SELF-DRIVING CAR USING UDACITY'S CAR SIMULATOR ENVIRONMENT AND TRAINED BY DEEP NEURAL NETWORKS COMPLETE GUIDE

Table of Contents

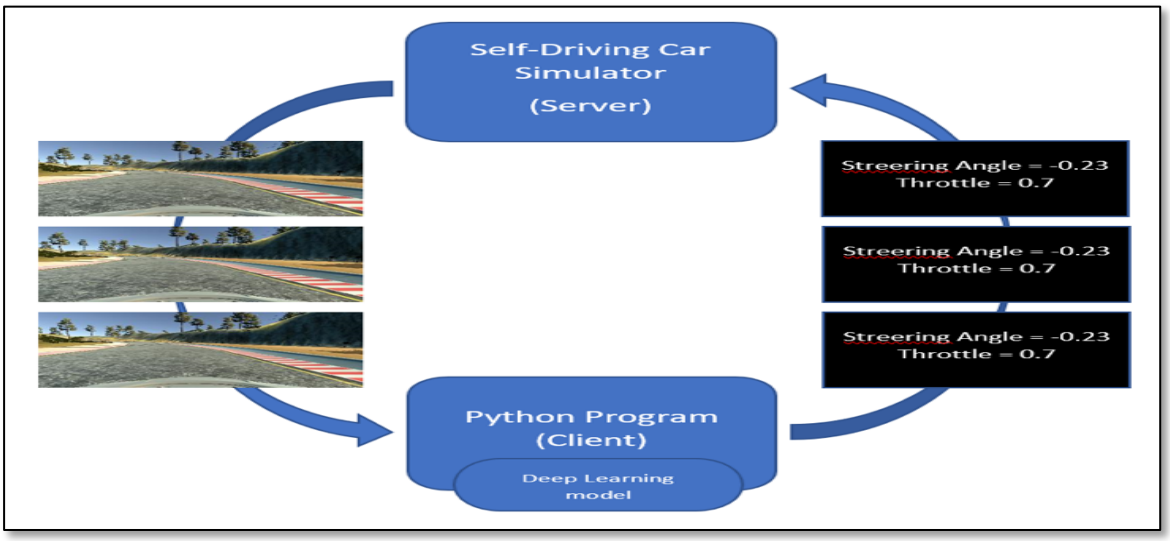
Introduction

Self-driving cars has become a trending subject with a significant improvement in the technologies in the last decade. The project purpose is to train a neural network to drive an autonomous car agent on the tracks of Udacity's Car Simulator environment. Udacity has released the simulator as an open source software and enthusiasts have hosted a competition (challenge) to teach a car how to drive using only camera images and deep learning. Driving a car in an autonomous manner requires learning to control steering angle, throttle and brakes. Behavioral cloning technique is used to mimic human driving behavior in the training mode on the track. That means a dataset is generated in the simulator by user driven car in training mode, and the deep neural network model then drives the car in autonomous mode. Ultimately, the car was able to run on Track 1 generalizing well. The project aims at reaching the same accuracy on real time data in the future.

Problem Definition

Udacity released an open source simulator for self-driving cars to depict a real-time environment. The challenge is to mimic the driving behavior of a human on the simulator with the help of a model trained by deep neural networks. The concept is called Behavioral Cloning, to mimic how a human drives. The simulator contains two tracks and two modes, namely, training mode and autonomous mode. The dataset is generated from the simulator by the user, driving the car in training mode. This dataset is also known as the "good" driving data. This is followed by testing on the track, seeing how the deep learning model performs after being trained by that user data.

Solution Approach



The problem is solved in the following steps:

- The simulator can be used to collect data by driving the car in the training mode using a joystick or keyboard, providing the so called “good-driving” behavior input data in form of a driving_log (.csv file) and a set of images. The simulator acts as a server and pipes these images and data log to the Python client.
- The client (Python program) is the machine learning model built using Deep Neural Networks. These models are developed on Keras (a high-level API over Tensorflow). Keras provides sequential models to build a linear stack of network layers. Such models are used in the project to train over the datasets as the second step. Detailed description of CNN models experimented and used can be referred to in the chapter on network architectures.
- Once the model is trained, it provides steering angles and throttle to drive in an autonomous mode to the server (simulator).
- These modules, or inputs, are piped back to the server and are used to drive the car autonomously in the simulator and keep it from falling off the track

Technologies Used

Technologies that are used in the implementation of this project and the motivation behind using these are described in this section.

TensorFlow: This an open-source library for dataflow programming. It is widely used for machine learning applications. It is also used as both a math library and for large computation. For this project Keras, a high-level API that uses TensorFlow as the backend is used. Keras facilitate in building the models easily as it more user friendly.

Different libraries are available in Python that helps in machine learning projects. Several of those libraries have improved the performance of this project. Few of them are mentioned in this section. First, “Numpy” that provides with high-level math function collection to support multi-dimensional metrices and arrays. This is used for faster computations over the weights (gradients) in neural networks. Second, “scikit-learn” is a machine learning library for Python which features different algorithms and Machine Learning function packages. Another one is OpenCV (Open Source Computer Vision Library) which is designed for computational efficiency with focus on real-time applications. In this project, OpenCV is used for image preprocessing and augmentation techniques.

The project makes use of Conda Environment which is an open source distribution for Python which simplifies package management and deployment. It is best for large scale data processing. The machine on which this project was built, is a personal computer.

Convolutional Neural Networks (CNN)

CNN is a type of feed-forward neural network computing system that can be used to learn from input data. Learning is accomplished by determining a set of weights or filter values that allow the network to model the behavior according to the training data. The desired output and the output generated by CNN initialized with random weights will be different. This difference (generated error) is backpropagated through the layers of CNN to adjust the weights of the neurons, which in turn reduces the error and allows us produce output closer to the desired one.

CNN is good at capturing hierarchical and spatial data from images. It utilizes filters that look at regions of an input image with a defined window size and map it to some output. It then slides the window by some defined stride to other regions, covering the whole image. Each convolution filter layer thus captures the properties of this input image hierarchically in a

series of subsequent layers, capturing the details like lines in image, then shapes, then whole objects in later layers. CNN can be a good fit to feed the images of a dataset and classify them into their respective classes.

Time-Distributed Layers

Another type of layers sometimes used in deep learning networks is a Time- distributed layer. Time-Distributed layers are provided in Keras as wrapper layers. Every temporal slice of an input is applied with this wrapper layer. The requirement for input is that to be at least three-dimensional, first index can be considered as temporal dimension. These Time-Distributed can be applied to a dense layer to each of the timesteps, independently or even used with Convolutional Layers. The way they can be written is also simple in Keras as shown in Figure 1 and Figure 2

```
# as the first layer in a model
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
# now model.output_shape == (None, 10, 8)
```

Fig. 1: TimeDistributed Dense layer

```
model = Sequential()
model.add(TimeDistributed(Conv2D(64, (3, 3)),
                          input_shape=(10, 299, 299, 3)))
```

Fig. 2: TimeDistributed Convolution layer

Udacity Simulator and Dataset

We will first download the simulator to start our behavioural training process.
Udacity has built a simulator for self-driving cars and made it open source for the enthusiasts, so they can work on something close to a real-time environment. It is built on Unity, the video game development platform. The simulator consists of a configurable resolution and controls setting and is very user friendly. The graphics and input configurations can be changed according to user preference and machine configuration as shown in Figure 3. The user pushes the “Play!” button to enter the simulator user interface. You can enter the Controls tab to explore the keyboard controls, quite similar to a racing game which can be seen in Figure 4.

