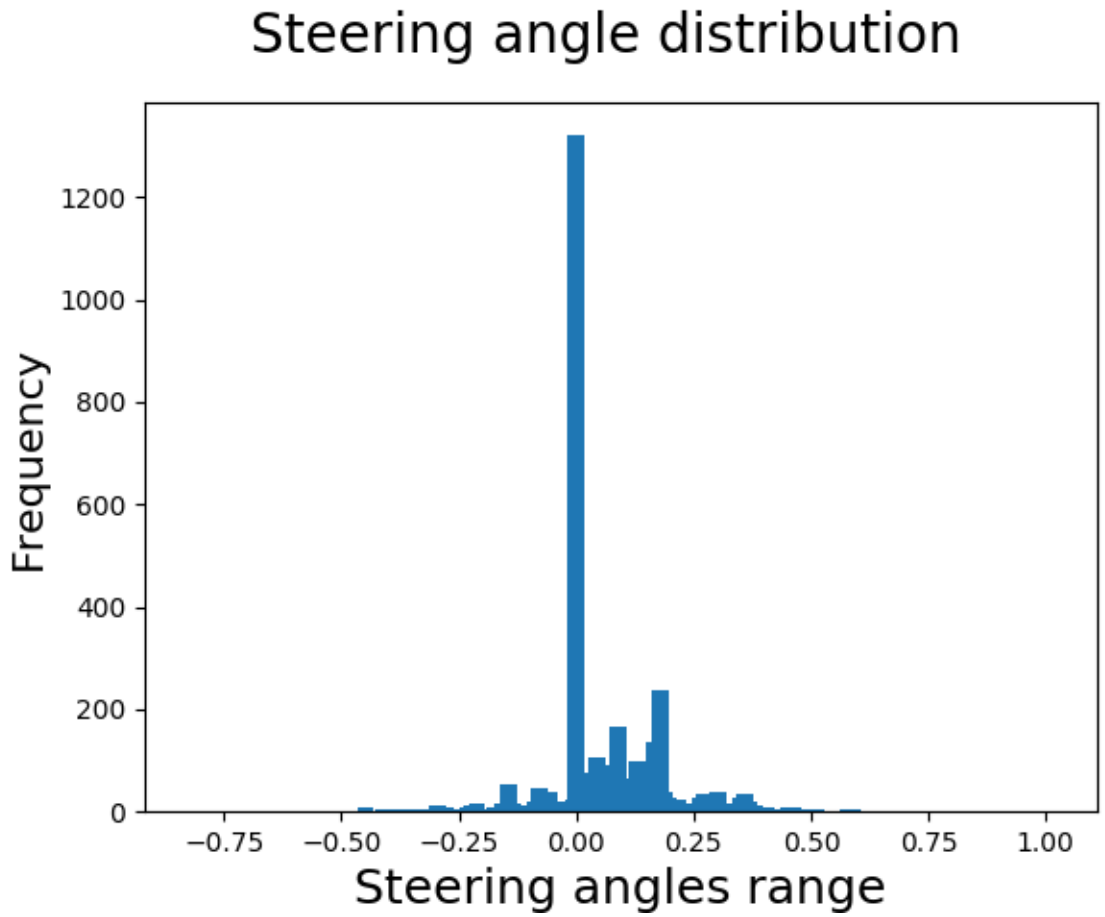
```
model.add(Dense(100, activation=ACTIVATION, name='fc2'))
model.add(Dense(100, activation=ACTIVATION, name='fc2'))
model.add(BatchNormalization())
model.add(Dropout(0.5, name='dropout_fc2'))
model.add(Dense(10, activation=ACTIVATION, name='fc3'))
model.add(BatchNormalization())
model.add(Dropout(0.5, name='dropout_fc3'))
model.add(Dense(1, name='output'))
return model
#even out the steering angle values in data
#filter out the 0 and lower speed values from data
def filter_dataset(data_frame):
data_frame = filter_steering(data_frame)
# replicate rows having extreme values of steering angles
#data_frame = replicate_steering(data_frame)
data_frame = replicate_proper_steering(data_frame)
num_rows_training = int(data_frame.shape[0]*TRAINING_SPLIT)
training_data = data_frame.loc[0:num_rows_training-1]
validation_data = data_frame.loc[num_rows_training:]
return training_data, validation_data, data_frame
def filter_steering(data_frame):
'''evens out the steering angle distribution in the data set by removing
70% of the rows with steering angle close to 0 (<0.01).'''
print('Steering filtering')
print ('number of rows with steering less than 0.01: ',
len(data_frame.loc[data_frame['steering']
<0.007]))
data_frame = data_frame.drop(data_frame[data_frame['steering']
<0.01].sample(frac=0.70).index)
print('new data frame length-->', len(data_frame))
return data_frame
def replicate_steering(data_frame):
print('Steering replicating')
new_data_frame = data_frame.loc[data_frame['steering'] >
0.20].append(data_frame.loc[data_frame['steering'] < - 0.20])
#print('new data frame-->', new_data_frame)
num_of_angles = len(data_frame.loc[data_frame['steering'] >
0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]))
data_frame = data_frame.append([new_data_frame]*5,ignore_index=True)
#print()
```

```
#df_new = data_frame.loc[np.repeat(data_frame.loc[data_frame['steering'] >
#0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]),
[5]*num_of_angles)]
return data_frame
def replicate_proper_steering(data_frame):
print('Steering proper replicating')
data_frame_0_10 = data_frame.loc[(data_frame['steering'] >= 0.008) &
(data_frame['steering'] <
0.10)]
data_frame_10_20 = data_frame.loc[(data_frame['steering'] >= 0.10) &
(data_frame['steering'] <
0.20)]
data_frame_20_30 = data_frame.loc[(data_frame['steering'] >= 0.20) &
(data_frame['steering'] <
0.30)]
data_frame_30_40 = data_frame.loc[(data_frame['steering'] >= 0.30) &
(data_frame['steering'] <
0.40)]
data_frame_40_50 = data_frame.loc[(data_frame['steering'] >= 0.40) &
