3.4 Behavioral Cloning

In behavioural cloning an amalgamation of Deep neural networks, Convolutional neural networks and feature extraction is used. We would train a model based on the data from our manual driving and then the model would clone our behaviour (steering angle) to drive the car autonomously. For the purpose of this project, the process of data collection is done through a state of the art simulator prepared by Udacity in collaboration with unity. So, in this part of the project we would first collect the data for the model through the simulator, then train our convolutional neural network on that data and then run the model on the autonomous mode of the simulator to see if it works correctly.

3.4.1 Data Collection

Data extraction for our model is done with the help of a simulator. This simulator is prepared by the Udacity along with unity and it is present for download at their GitHub account. The simulator has 2 modes:

1. Training mode: This mode is used for the data collection. Later our model would be trained on that data.
2. Autonomous mode: In this mode we would be testing our model on how it behaves on a general track.

In driving mode, we use keyboard's directional keys to drive the car in the simulator on the track and then collect the data of the steering angle and pictures from three points of view (centre, left and right) for that particular steering angle. Driving in the training mode requires perfection for obtaining clean data. It is suggested that one should practice 4-5 laps before collecting data. For data collection 3 laps of one track are considered sufficient. While collecting data it is suggested to stay at the centre of the track for clear data collection. As observed in the training mode the track was biased towards the left direction so to reduce this bias we drove the car in opposite direction for 3 laps thus obtaining a balanced data of steering angle and reducing bias towards the left direction. Car is mounted with three cameras, one on centre, one on the left and one on the right. The idea of mounting 3 cameras on the vehicle was proposed by NVIDIA in order to generalize and diversify our data more and simulating the effect of car being in different positions. Steering angle data is collected in a csv file and images from three mounted cameras in that particular angle is also recorded. We also collected the data for brake, throttle and speed but it is not of much use in our project as it is oriented around the steering angle. Steering angle collected in the csv file is in radians and it is adjusted in the range of -1 to 1 for better labelling of the images. To put pictures in words or to summarize the data collection process what we actually did is take pictures from the mounted camera which would be our data and we would label these pictures with the steering angles thus helping our neural network to learn from these features, in this case the features would be:

1. Steering angle.
2. Lane lines and boundaries of the road in the image.
3. Brightness or colour intensity of the image.

And many more features.

One thing that is to be noted here is that this is not a classification problem. It is a regression based problem. Even though we are labelling the images with particular numbers but the data is not being classified as a particular value instead we are developing a particular output based on a regression based model.

The data that has been collected here is then uploaded on google collab as that was the platform we used for preparation of our deep neural network model and pre-processing of data. We would be using the pandas library for reading the data.

	center	left	right	steering	throttle	reverse	speed
0	center_2018_07_16_17_11_43_382.jpg	left_2018_07_16_17_11_43_382.jpg	right_2018_07_16_17_11_43_382.jpg	0.0	0.0	0.0	0.649786
1	center_2018_07_16_17_11_43_670.jpg	left_2018_07_16_17_11_43_670.jpg	right_2018_07_16_17_11_43_670.jpg	0.0	0.0	0.0	0.627942
2	center_2018_07_16_17_11_43_724.jpg	left_2018_07_16_17_11_43_724.jpg	right_2018_07_16_17_11_43_724.jpg	0.0	0.0	0.0	0.622910
3	center_2018_07_16_17_11_43_792.jpg	left_2018_07_16_17_11_43_792.jpg	right_2018_07_16_17_11_43_792.jpg	0.0	0.0	0.0	0.619162
4	center_2018_07_16_17_11_43_860.jpg	left_2018_07_16_17_11_43_860.jpg	right_2018_07_16_17_11_43_860.jpg	0.0	0.0	0.0	0.615438

Fig 3.8 *snippet of data collected from the simulator*

3.4.2 Data Balancing

As it can be seen in the graph below, our steering angle data is biased towards the centre or zero. To remove this bias, we have to take some steps for balancing of the dataset. The graph shows the frequency of each driving angle. So, one method for balancing is to a limit to the number of inputs for each steering angle. That is, if any steering angle has a frequency greater than the set limit, it would not be considered in the dataset.