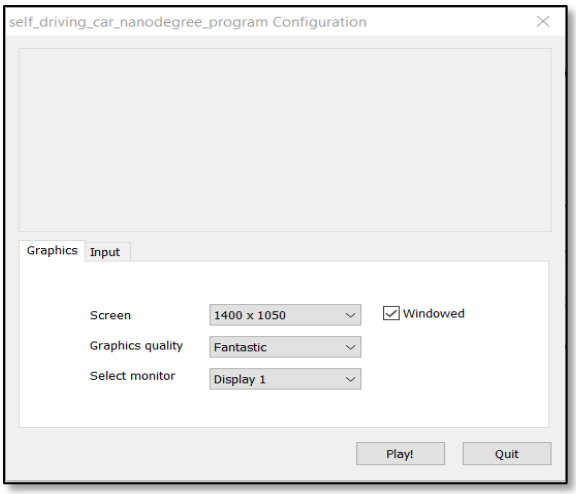


Fig. 3: Configuration screen

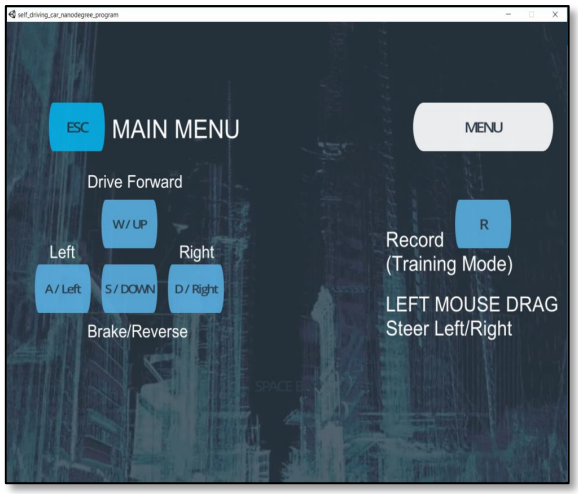


Fig. 4: Controls Configuration

The first actual screen of the simulator can be seen in Figure 5 and its components are discussed below. The simulator involves two tracks. One of them can be considered as simple and another one as complex that can be evident in the screenshots attached in Figure 6 and Figure 7. The word “simple” here just means that it has fewer curvy tracks and is easier to drive on, refer Figure 6. The “complex” track has steep elevations, sharp turns, shadowed environment, and is tough to drive on, even by a user doing it manually. Please refer Figure 6. There are two modes for driving the car in the simulator: (1) Training mode and (2) Autonomous mode. The training mode gives you the option of recording your run and capturing the training dataset. The small red sign at the top right of the screen in the Figure 6 and 7 depicts the car is being driven in training mode. The autonomous mode can be used to test the models to see if it can drive on the track without human intervention. Also, if you try to press the controls to get the car back on track, it will immediately notify you that it shifted to manual controls. The mode screenshot can be as seen in Figure 8. Once we have mastered how the car driven controls in simulator using keyboard keys, then we get started with record button to collect data. We will save the data from it in a specified folder as you can see below.

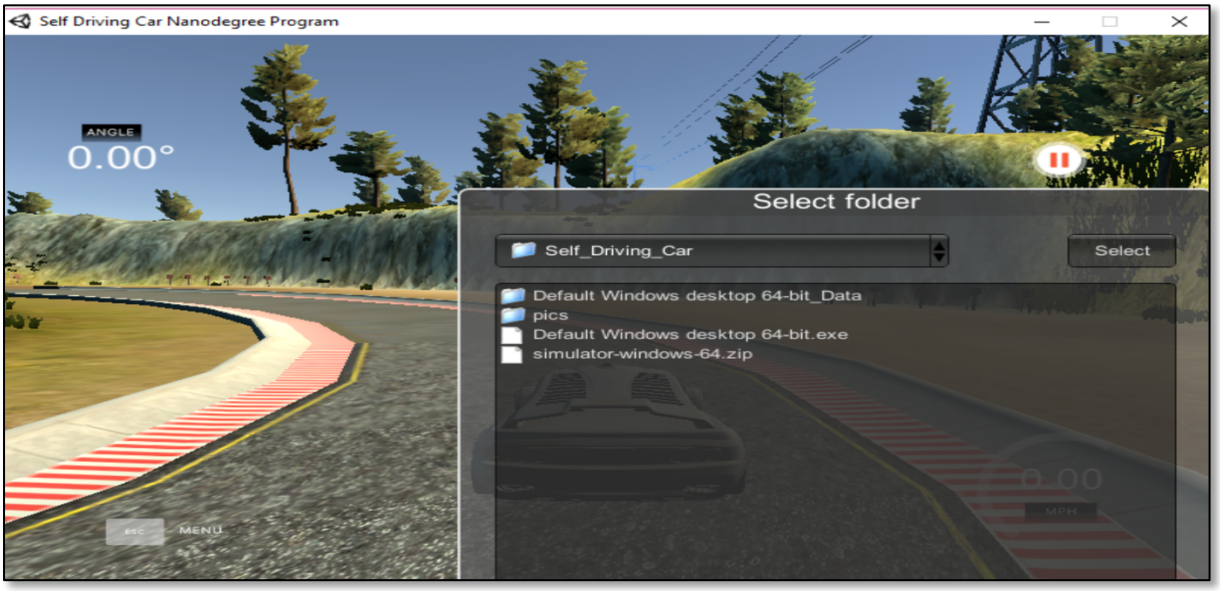


Fig. 5: First Screen

Fig 6. Track 1

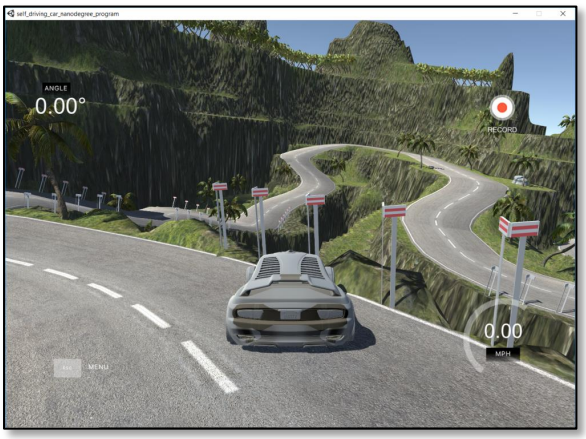


Fig 7. Track 2

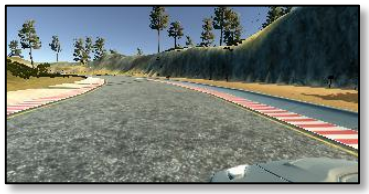
Fig 8 . Autonomous mode

The simulator's feature to create your own dataset of images makes it easy to work on the problem. Some reasons why this feature is useful are as follows:

- The simulator has built the driving features in such a way that it simulates that there are three cameras on the car. The three cameras are in the center, right and left on the front of the car, which captures continuously when we record in the training mode.
- The stream of images is captured, and we can set the location on the disk for saving the data after pushing the record button. The image set are labelled in a sophisticated manner with a prefix of center, left, or right indicating from which camera the image has been captured.
- Along with the image dataset, it also generates a datalog.csv file. This file contains the image paths with corresponding steering angle, throttle, brakes, and speed of the car at that instance.

A few images from the dataset are shown below .

Center0001



Right0001

Left0001

Center0099

Right0099

Left0099

A sample of driving_log.csv file is shown in Figure 9.

Column 1, 2, 3: contains paths to the dataset images of center, right and left respectively
Column 4: contains the steering angle
Column value as 0 depicts straight, positive value is right turn and negative value is left turn.
Column 5: contains the throttle or acceleration at that instance
Column 6: contains the brakes or deceleration at that instance
Column 7: contains the speed of the vehicle