(data_frame['steering'] <
0.50)]
data_frame_50 = data_frame.loc[data_frame['steering'] >= 0.50]
data_frame = data_frame.append([data_frame_0_10]*3,ignore_index=True)
data_frame = data_frame.append([data_frame_10_20]*2,ignore_index=True)
data_frame = data_frame.append([data_frame_20_30]*8,ignore_index=True)
data_frame = data_frame.append([data_frame_30_40]*6,ignore_index=True)
data_frame = data_frame.append([data_frame_40_50]*8,ignore_index=True)
data_frame = data_frame.append([data_frame_50]*60,ignore_index=True)
data_frame_neg_1_10 = data_frame.loc[(data_frame['steering'] <= -0.008) &
(data_frame['steering'] > -0.10)]
data_frame_neg_10_20 = data_frame.loc[(data_frame['steering'] <= -0.10) &
(data_frame['steering'] > -0.20)]
data_frame_neg_20_30 = data_frame.loc[(data_frame['steering'] <= -0.20) &
(data_frame['steering'] > -0.30)]
data_frame_neg_30_40 = data_frame.loc[(data_frame['steering'] <= -0.30) &
(data_frame['steering'] > -0.40)]
data_frame_neg_40_50 = data_frame.loc[(data_frame['steering'] <= -0.40) &
(data_frame['steering'] > -0.50)]
data_frame_neg_50 = data_frame.loc[data_frame['steering'] <= -0.50]
data_frame = data_frame.append([data_frame_neg_1_10]*8,ignore_index=True)
```

```
data_frame = data_frame.append([data_frame_neg_10_20]*8,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_20_30]*20,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_30_40]*20,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_40_50]*20,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_50]*80,ignore_index=True)
#print('new data frame-->', new_data_frame)
#num_of_angles = len(data_frame.loc[data_frame['steering'] >
#0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]))
#print()
#df_new = data_frame.loc[np.repeat(data_frame.loc[data_frame['steering'] >
#0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]),
[5]*num_of_angles)]
return data_frame
def filter_throttle():
'''filters out the throttle values less than 0.25'''
print('Throttle filtering')
#print (len(org_data_frame.loc[org_data_frame['throttle'] >=0.5]))
def load_data():
'''loads the data from the driving log .csv file into a dataframe'''
data_frame = pd.read_csv(DATA_DIRECTORY +'driving_log.csv')
# shuffle the data
data_frame = data_frame.sample(frac=1).reset_index(drop=True)
# 80-20 training validation split
'''num_rows_training = int(data_frame.shape[0]*TRAINING_SPLIT)
training_data = data_frame.loc[0:num_rows_training-1]
validation_data = data_frame.loc[num_rows_training:]'''
return data_frame
if __name__ == '__main__':
main()
```