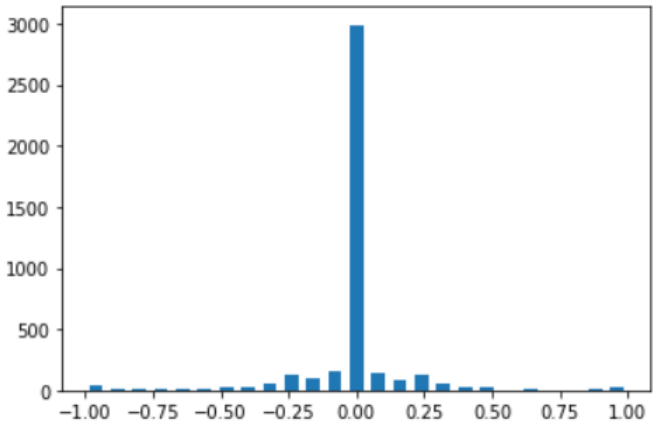


Fig 3.9 Histogram showing the frequency of each steering angle from -1 to 1

So, we set the limit of 400 samples per steering angle. Hence no more data was included once the threshold was crossed.
Our data consisted of 4093 samples. After setting the threshold 2590 sample points were removed and remaining 1463 sample points were used.

Other methods of data balancing may include drifting towards the edge of the road without hitting the edge or getting derailed. This way we can remove the zero bias by increasing the frequency of steering angles towards the outer regions. One other way that can be employed for this task is to stop recording

the data while driving towards the edge and start recording while we move towards the centre of the lane. This can also remove the zero degree steering angle bias.

3.4.3 TRAINING AND VALIDATION SPLIT

The data that we collected is now divided into two parts that is training dataset and validation dataset. As the name suggests, training dataset is used for training of our model. Whereas the validation dataset is part of the dataset that has been held back and later used for testing the accuracy or performance of our model.
The first step of this process is labelling the images in our dataset by the corresponding steering angle. For that we used the iloc function which helps us access each element of a row in a table. Using this function, we create two lists, one for images and one for steering angles. We append the images from centre camera in the list with corresponding steering angles as it is. But with the images from left and right camera we need to append the steering angle with slight modification of 0.15 in that particular direction. This is done to prevent the bias.
Now, the splitting is done using a function called train_test_split. Out of 4093 samples around 20% are reserved for the test or validation dataset and rest is for training dataset. One thing that is to be noticed is that both training and validation dataset must have similar distribution of steering angle. That means we do not want bias towards a particular steering angle in both the dataset. If there is high bias in training dataset, then our model would be under fitting. Below plot shows the comparison between the steering angle data for training and Validation dataset.

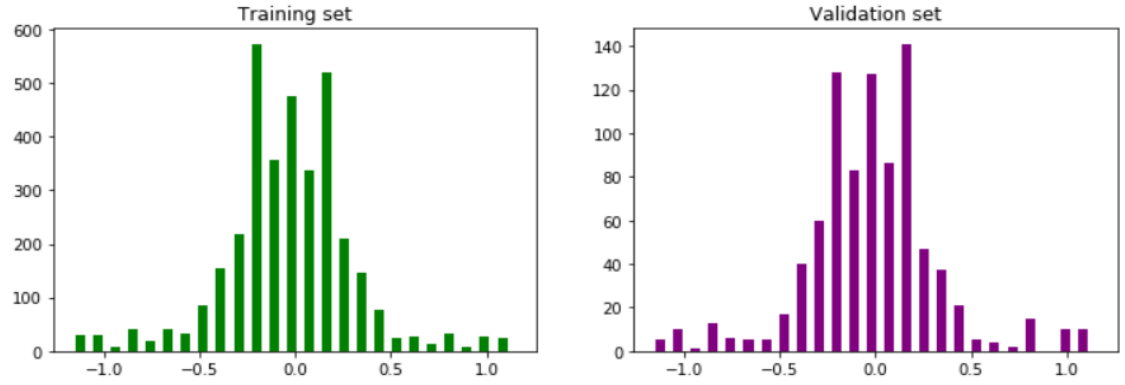


Fig 3.9: Comparison between steering angle data for training and validation

The model will be trained on training dataset and later tested on validation dataset to check whether it is overfitting, under fitting or just right!