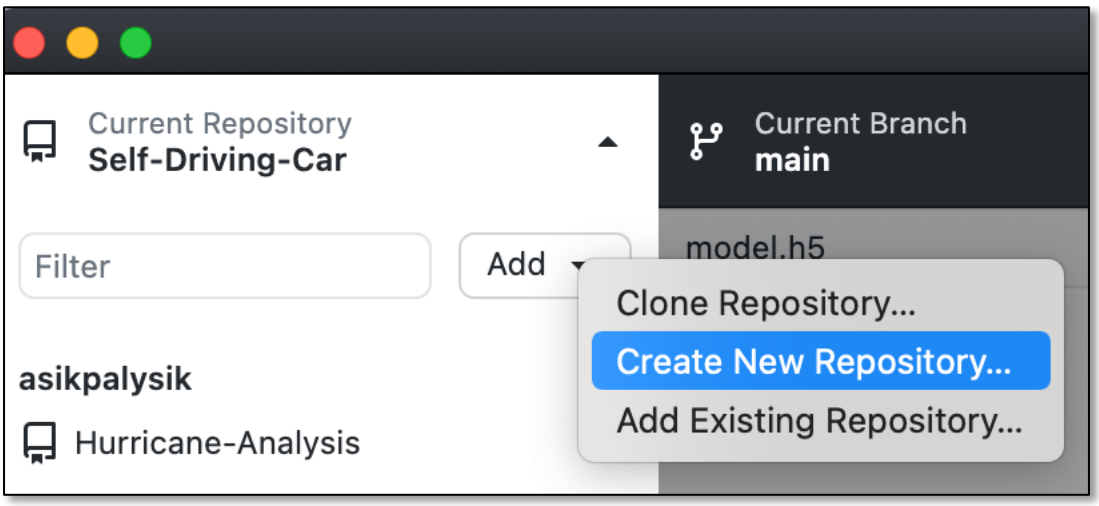
driving_log

/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_145.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_145.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_145.jpg	0	0	0	7.878379E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_259.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_259.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_259.jpg	0	0	0	7.791131E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_367.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_367.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_367.jpg	0	0	0	7.807511E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_484.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_484.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_484.jpg	0	0	0	7.795308E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_584.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_584.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_584.jpg	0	0	0	7.775518E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_719.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_719.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_719.jpg	0	0	0	7.80936E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_08_910.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_08_910.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_08_910.jpg	0	0	0	8.027026E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_031.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_031.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_031.jpg	0	0	0	8.001504E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_175.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_175.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_175.jpg	0	0	0	7.808222E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_310.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_310.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_310.jpg	0	0	0	7.82387E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_443.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_443.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_443.jpg	0	0	0	7.833329E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_561.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_561.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_561.jpg	0	0	0	7.902341E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_663.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_663.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_663.jpg	0	0	0	7.829595E-05
/Users/asik/Desktop/data/IMG/center_2021_12_14_20_51_09_773.jpg	/Users/asik/Desktop/data/IMG/left_2021_12_14_20_51_09_773.jpg	/Users/asik/Desktop/data/IMG/right_2021_12_14_20_51_09_773.jpg	0	0	0	7.761876E-05

Fig 9 . driving_log.csv

The Training Process

For the process of getting the self driving car working, we have to upload the images that we recorded using the simulator. First of all, we will open GitHub Desktop. If we do not have an account, we will create a new one. With that, we will create a new repository.



We will be using Google Colab for doing the training process or Kaggle.

We will open a new python3 notebook and get started. Next, we will git clone the repo.

```
!git clone https://github.com/Asikpalysik/Self-Driving-Car.git
```

We will now import all the libraries needed for training process. It will use Tensorflow backend and keras at frontend.

```
import os
from posixpath import splitext
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import keras
from numpy.core.fromnumeric import size
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Convolution2D, MaxPooling2D, Dropout, Flatten,
Dense
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from imgaug import augmenters as iaa
import cv2
import pandas as pd
import ntpath
import random
import warnings
warnings.filterwarnings("ignore")
```

We wil use datadir as the name given to the folder itself and take the parameters itself. Using head, we will show the first five values for the CSV on the desired format.

```
dir = "/Users/asik/Desktop/SelfDrivingCar"
columns = ["center", "left", "right", "steering", "throttle", "reverse", "speed"]
data = pd.read_csv(os.path.join(dir, "driving_log.csv"), names=columns)
pd.set_option("display.max_colwidth", -1)
data.head()
```

	center	left	right	steering	throttle	reverse	speed
0	/Users/asik/Desktop/Self Driving Car/IMG/center_2021_12_19_18_46_10_430.jpg	/Users/asik/Desktop/Self Driving Car/IMG/left_2021_12_19_18_46_10_430.jpg	/Users/asik/Desktop/Self Driving Car/IMG/right_2021_12_19_18_46_10_430.jpg	0.0	0.0	0.0	0.000079
1	/Users/asik/Desktop/Self Driving Car/IMG/center_2021_12_19_18_46_10_551.jpg	/Users/asik/Desktop/Self Driving Car/IMG/left_2021_12_19_18_46_10_551.jpg	/Users/asik/Desktop/Self Driving Car/IMG/right_2021_12_19_18_46_10_551.jpg	0.0	0.0	0.0	0.000079
2	/Users/asik/Desktop/Self Driving Car/IMG/center_2021_12_19_18_46_10_723.jpg	/Users/asik/Desktop/Self Driving Car/IMG/left_2021_12_19_18_46_10_723.jpg	/Users/asik/Desktop/Self Driving Car/IMG/right_2021_12_19_18_46_10_723.jpg	0.0	0.0	0.0	0.000080
3	/Users/asik/Desktop/Self Driving Car/IMG/center_2021_12_19_18_46_10_830.jpg	/Users/asik/Desktop/Self Driving Car/IMG/left_2021_12_19_18_46_10_830.jpg	/Users/asik/Desktop/Self Driving Car/IMG/right_2021_12_19_18_46_10_830.jpg	0.0	0.0	0.0	0.000080
4	/Users/asik/Desktop/Self Driving Car/IMG/center_2021_12_19_18_46_10_980.jpg	/Users/asik/Desktop/Self Driving Car/IMG/left_2021_12_19_18_46_10_980.jpg	/Users/asik/Desktop/Self Driving Car/IMG/right_2021_12_19_18_46_10_980.jpg	0.0	0.0	0.0	0.000079

As this is picking up the entire path from the local machine, we need to use ntpath function to get the network path assigned. We will declare a name path_leaf and assign accordingly.

```
def pathleaf(path):
    head, tail = ntpath.split(path)
    return tail
```

```
data["center"] = data["center"].apply(pathleaf)
data["left"] = data["left"].apply(pathleaf)
data["right"] = data["right"].apply(pathleaf)
data.head()
```

	center	left	right	steering	throttle	reverse	speed
0	center_2021_12_19_18_46_10_430.jpg	left_2021_12_19_18_46_10_430.jpg	right_2021_12_19_18_46_10_430.jpg	0.0	0.0	0.0	0.000079
1	center_2021_12_19_18_46_10_551.jpg	left_2021_12_19_18_46_10_551.jpg	right_2021_12_19_18_46_10_551.jpg	0.0	0.0	0.0	0.000079
2	center_2021_12_19_18_46_10_723.jpg	left_2021_12_19_18_46_10_723.jpg	right_2021_12_19_18_46_10_723.jpg	0.0	0.0	0.0	0.000080
3	center_2021_12_19_18_46_10_830.jpg	left_2021_12_19_18_46_10_830.jpg	right_2021_12_19_18_46_10_830.jpg	0.0	0.0	0.0	0.000080
4	center_2021_12_19_18_46_10_980.jpg	left_2021_12_19_18_46_10_980.jpg	right_2021_12_19_18_46_10_980.jpg	0.0	0.0	0.0	0.000079

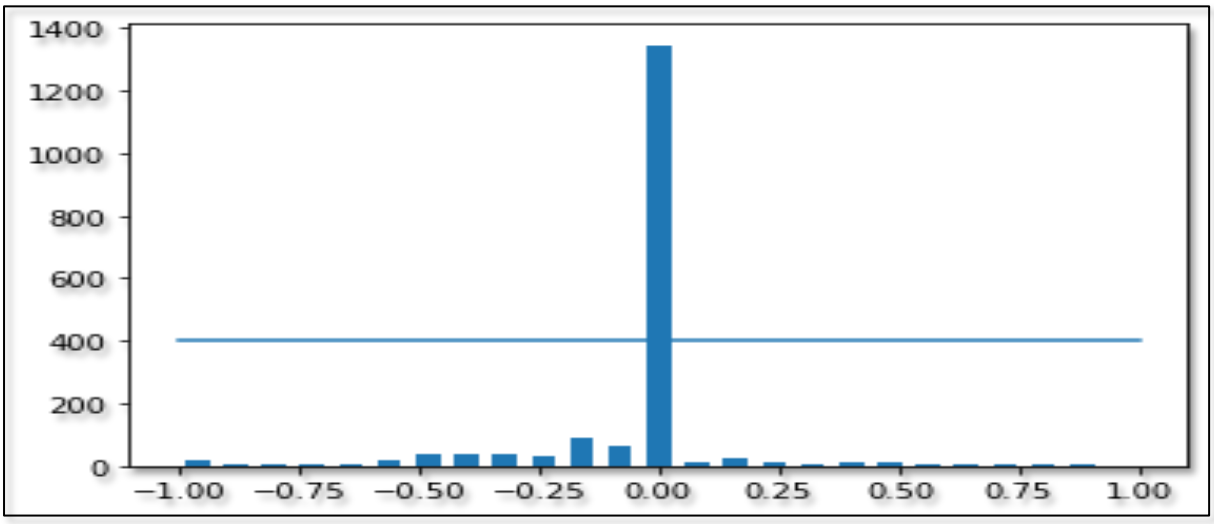
We will bin the number of values where the number will be equal to 25 (odd number aimed to get center distribution). We will see the histogram using the np.histogram option on data frame ‘steering’, we will divide it to the number of bins. We keep samples at 400 and then we draw a line. We see the data is centered along the middle that is 0.

```
num_bins = 25
samples_per_bin = 400
hist, bins = np.histogram(data["steering"], num_bins)
print(bins)
```

```
[-1.   -0.92 -0.84 -0.76 -0.68 -0.6  -0.52 -0.44 -0.36 -0.28 -0.2  -0.12
 -0.04  0.04  0.12  0.2   0.28  0.36  0.44  0.52  0.6   0.68  0.76  0.84
  0.92  1.  ]
```