Outputs and Results

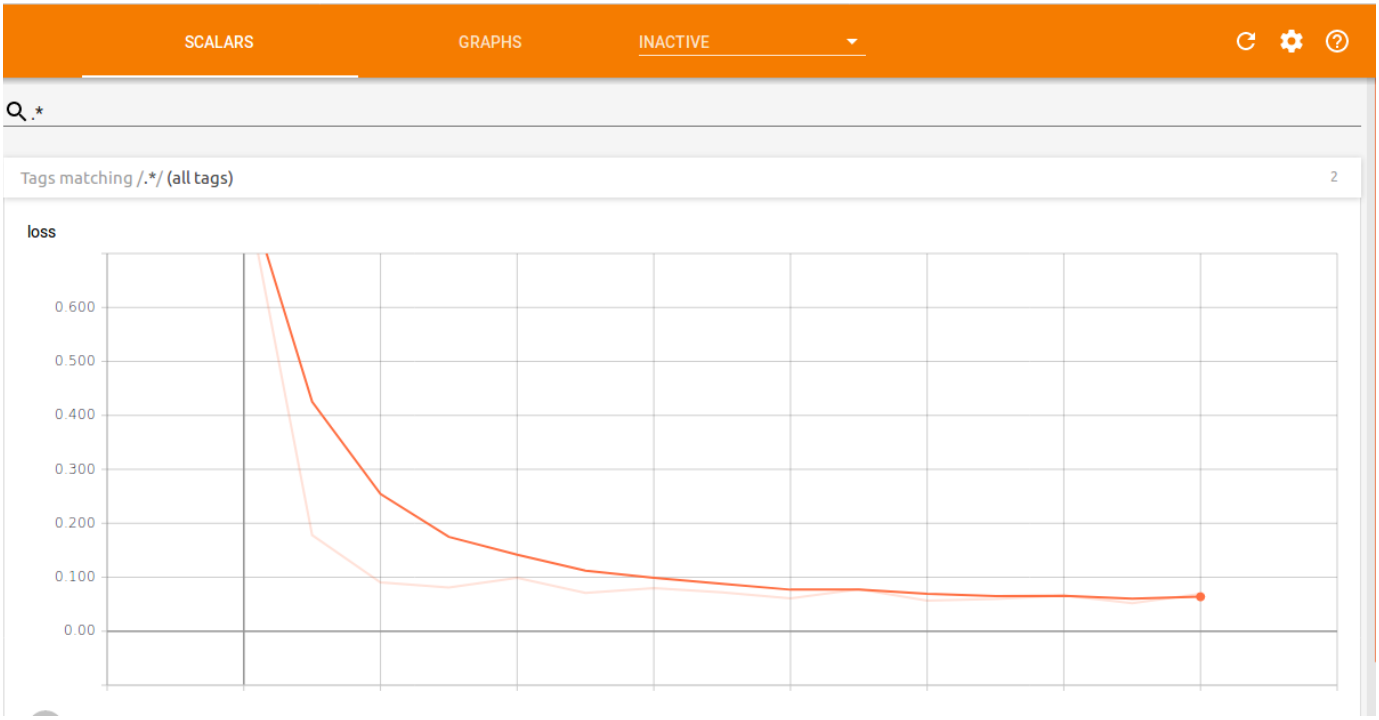
we can see that steering angle distribution is balanced