3.4.4 PREPROCESSING IMAGES

Pre-processing of images in our dataset is a fundamental step. Pre-processing is done to improve the quality of the images so that it can be processed easily by our model. We made a function that takes the image from dataset as an argument and returns a pre-processed image. Library that was used for this purpose was matplotlib. image imported as mpimg.

Various pre-processing techniques that were used for the dataset are:

1. **Cropping:** if we see a sample image it is evident that the scenery and the hood of the car are not contributing factors in feature extraction process. The hood and scenery acts **as** the noise in the image. They are removed using numpy slicing array function.

2. **Colour conversion:** we would convert the colour of the image in YUV format as suggested by the NVIDIA engineers as it provides better scaling of image as compared to RGB or grayscale image. In YUV format Y stands for brightness of image or luminance of image and UV looks for the colour component. Experts at NVIDIA says that this colour conversion is very efficient for training of neural network.

3. **Gaussian blur:** This is used to remove the noise from the image and to smoothen out the image. Less noise would make it easy for the neural network to extract features from the images.

4. **Resizing:** we decrease the size of the image as smaller images can be processed faster. We use the function cv2.resize().

5. **Familiar normalization:** this is a simple process of dividing each image by 255 as 255 is the maximum pixel intensity in a 1-D array.

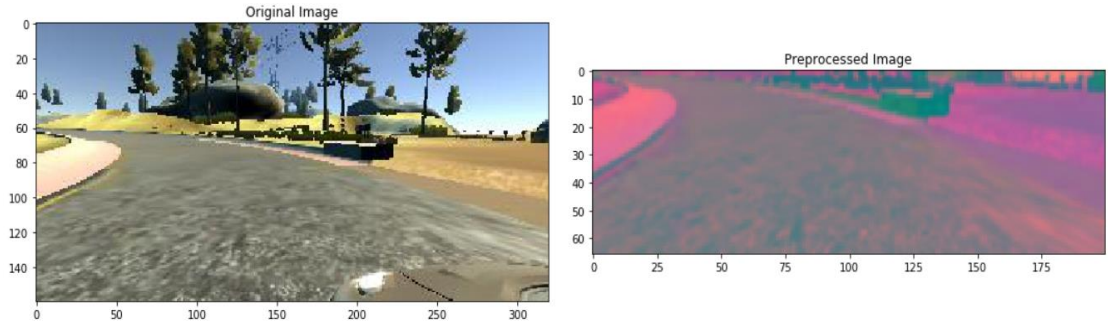


Fig 3.10: Comparison between original and pre-processed image

3.4.5 DEFINING NVIDIA MODEL

It is one of the most popular model for behavioural cloning. Developed in NVIDIA labs this model has been proved effective for several self-driving car models. First step of NVIDIA model is preprocessing of images which we have done already. Preprocessed images are passed in convolutional layer as input. The first layer of NVIDIA model is a convolutional layer that uses 25 filters along with 5x5 kernel

One thing to notice here is the subsample argument it defines the number of pixels' kernel will translate through vertically or horizontally. As this is the first convolutional layer input data has to be provided in it. Next convolutional layer has 36 filters with a kernel of 5x5.

Now, according to the NVIDIA model there are 3 more convolutional layers. One having 48 filters with a 5x5 kernel and 2 more layers with 64 filters and a 3x3 kernel.

Next layer is a flatten layer that takes the input from the convolutional layer and converts it into a 1-D array. After the flatten layer we have 3 dense layers of size 100, 50 10 respectively. Following line of codes are used for implementation of flatten and dense layers:

At the end of the NVIDIA model we will add a single dense layer with output that will give the steering angle as the output. This finishes the architecture of our model. Now, we will compile our model using the Adam optimizer.