Plot on it

```
center = (bins[:-1] + bins[1:]) * 0.5
plt.bar(center, hist, width=0.05)
plt.plot(
    (np.min(data["steering"]), np.max(data["steering"])),
    (samples_per_bin, samples_per_bin),
)
```



```
print("Total Data:", len(data))
->>Total Data: 1795
```

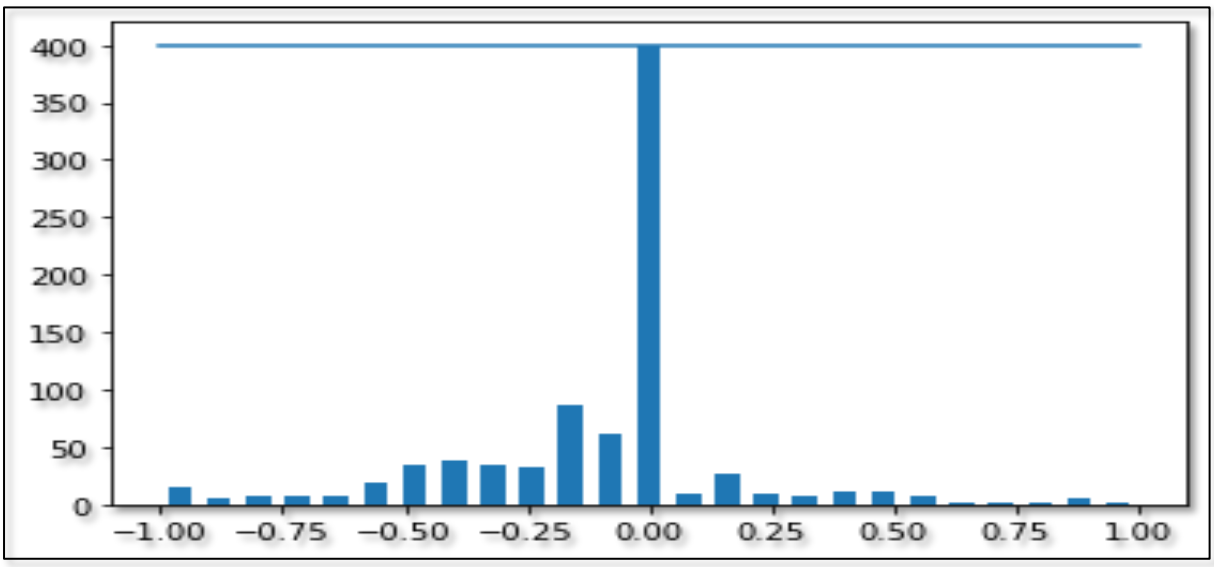
We wil specify a variable rove_list.We will specify samples we want to remove using looping construct through every single bin we will iterate through all the steering data. We will shuffle the data and romve some from it as it is now uniformly structured after shuffling.The output will be the distribution of steering angle that are much more uniform. There are significant amount of left steering angle and right steering angle eliminating the bias to drive straight all the time.

```
remove_list = []
for j in range(num_bins):
    list_ = []
    for i in range(len(data["steering"])):
        if data["steering"][i] >= bins[j] and data["steering"][i] <= bins[j + 1]:
            list_.append(i)
    list_ = shuffle(list_)
    list_ = list_[samples_per_bin:]
    remove_list.extend(list_)
print("Removed:", len(remove_list))
->> Removed: 945
```

```
data.drop(data.index[remove_list], inplace=True)
print("Remaining:", len(data))
->>Remaining: 850
```

Plot on it

```
hist, _ = np.histogram(data["steering"], (num_bins))
plt.bar(center, hist, width=0.05)
plt.plot(
    (np.min(data["steering"]), np.max(data["steering"])),
    (samples_per_bin, samples_per_bin),
)
```



```
print(data.iloc[1])
```

```
center      center_2021_12_19_18_46_10_980.jpg
left        left_2021_12_19_18_46_10_980.jpg
right       right_2021_12_19_18_46_10_980.jpg
steering    0.0
throttle    0.0
reverse     0.0
speed       0.000079
Name: 4, dtype: object
```

We will now load the image into array to manipulate them accordingly. We will define a function named locd_img_steering. We will have image path as empty list and steering as empty list and then loop through. We use iloc selector as data frame based on the specific index, we will use cut data for now.

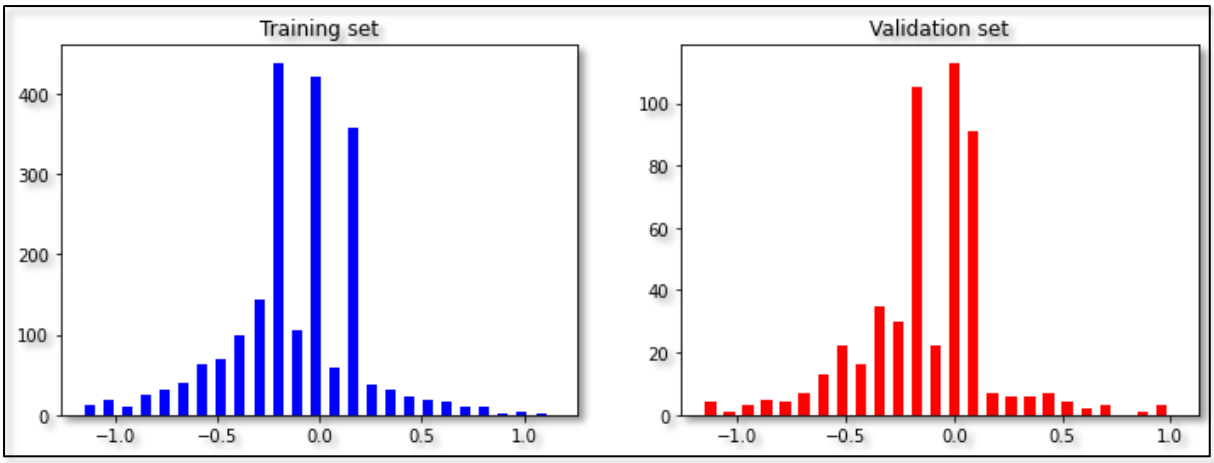
```
def load_img_steering(datadir, df):
    image_path = []
    steering = []
    for i in range(len(data)):
        indexed_data = data.iloc[i]
        center, left, right = indexed_data[0], indexed_data[1], indexed_data[2]
        image_path.append(os.path.join(datadir, center.strip()))
        steering.append(float(indexed_data[3]))
        image_path.append(os.path.join(datadir, left.strip()))
        steering.append(float(indexed_data[3]) + 0.15)
        image_path.append(os.path.join(datadir, right.strip()))
        steering.append(float(indexed_data[3]) - 0.15)
    image_paths = np.asarray(image_path)
    steerings = np.asarray(steering)
    return image_paths, steerings
```

We will be splitting the image path as well as storing arrays accordingly.

```
image_paths, steerings = load_img_steering(dir + "/IMG", data)
X_train, X_valid, y_train, y_valid = train_test_split(
    image_paths, steerings, test_size=0.2, random_state=6
)
print("Training Samples: {}\nValid Samples: {}".format(len(X_train), len(X_valid)))
->>Training Samples: 2040
->>Valid Samples: 510
```