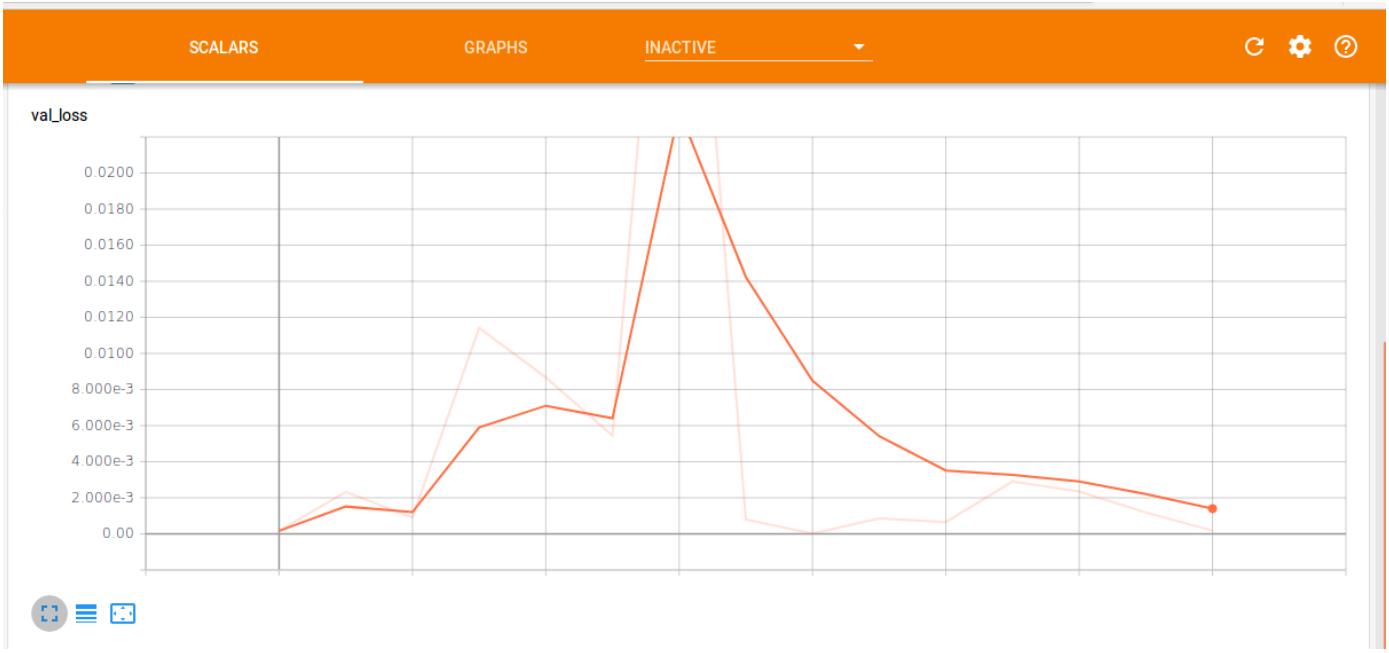
From the screen shot below, We can observe the loss and val_loss of our model per each epoch while training the model.

```
nikhil@CVC147:~$ cd autonomouscar/
nikhil@CVC147:~/autonomouscar$ python3 model.py
Using TensorFlow backend.
/home/nikhil/.local/lib/python3.5/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of th
ed as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
 Behavioral Cloning
Data loaded
Steering filtering
number of rows with steering less than 0.01:  6148
new data frame length--> 3732
Steering proper replicating
Data filtered
calling generators
2018-06-13 15:04:24.478317: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports i
2018-06-13 15:04:24.525261: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:892] successful N
 NUMA node zero
2018-06-13 15:04:24.525676: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Found device 0
name: GeForce GTX 770 major: 3 minor: 0 memoryClockRate(GHz): 1.163
pciBusID: 0000:08:00.0
totalMemory: 1.95GiB freeMemory: 1.65GiB
2018-06-13 15:04:24.525710: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Creating Tensor
bility: 3.0)
data generator called
N --> 13796
batches per epoch--> 431
Epoch 1/15
19968/20000 [============================>.] - ETA: 0s - loss: 0.7635data generator called
N --> 3450
batches per epoch--> 107
20000/20000 [==============================] - 108s - loss: 0.7627 - val_loss: 0.0472
Epoch 2/15
20000/20000 [==============================] - 108s - loss: 0.1773 - val_loss: 0.0011
Epoch 3/15
20000/20000 [==============================] - 108s - loss: 0.0866 - val_loss: 0.0020
Epoch 4/15
20000/20000 [==============================] - 109s - loss: 0.0980 - val_loss: 4.7337e-04
Epoch 5/15
20000/20000 [==============================] - 108s - loss: 0.0723 - val_loss: 0.0019
Epoch 6/15
20000/20000 [==============================] - 108s - loss: 0.0711 - val_loss: 4.4802e-04
Epoch 7/15
20000/20000 [==============================] - 108s - loss: 0.0793 - val_loss: 0.0012
Epoch 8/15
20000/20000 [==============================] - 108s - loss: 0.0615 - val_loss: 0.0012
Epoch 9/15
20000/20000 [==============================] - 107s - loss: 0.0760 - val_loss: 0.0020
Epoch 10/15
20000/20000 [==============================] - 107s - loss: 0.0560 - val_loss: 0.0031
Epoch 11/15
20000/20000 [==============================] - 107s - loss: 0.0744 - val_loss: 0.0020
Epoch 12/15
20000/20000 [==============================] - 106s - loss: 0.0537 - val_loss: 5.4811e-04
Epoch 13/15
20000/20000 [==============================] - 108s - loss: 0.0694 - val_loss: 0.0022
Epoch 14/15
20000/20000 [==============================] - 109s - loss: 0.0512 - val_loss: 0.0021
Epoch 15/15
20000/20000 [==============================] - 108s - loss: 0.0610 - val_loss: 0.0711
Model Saved!
nikhil@CVC147:~/autonomouscar$ python3 model.py
```

Loss



Validation_Loss



from the screen shot below, you can see that the car driving autonomously in the simulator.

for complete video of the car running autonomously in the simulator, see this video[33].