Now, one of the major problems with the NVIDIA model is the problem of overfitting. To resolve this problem, we will be adding a dropout layer.

Dropout layer in CNN is specifically designed to overcome the problem of overfitting. Dropout layers set some fraction of nodes = 0 during each update. This helps the model to generalize the data as it is forced to use a variety of combination of nodes to learn from the same data. We will add 4 dropout layers one after the implementation of convolutional layers and one after each dense layer.
Following is the summary of the NVIDIA model:

```
[31]
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 31, 98, 24)        1824
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 47, 36)        21636
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 5, 22, 48)         43248
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 3, 20, 64)         27712
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 1, 18, 64)         36928
_________________________________________________________________
dropout_4 (Dropout)          (None, 1, 18, 64)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_9 (Dense)              (None, 100)               115300
_________________________________________________________________
dropout_5 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_10 (Dense)             (None, 50)                5050
_________________________________________________________________
dropout_6 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_11 (Dense)             (None, 10)                510
_________________________________________________________________
dropout_7 (Dropout)          (None, 10)                0
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 11
=================================================================
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
_________________________________________________________________
```

Fig 3.11: Summary of the NVIDIA model of convolutional neural networks

Next step in the process is training the model. Following line of code is used for training the model. After training the model, we will now plot the training loss and validation loss.

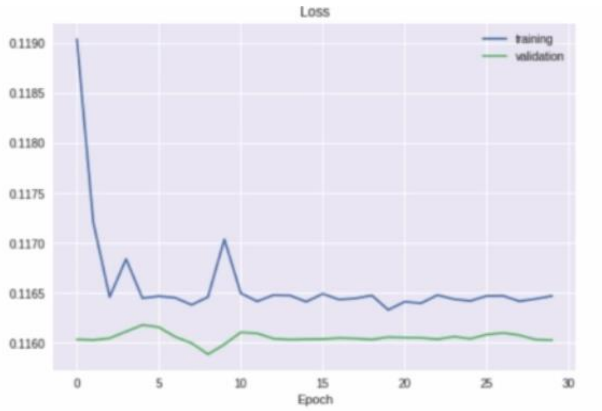


Fig 3.12: Plot comparison of training and validation losses

As we can see that the plots do not converge at all and the difference between the losses doesn't appear to reduce hence it is concluded that the model is not very efficient. One way to increase the efficiency of the model is by replacing the reLU function with the eLU function. reLU function gives value 0 as output for values less than zero with gradient also equal to 0. This creates a bias in the backpropagation and thus creates a problem in updating the weights in our model. Whereas the elu function doesn't have a gradient equal to zero for input values less than zero. Hence we shall replace the relu function with the eLU function.

After solving the problem of dying reLU function, if we plot the losses we can see that they converge hence proving that our model is effective but again, one more problem arises, that is the problem of overfitting. If we see the plot it is apparent that both the loss curves overlap hence denoting the problem of overfitting. This problem is again solved by adding more dropout layers.

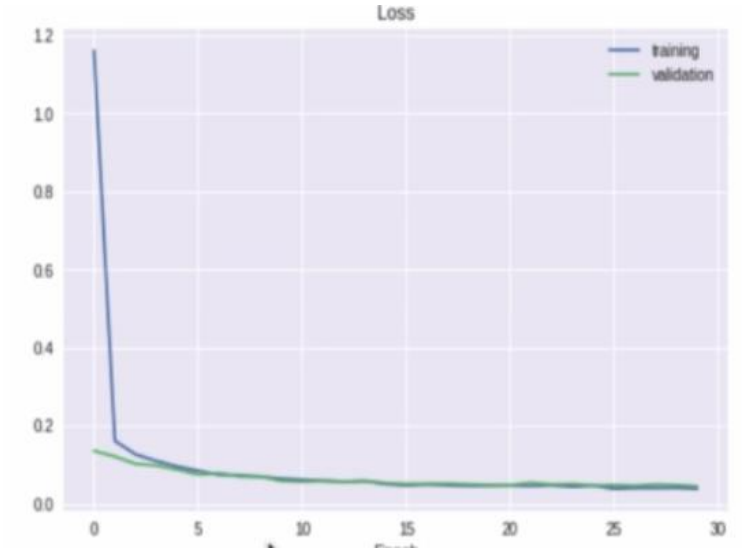


Fig 3.13: Plot of losses in training and validation denoting the overfitting of our model