We will have the histograms now.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].hist(y_train, bins=num_bins, width=0.05, color="blue")
axes[0].set_title("Training set")
axes[1].hist(y_valid, bins=num_bins, width=0.05, color="red")
axes[1].set_title("Validation set")
```



Augmentation and image pre-processing

The biggest challenge was generalizing the behavior of the car on Track_2 which it was never trained for. In a real-life situation, we can never train a self-driving car model for every track possible, as the data will be too huge to process. Also, it is not possible to gather the dataset for all the weather conditions and roads. Thus, there is a need to come up with an idea of generalizing the behavior on different tracks. This problem is solved using image preprocessing and augmentation techniques.

• **Zoom**

The images in the dataset have relevant features in the lower part where the road is visible. The external environment above a certain image portion will never be used to determine the output and thus can be cropped. Approximately, 30% of the top portion of the image is cut and passed in the training set. The snippet of code and transformation of an image after cropping and resizing it to original image can be seen in below.

to

```
def zoom(image):
    zoom = iaa.Affine(scale=(1, 1.3))
    image = zoom.augment_image(image)
    return image


image = image_paths[random.randint(0, 1000)]
original_image = mpimg.imread(image)
zoomed_image = zoom(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()
```

```
axs[0].imshow(original_image)
axs[0].set_title("Original Image")

axs[1].imshow(zoomed_image)
axs[1].set_title("Zoomed Image")
```

• **Flip (horizontal)**
The image is flipped horizontally (i.e. a mirror image of the original image is passed to the dataset). The motive behind this is that the model gets trained for similar kinds of turns on opposite sides too. This is important because Track 1 includes only left turns. The snippet of code and transformation of an image after flipping it can be seen in below.


to


```
def random_flip(image, steering_angle):
    image = cv2.flip(image, 1)
    steering_angle = -steering_angle
    return image, steering_angle


random_index = random.randint(0, 1000)
image = image_paths[random_index]
steering_angle = steerings[random_index]

original_image = mpimg.imread(image)
flipped_image, flipped_steering_angle = random_flip(original_image, steering_angle)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title("Original Image - " + "Steering Angle:" + str(steering_angle))

axs[1].imshow(flipped_image)
axs[1].set_title("Flipped Image - " + "Steering Angle:" +
str(flipped_steering_angle))
```

• **Shift (horizontal/vertical)**
The image is shifted by a small amount, it is vertical shift and horizontal shift as below.

 to 

 to 

```
def pan(image):
    pan = iaa.Affine(translate_percent={"x": (-0.1, 0.1), "y": (-0.1, 0.1)})
    image = pan.augment_image(image)
    return image


image = image_paths[random.randint(0, 1000)]
original_image = mpimg.imread(image)
panned_image = pan(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title("Original Image")

axs[1].imshow(panned_image)
axs[1].set_title("Panned Image")
```

• **Brightness**

To generalize to the weather conditions with bright sunny day or cloudy, lowlight conditions, the brightness augmentation can prove to be very useful. The code snippet and increase of brightness can be seen below. Similarly, I have randomly also lowered down the level of brightness for other conditions.

 to 

```
def random_brightness(image):
    brightness = iaa.Multiply((0.2, 1.2))
    image = brightness.augment_image(image)
    return image


image = image_paths[random.randint(0, 1000)]
original_image = mpimg.imread(image)
brightness_altered_image = random_brightness(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title("Original Image")

axs[1].imshow(brightness_altered_image)
axs[1].set_title("Brightness altered image ")
```

To have a look what we have at this moment

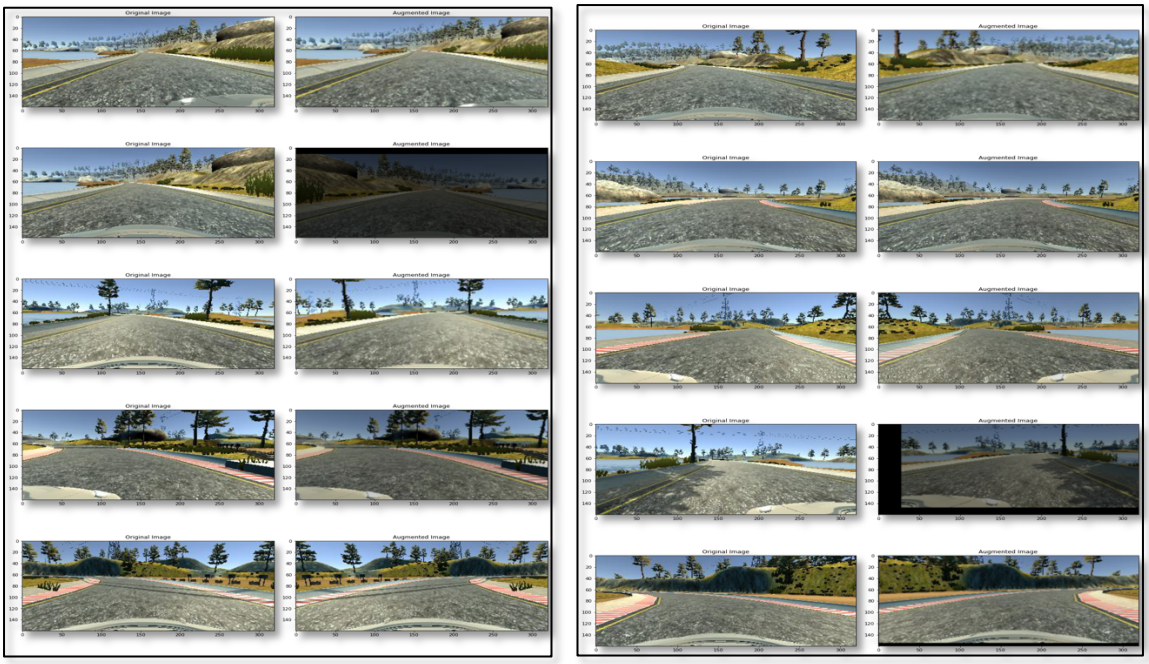
```
def random_augment(image, steering_angle):
    image = mpimg.imread(image)
    if np.random.rand() < 0.5:
        image = pan(image)
    if np.random.rand() < 0.5:
        image = zoom(image)
    if np.random.rand() < 0.5:
        image = random_brightness(image)
    if np.random.rand() < 0.5:
        image, steering_angle = random_flip(image, steering_angle)
    return image, steering_angle


ncol = 2
nrow = 10

fig, axs = plt.subplots(nrow, ncol, figsize=(15, 50))
fig.tight_layout()

for i in range(10):
    randnum = random.randint(0, len(image_paths) - 1)
    random_image = image_paths[randnum]
    random_steering = steerings[randnum]
    original_image = mpimg.imread(random_image)
    augmented_image, steering = random_augment(random_image, random_steering)
    axs[i][0].imshow(original_image)
    axs[i][0].set_title("Original Image")

    axs[i][1].imshow(augmented_image)
    axs[i][1].set_title("Augmented Image")
```



I continued by doing some image processing. I cropped the image to remove the unnecessary features, changes the images to YUV format, used gaussian blur, decreased the size for easier processing and normalized the values.

```
ef img_preprocess(img):
    ## Crop image to remove unnecessary features
    img = img[60:135, :, :]
    ## Change to YUV image
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    ## Gaussian blur
    img = cv2.GaussianBlur(img, (3, 3), 0)
    ## Decrease size for easier processing
    img = cv2.resize(img, (200, 66))
    ## Normalize values
    img = img / 255
    return img
```

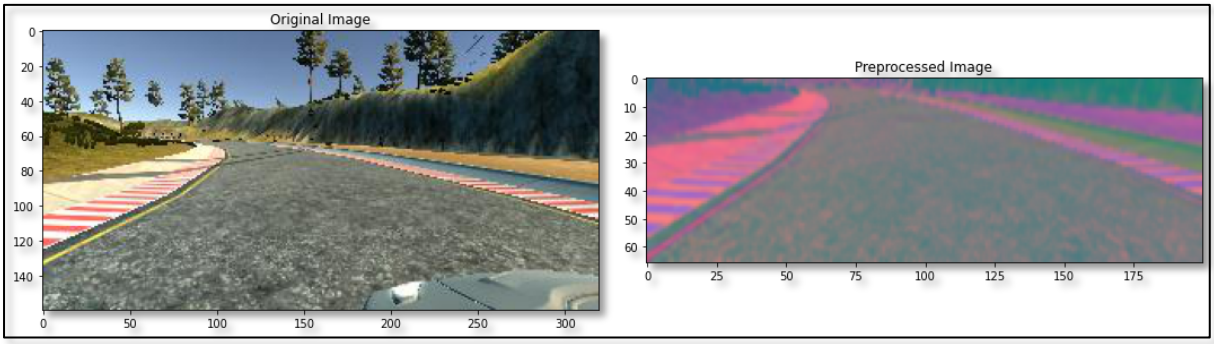
To compare and visualize I plotted the original and the pre-processed image.