Conclusion

The car driven by the model was able to correctly traverse the track. The car exhibited minimum over-correcting on the straight – aways, and it had a couple close calls with the edge of the road and the bridge. Overall, it performed well on the track.

Thoughts

Having seen in this project, data is everything. This solution works, but it could be improved ina number of ways.

1. Different Model Architecture

Altought the dataset is not that large or complex, still it could be insightful how some of the other arhitectures would perform for this project.

2. Data collection

We could actually try collecting data **ourselves**, the way it is suggested with recovery and regular laps. The point is, we could see what effect does more data have on this implementation and how that would change things.

3. Driving like a human

Right now, the car is able to drive around both tracks, but let’s face it, if it were a real world driving, it would end up getting ticktets even before reaching the first turn. We could try teaching the model lane driving, which would mean some added data augmentation techiniques and better data to work with.

4. Break and Throttle

Currently, our convolutional neural network only predicts the values of steering angles using images. It could be enhanced to predict the throttle and break values as well.

References

[1]. Udacity Self Driving Car Simulator - https://github.com/udacity/self-driving-car-sim

[2]. End to End Learning for Self Driving Cars Paper – https://arxiv.org/pdf/1604.07316v1.pdf

[3]. NVIDIA Website - http://www.nvidia.com/content/global/global.php

[4]. Neural Networks Basics - http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

[5]. Python Website - https://www.python.org/

[6]. ImageNet Classification with Deep Convolutional Neural Networks - http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[7]. L. D. Jackel, D. Sharman, Stenard C. E., Strom B. I., , and D Zuckert. Optical character recognition for self-service banking. AT&T Technical Journal, 74(1):16–24, 1995

[8]. IMAGENET Large Scale Visual Recognition Challenge (ILSVRC) - http://www.image-net.org/challenges/LSVRC/

[9]. DAVE Final Technical Report – http://net-scale.com/doc/net-scale-dave-report.pdf

[10]. Autonomous Land Vehicle In a Neural Network paper by Dean A. Pamerleau, January 1989 - http://repository.cmu.edu/cgi/viewcontent.cgi?article=2874&context=compsci

[11]. DAPRA LAGR Program - https://en.wikipedia.org/wiki/DARPA_LAGR_Program

[12]. Danwei Wang and Feng Qi. Trajectory planning for a four-wheel-steering vehicle. In Proceedings of the 2001 IEEE International Conference on Robotics & Automation, May 21–26 2001. URL: http://www.ntu.edu.sg/home/edwwang/confpapers/wdwicar01.pdf.

[13]. DAVE -2, Neural Network drives a car video - https://www.youtube.com/watch?time_continue=2&v=NJU9ULQUwng

[14]. Python Documentation - https://docs.python.org/3/index.html

[15]. Pandas Documentation - https://pandas.pydata.org/pandas-docs/stable/

[16]. Numpy Documentation - https://docs.scipy.org/doc/

[17]. Matplotlib Documentation - https://matplotlib.org/contents.html

[18]. Scikit-learn Documentation - http://scikit-learn.org/stable/documentation.html

[19]. OpenCV Installation Linux - https://www.learnopencv.com/install-opencv3-on-ubuntu/

[20]. OpenCV Documentation - https://docs.opencv.org/2.4/index.html

[21]. Pillow Documentation - https://pillow.readthedocs.io/en/5.1.x/

[22]. Scikit-image Documentation - http://scikit-image.org/docs/stable/

[23]. Scipy Documentation - https://www.scipy.org/docs.html

[24]. h5py Documentation - http://docs.h5py.org/en/latest/

[25]. Eventlet Documentation - http://eventlet.net/doc/

[26]. Flask-Socketio Documentation - https://python-socketio.readthedocs.io/en/latest/

[27]. Seaborn Documentation - https://seaborn.pydata.org/

[28]. Imageio Documentation - https://imageio.readthedocs.io/en/stable/

[29]. Moviepy Documentation - https://zulko.github.io/moviepy/

[30]. Google AI Organisation was announced at Google I/O 2017 by CEO Sundar Pichai URL: https://ai.google/

[31]. Tensor Flow Documentation guide - https://www.tensorflow.org/programmers_guide/

[32]. Keras Documentation - https://keras.io/

[33]. Complete Result Video - https://drive.google.com/file/d/1oli3yQrNIskrSsjFzahlfXmLCShPTv_s/view?usp=sharing