3.4.6 Flask and Socket.io

One of the important tasks here is to establish a connection between the simulator and our model. This can be done with the help of Flask and socket.io. These are libraries of python that helps in establishing bi-directional client-server connection. A bi-directional client-server connection means that there is transfer of data from both the ends i.e. the client end and the server end. In our scenario the simulator will provide our model with the images and video feed of the track of autonomous mode and our model will calculate and provide the steering angle for the track. We are creating a WSGi server using socket.io so that any request can be sent from client to the web based application. It is not an adamant part of our project. It is not related to the deep learning aspects of our project. It just contains the code for creating a connection between our model and simulator.

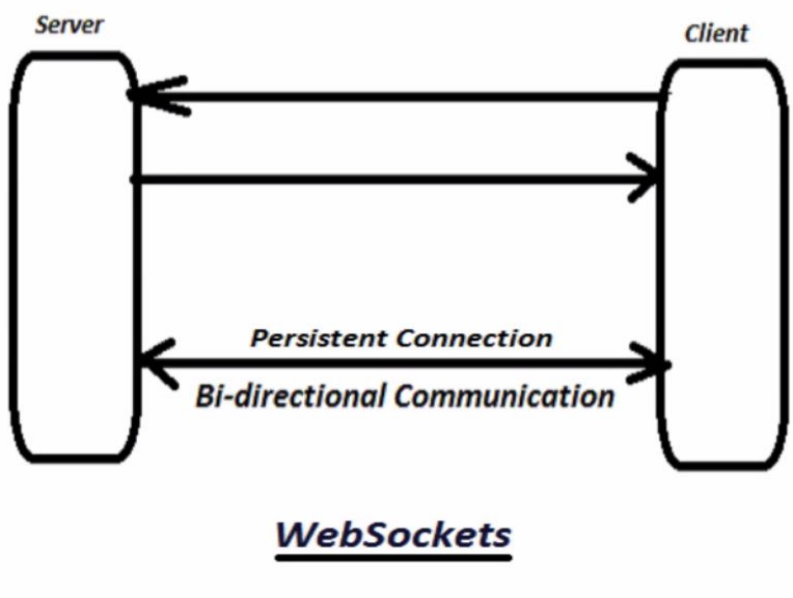


Fig 3.14: Web sockets

3.4.7 Generator-Augmentation technique

When we did our first trail run in autonomous mode it was evident that our model was not behaving perfectly on a general track. That problem can be traced to the lack of data. With only 1490 datasets in use, shortage of data is causing a problem for our model as it doesn't train that well. We can collect more data from the simulator or we can augment our current data. Augmentation of current data is a cheaper and efficient way of increasing the size of our dataset. Augmentation of data is done by designing an image data generator. Imgaug library is the library that has a large amount of image augmentation techniques that we can use in our image data generator. We install the imgaug library and import the augmenters as iaa.

In this project we are going to focus on 4 image augmentation techniques:

1. **ZOOMING:** this is pretty self-explanatory. It is evident that zooming 'in' the image is more helpful for us than zooming 'out' as it improves the feature extraction. We generate following function for image zooming.

```
Def zoom(image) :
    Zoom = iaa.Affine(scale(1, 1.3))          ##30% increase or 30% zoom
    Image = zoom_augment_image(image)
    return image
```