```
image = image_paths[100]
original_image = mpimg.imread(image)
preprocessed_image = img_preprocess(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title("Original Image")

axs[1].imshow(preprocessed_image)
axs[1].set_title("Preprocessed Image")
```




```
def batch_generator(image_paths, steering_ang, batch_size, istraining):
    while True:
        batch_img = []
        batch_steering = []

        for i in range(batch_size):
            random_index = random.randint(0, len(image_paths) - 1)

            if istraining:
                im, steering = random_augment(
                    image_paths[random_index], steering_ang[random_index]
                )

            else:
                im = mpimg.imread(image_paths[random_index])
                steering = steering_ang[random_index]

            im = img_preprocess(im)
            batch_img.append(im)
            batch_steering.append(steering)

        yield (np.asarray(batch_img), np.asarray(batch_steering))
```

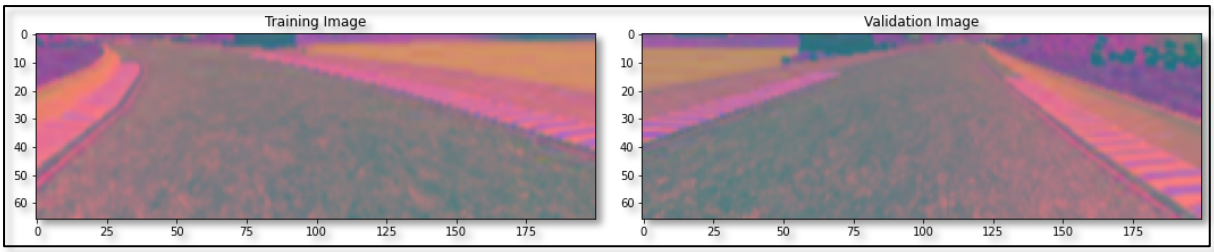
So far so good. Next, I converted all the images into numpy array

```
x_train_gen, y_train_gen = next(batch_generator(X_train, y_train, 1, 1))
x_valid_gen, y_valid_gen = next(batch_generator(X_valid, y_valid, 1, 0))

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(x_train_gen[0])
axs[0].set_title("Training Image")

axs[1].imshow(x_valid_gen[0])
axs[1].set_title("Validation Image")
```



Experimental configurations

Configurations used to set up the models for training the Python Client to provide the Neural Network outputs that drive the car on the simulator. The tweaking of parameters and rigorous experiments were tried to reach the best combination. Though each of the models had their unique behaviors and differed in their performance with each tweak, the following combination of configuration can be considered as the optimal:

- The sequential models built on Keras with deep neural network layers are used to train the data.
- Models are only trained using the dataset from Track 1.
- 80% of the dataset is used for training, 20% is used for testing.
- Epochs = 10, i.e. number of iterations or passes through the complete dataset. Experimented with larger number of epochs also, but the model tried to “overfit”. In other words, the model learns the details in the training data too well, while impacting the performance on new dataset.
- Batch-size = 100, i.e. number of image samples propagated through the network, like a subset of data as complete dataset is too big to be passed all at once.
- Learning rate = 0.0001, i.e. how the coefficients of the weights or gradients change in the network.

There are different combinations of Convolution layer, Time-Distributed layer, MaxPooling layer, Flatten, Dropout, Dense and so on, that can be used to implement the Neural Network models.

Network architectures

The design of the network is based on the NVIDIA model, which has been used by NVIDIA for the end-to-end self driving test. As such, it is well suited for the project.It is a deep convolution network which works well with supervised image classification / regression problems. As the NVIDIA model is well documented, I was able to focus how to adjust the training images to produce the best result with some adjustments to the model to avoid overfitting and adding non-linearity to improve the prediction.

I've added the following adjustments to the model.

- I used Lambda layer to normalized input images to avoid saturation and make gradients work better.
- I've added an additional dropout layer to avoid overfitting after the convolution layers.
- I've also included ELU for activation function for every layer except for the output layer to introduce non-linearity.

In the end, the model looks like as follows:

- Image normalization
- Convolution: 5x5, filter: 24, strides: 2x2, activation: ELU
- Convolution: 5x5, filter: 36, strides: 2x2, activation: ELU
- Convolution: 5x5, filter: 48, strides: 2x2, activation: ELU
- Convolution: 3x3, filter: 64, strides: 1x1, activation: ELU
- Convolution: 3x3, filter: 64, strides: 1x1, activation: ELU
- Drop out (0.5)
- Fully connected: neurons: 100, activation: ELU
- Fully connected: neurons: 50, activation: ELU
- Fully connected: neurons: 10, activation: ELU
- Fully connected: neurons: 1 (output)

Layer (type)	Output Shape	Params	Connected to
lambda_1 (Lambda)	(None, 66, 200, 3)	0	lambda_input_1
convolution2d_1 (Convolution2D)	(None, 31, 98, 24)	1824	lambda_1
convolution2d_2 (Convolution2D)	(None, 14, 47, 36)	21636	convolution2d_1
convolution2d_3 (Convolution2D)	(None, 5, 22, 48)	43248	convolution2d_2
convolution2d_4 (Convolution2D)	(None, 3, 20, 64)	27712	convolution2d_3
convolution2d_5 (Convolution2D)	(None, 1, 18, 64)	36928	convolution2d_4
dropout_1 (Dropout)	(None, 1, 18, 64)	0	convolution2d_5
flatten_1 (Flatten)	(None, 1152)	0	dropout_1
dense_1 (Dense)	(None, 100)	115300	flatten_1
dense_2 (Dense)	(None, 50)	5050	dense_1
dense_3 (Dense)	(None, 10)	510	dense_2
dense_4 (Dense)	(None, 1)	11	dense_3
	Total params	252219	

As per the NVIDIA model, the convolution layers are meant to handle feature engineering and the fully connected layer for predicting the steering angle. However, as stated in the NVIDIA document, it is not clear where to draw such a clear distinction. Overall, the model is very functional to clone the given steering behavior. The below is a model structure output from the Keras which gives more details on the shapes and the number of parameters.

We will design our Model architecture. We have to classify the traffic signs too that's why we need to shift from Lenet 5 model to NVDIA model. With behavioural cloning, our dataset is much more complex then any dataset we have used. We are dealing with images that have (200,66) dimensions. Our current datset has 5386 images to train with but MNSIT has around 60,000 images to train with. Our behavioural cloning code has simply has to return appropriate steering angle which is a regression type example. For these things, we need a more advanced model which is provided by nvdia and known as nvdia model.

For defining the model architecture, we need to define the model object. Normalization state can be skipped as we have already normalized it. We will add the convolution layer. As

compared to the model, we will organize accordingly. The Nvdia model uses 24 filters in the layer along with a kernel of size 5,5. We will introduce sub sampling. The function reflects to stride length of the kernel as it processes through an image, we have large images. Horizontal movement with 2 pixels at a time, similarly vertical movement to 2 pixels at a time. As this is the first layer, we have to define input shape of the model too i.e., (66,200,3) and the last function is an activation function that is “elu”.

Revisting the model, we see that our second layer has 36 filters with kernel size (5,5) same subsampling option with stride length of (2,2) and conclude this layer with activation ‘elu’.

According to Nvdia model, it shows we have 3 more layers in the convolutional neural network. With 48 filters, with 64 filters (3,3) kernel 64 filters (3,3) kernel Dimensions have been reduced significantly so for that we will remove subsampling from 4th and 5th layer.

Next we add a flatten layer. We will take the output array from previous convolution neural network to convert it into a one dimensional array so that it can be fed to fully connected layer to follow.

Our last convolution layer outputs an array shape of (1,18) by 64.

We end the architecture of Nvdia model with a dense layer containing a single output node which will output the predicted steering angle for our self driving car. Now we will use model.compile() to compile our architecture as this is a regression type example the metrics that we will be using will be mean squared error and optimize as Adam. We will be using relatively a low learning rate that it can help on accuracy. We will use dropout layer to avoid overfitting the data. Dropout Layer sets the input of random fraction of nodes to “0” during each update. During this, we will generate the training data as it is forced to use a variety of combination of nodes to learn from the same data. We will have to separate the convolution layer from fully connected layer with a factor of 0.5 is added so it converts 50 percent of the input to 0. We Will define the model by calling the nvdia model itself. Now we will have the model training process.To define training parameters, we will use model.fit(), we will import our training data X_Train, training data ->y_train, we have less data on the datasets we will require more epochs to be effective. We will use validation data and then use Batch size.

```
def NvidiaModel():
  model = Sequential()

model.add(Convolution2D(24,(5,5),strides=(2,2),input_shape=(66,200,3),activation="e
lu"))
  model.add(Convolution2D(36,(5,5),strides=(2,2),activation="elu"))
  model.add(Convolution2D(48,(5,5),strides=(2,2),activation="elu"))
  model.add(Convolution2D(64,(3,3),activation="elu"))
  model.add(Convolution2D(64,(3,3),activation="elu"))
  model.add(Dropout(0.5))
  model.add(Flatten())
  model.add(Dense(100,activation="elu"))
  model.add(Dropout(0.5))
  model.add(Dense(50,activation="elu"))
  model.add(Dropout(0.5))
  model.add(Dense(10,activation="elu"))
```

```
  model.add(Dropout(0.5))
  model.add(Dense(1))
  model.compile(optimizer=Adam(lr=1e-3),loss="mse")
  return model
```

```
model = NvidiaModel()
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 31, 98, 24)        1824
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 47, 36)        21636
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 5, 22, 48)         43248
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 20, 64)         27712
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
dropout (Dropout)            (None, 1, 18, 64)         0
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 100)               115300
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
show more (open the raw output data in a text editor) ...

Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
_________________________________________________________________
None
```

Results

The following results were observed for described architectures. I had to come up with two different performance metrics.

- Value loss or Accuracy (computed during training phase)
- Generalization on Track 1 (drive performance)

Value loss or Accuracy

The first evaluation parameter considered here is “Loss” over each epoch of the training run. To calculate value loss over each epoch, Keras provides “val_loss”, which is the average loss after that epoch. The loss observed during the initial epochs at the beginning of training phase is high, but it falls gradually, and that is evident by the screenshots below which shows the run of Architecture in the training phase.

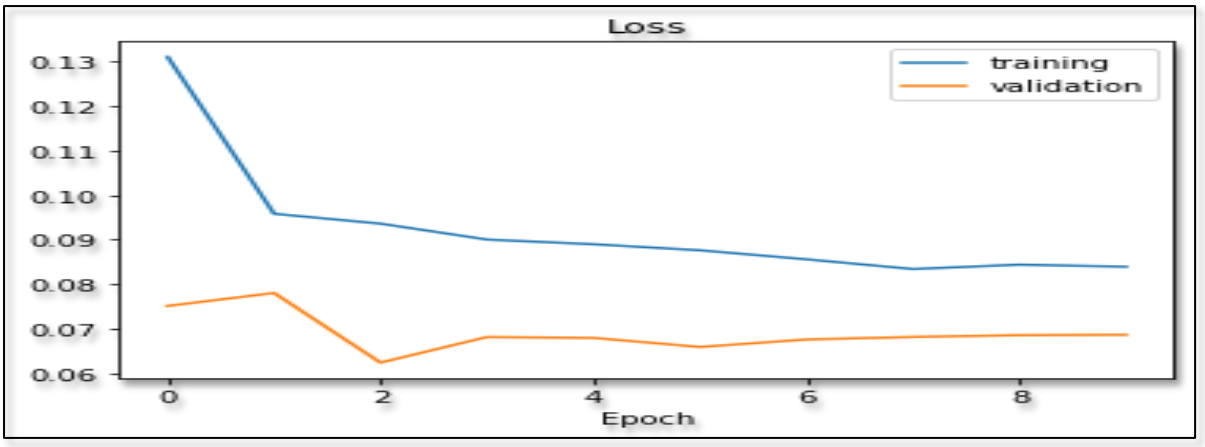
```
history = model.fit_generator(
    batch_generator(X_train, y_train, 100, 1),
    steps_per_epoch=300,
    epochs=10,
    validation_data=batch_generator(X_valid, y_valid, 100, 0),
    validation_steps=200,
    verbose=1,
    shuffle=1,
)
```

```
Epoch 1/10
300/300 [==============================] - 317s 1s/step - loss: 0.1311 - val_loss: 0.0751
Epoch 2/10
300/300 [==============================] - 260s 867ms/step - loss: 0.0958 - val_loss: 0.0780
Epoch 3/10
300/300 [==============================] - 252s 842ms/step - loss: 0.0936 - val_loss: 0.0624
Epoch 4/10
300/300 [==============================] - 266s 890ms/step - loss: 0.0900 - val_loss: 0.0681
Epoch 5/10
300/300 [==============================] - 250s 836ms/step - loss: 0.0890 - val_loss: 0.0679
Epoch 6/10
300/300 [==============================] - 263s 877ms/step - loss: 0.0876 - val_loss: 0.0659
Epoch 7/10
300/300 [==============================] - 256s 854ms/step - loss: 0.0856 - val_loss: 0.0676
Epoch 8/10
300/300 [==============================] - 230s 769ms/step - loss: 0.0834 - val_loss: 0.0681
Epoch 9/10
300/300 [==============================] - 239s 799ms/step - loss: 0.0844 - val_loss: 0.0685
Epoch 10/10
300/300 [==============================] - 238s 796ms/step - loss: 0.0839 - val_loss: 0.0686
```

Why We Use ELU Over RELU

We can have dead relu this is when a node in neural network essentially dies and only feeds a value of zero to nodes which follows it. We will change from relu to elu. Elu function has always a chance to recover and fix it errors means it is in a process of learning and contributing to the model. We will plot the model and then save it accordingly in h5 format for a keras file.