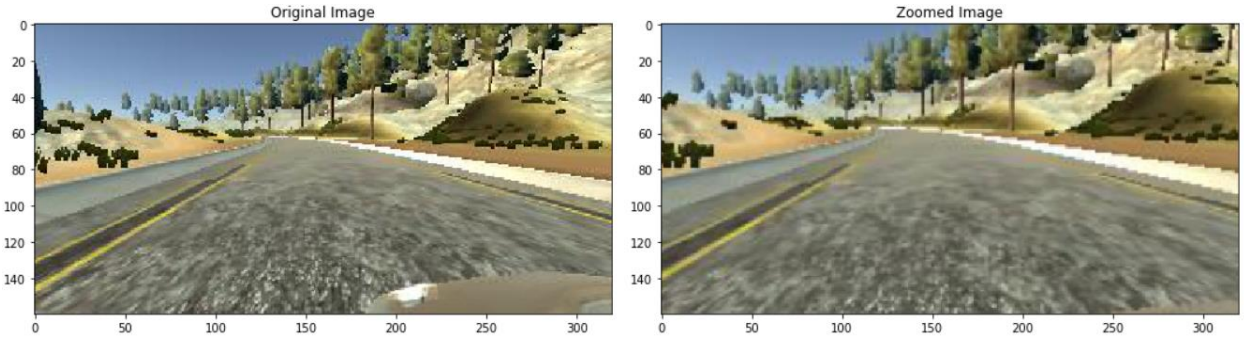
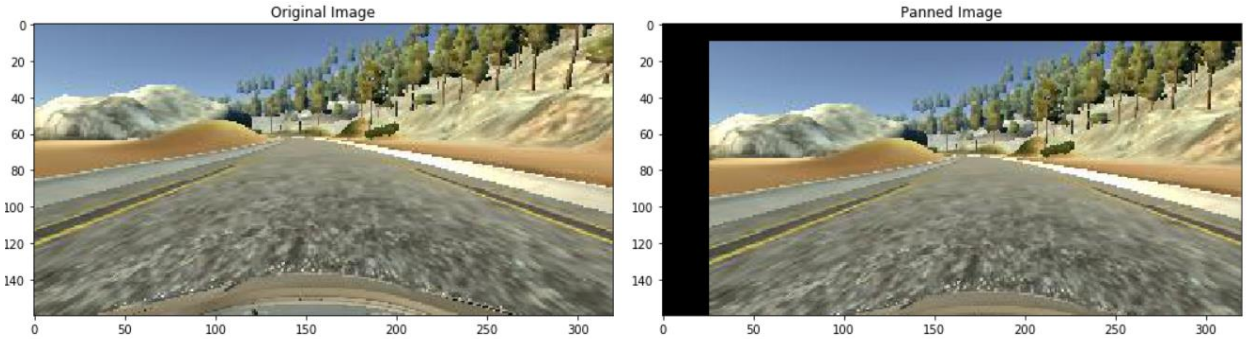


Fig 3.15: Original and Zoomed image

2. **Image Panning:** In this technique slight panning of image is done in horizontal or vertical direction. Following lines of code can be used for this process:

```
Def pan(image):
        Pan = iaa.Affine(translate_present = {"x" : (-0.1, 0.1), "y" : (-0.1, 0.1))
        pan.augment_image(image)
        return image
```



3. **Altering brightness of Image:** In this technique we change the brightness of the image. It multiplies all the pixel intensities with a specific value thus changing the brightness value. Following line of codes can be used to implement this

```
Def brightness(image):
    Brightness = iaa.multiply((0.2, 1.2))
    Image = brightness_augment_image(image)
    Return image
```