```
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.legend(["training", "validation"])
plt.title("Loss")
plt.xlabel("Epoch")
```

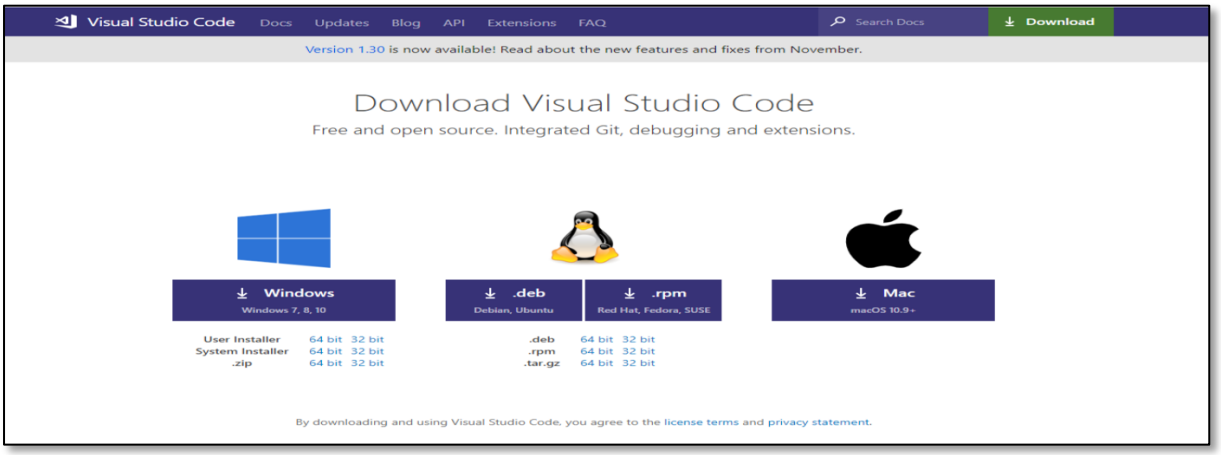


Save model

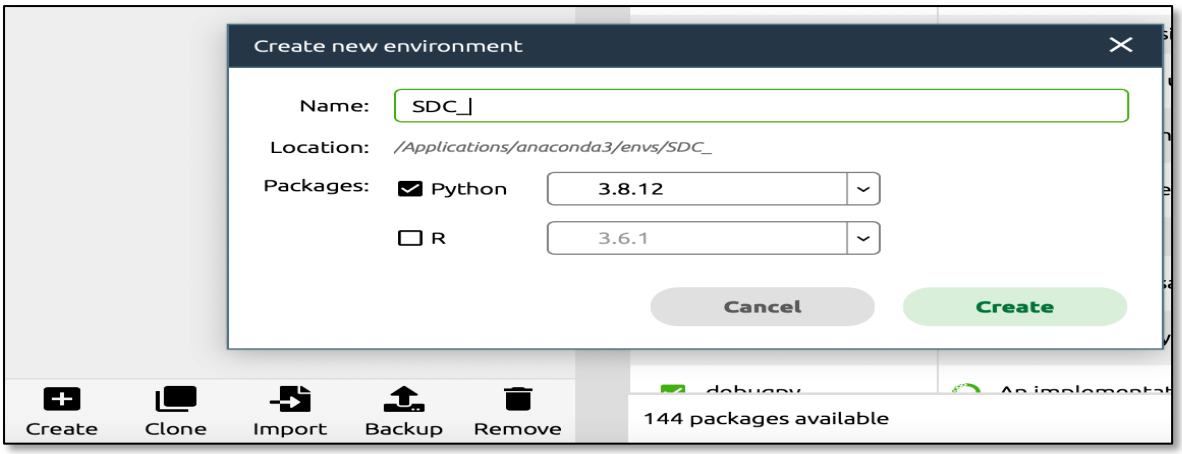
```
model.save('model.h5')
```

The Connection Part

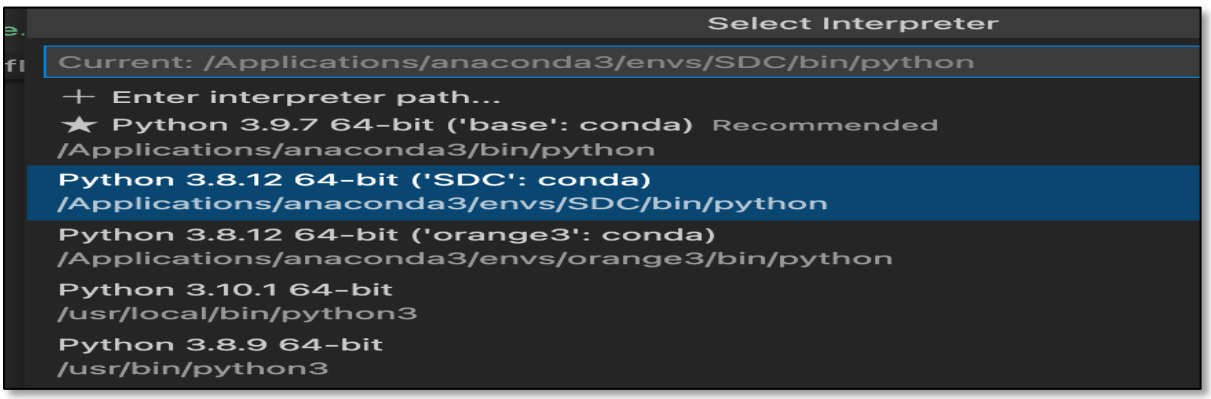
This step is required to run the model in the simulated car. For implementing web service using python, we need to install flask. We will use Anaconda environment. Flask is a python micro framework that is used to build the web app. We will use Visual Studio Code.

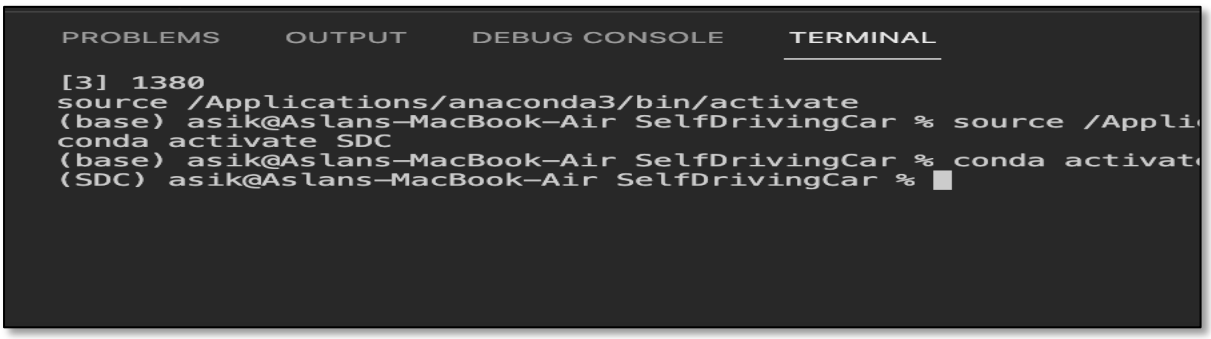


We will open the folder where we kept the saved *.h5 file, then again open a file but before that, we will install some dependencies. We will also create an anaoconda environment too for doing our work. Click Create Python 3.8.12. with my experience I need to mention that i had some problems with environments, the was a lot of conflicts so I will advise you just to follow my steps exactly the same.



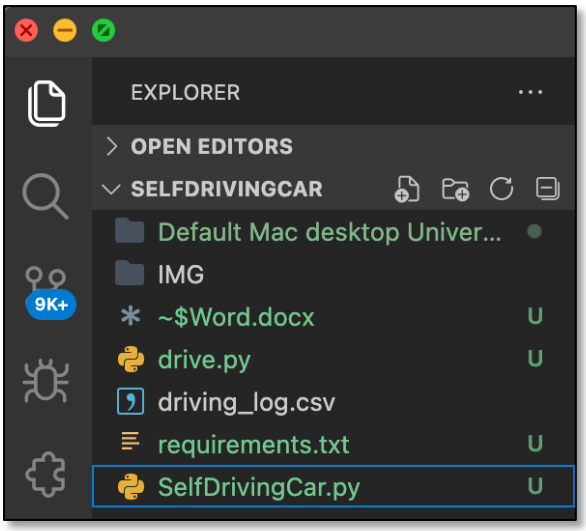
After we created new env on Python 3.8.12 moving to VScode and opening it under SDC environment. So basically, we now will work with special environment on special Python version. Terminal need to be as well under same.





I will create a list `conda list -e > requirements.txt` **requirements.txt** for you to know what exactly I used there. It important to use `keras 2.4.3` and be careful with `python-engineio=3.13.0` and lastly most important `python-socketio=4.6.1`(follow requirements.txt). As well you will see **drive.py** file without this code will not work you can simple copy paste it I will not go deep about this file.

Now when all your requirements are installed, we are ready to run magic code in terminal. In your folder you will see something like I show below.



Just type `python drive.py model.h5` in your Terminal. wait few few second open Udacity Simulator and choose option Autonomous mode. That it, you will see something as I show below.

Files

The project contains the following files:

- *SelfDrivingCar.py* (script used to create the model and train the model)
- *drive.py* (script to drive the car - feel free to modify this file)
- *driving_log.csv* (csv file from simulator - data)
- *IMG (folder)* (folder with images from simulator - data)
- *model.h5* (a trained Keras model)
- *requirements.txt* (important requirements for project)
- *SelfDrivingCar.ipynb* (full explanation in notebook)
- *SelfDrivingCar.pdf (*full explanation in PDF)

Overview

In this project, we use deep neural networks and convolutional neural networks to clone driving behavior. The model is trained, validated and tested using Keras. The model outputs a

steering angle to an autonomous vehicle. The autonomous vehicle is provided as a simulator. Image data and steering angles are used to train a neural network and drive the simulation car autonomously around the track.
This project started with training the models and tweaking parameters to get the best performance on the track
The use of CNN for getting the spatial features and RNN for the temporal features in the image dataset makes this combination a great fit for building fast and lesser computation required neural networks. Substituting recurrent layers for pooling layers might reduce the loss of information and would be worth exploring in the future projects.
It is interesting to find the use of combinations of real world dataset and simulator data to train these models. Then I can get the true nature of how a model can be trained in the simulator and generalized to the real world or vice versa. There are many experimental implementations carried out in the field of self-driving cars and this project contributes towards a significant part of it.

References

GitHub - https://github.com/woges/UDACITY-self-driving-car/tree/master/term1_project3_behavioral_cloning
GitHub - https://github.com/SakshayMahna/P4-BehavioralCloning
Deep Learning for Self-Driving Cars - https://towardsdatascience.com/deep-learning-for-self-driving-cars-7f198ef4cfa2
GitHub - https://github.com/llSourcell/How_to_simulate_a_self_driving_car
GitHub - https://github.com/naokishibuya/car-behavioral-cloning
End to End Learning for Self-Driving Cars - https://arxiv.org/pdf/1604.07316v1.pdf
Aditya Babhulkar - https://scholarworks.calstate.edu/downloads/fx719m76s
GitHub -https://github.com/udacity/self-driving-car-sim