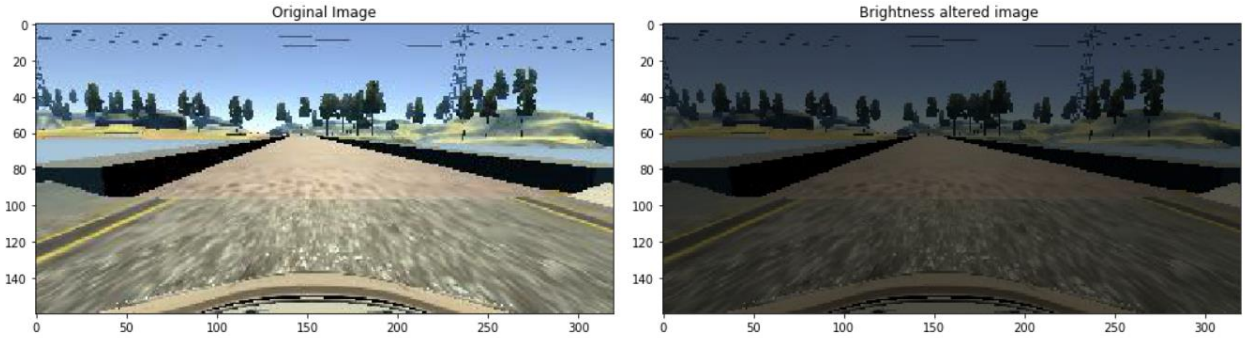


Fig 3.16 : Original and Altered Brightness Image

4. **Flipping of Image:** Images can be flipped in horizontal or vertical direction thus providing us with the opportunity of data augmentation. Random flipping can also provide us with a balanced data. Though vertical flipping of images is of no use to us. Horizontal flipping of images is done. Care should be done while horizontal flipping and steering angle must be changed to its negative value. Following lines of code were used for this process:

```
Def random_flip (image, steering_angle):
    Image = cv2.flip(image, 1)                    #1 for horizontal flipping
    Steering_angle = -steering_angle
    Return image, steering angle
```

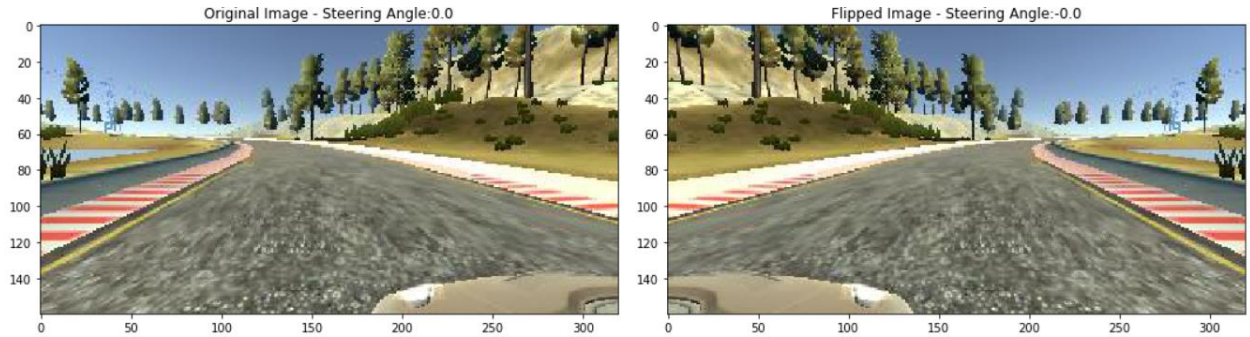


Fig 3.17 : Original Image and Flipped Image

Now we have to randomize these augmentation techniques over our data set as applying all the augmentation techniques on all the dataset images will reduce the overall variety of our dataset. Randomization causes overall generalization of our dataset. Thus we have randomized our augmentation function such that each augmentation function is applied 50% of the time. Following lines of code, we implemented for the above mentioned task.

```
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 = img_random_brightness(image)
    if np.random.rand() < 0.5:
           image, steering_angle = img_random_flip(image, steering_angle)

    return image, steering_angle
```

3.4.8 BATCH GENERATOR

We are creating an image data generator. Batch generator will take input data, create some augmented images and label these augmented images. When generator is called it will create small amount of augmented images at a time. For large datasets, we will use the fit generator() which creates the augmented images on the fly when needed it is done in batches. Batch generator shall only be used on training images i.e. augmentation of validation dataset is not required. Batch generator uses yield instead of return hence when the function is called again the values inside the function doesn't get reinitialized but they only take the previous value.