Behavioral Cloning

Behavioral Cloning Project

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Build, a convolution neural network in Keras that predicts steering angles from images
- Train and validate the model with a training and validation set
- Test that the model successfully drives around track one without leaving the road
- Summarize the results with a written report

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Files Submitted & Code Quality

1. Submission includes all required files and can be used to run the simulator in autonomous mode

My project includes the following files:

- model.py containing the script to create and train the model
- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- writeup_report.pdf summarizing the results

2. Submission includes functional code

Using the Udacity provided simulator and my drive.py file, the car can be driven autonomously around the track by executing `sh python drive.py model.h5`

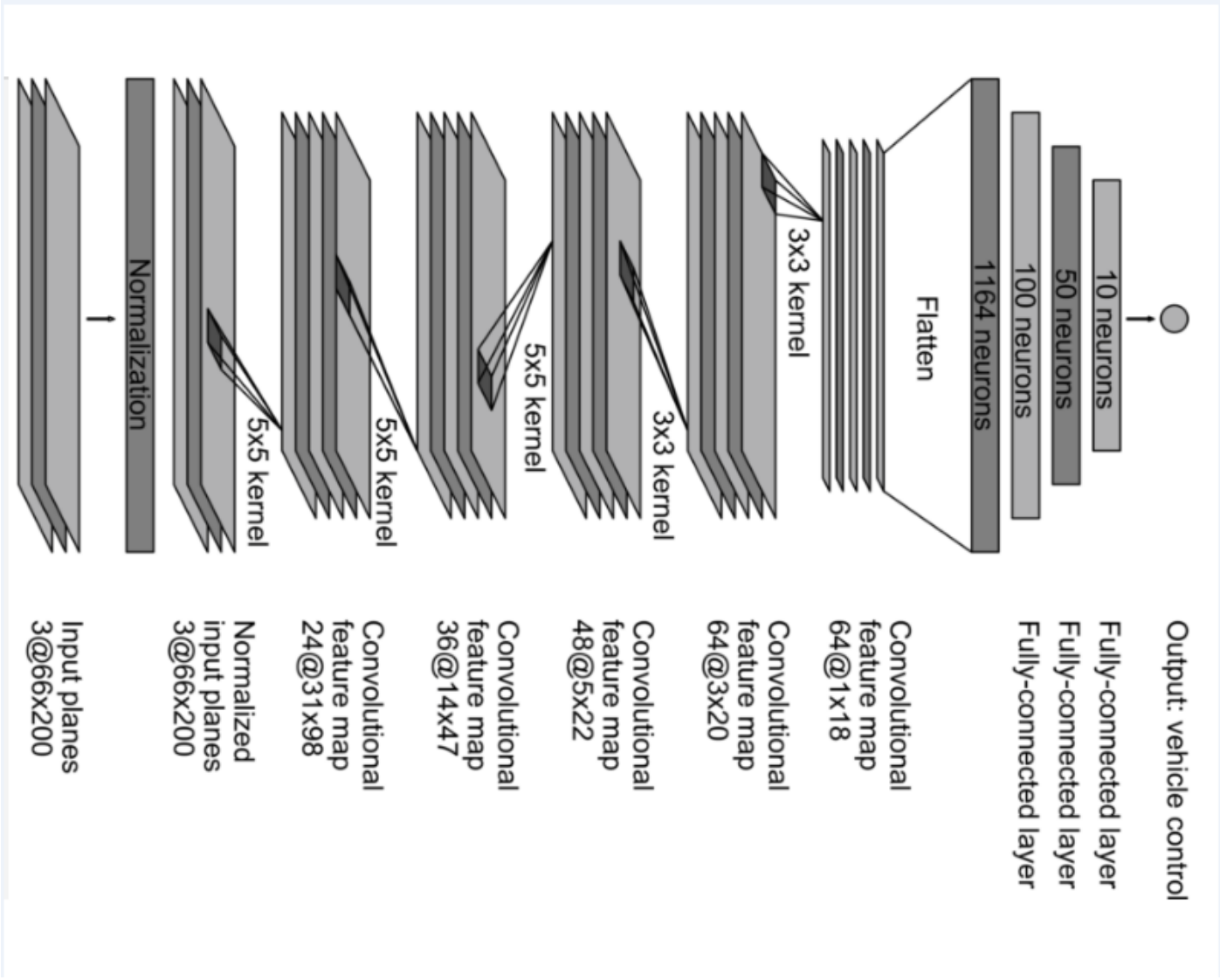
3. Submission code is usable and readable

The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

I am using the Nvidia model as my final submission. The details of this model is illustrated in Nvidia Blog, and in the course. The model consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. I modified this model by first cropping the images to remove the top of the image (by 70 pixels) which contained non-related road information and cropped the bottom of image (by 25 pixels) to remove the hood of the car. The data was normalized, zero-centered in range of -1 to 1. However I found two conflicting points from the Nvidai paper; first I found out that changing the activation from Relu to Elu did not improve the accuracy (did not decrease the MSE). And second in the blog they explain that they convert the colorspace from RGB to YUV. I did not find that helpful in my data. When I converted the image from RGB to YUV, the MSE increased. The model is shown in the following figures:



I had an attempt to use the keras implementation of VGG model to train a network for my data but the network did not converge. Considering that my main focus was to better train the model by using more

data collection and augmentation, I abandoned the experimentation of VGG network.

2. Attempts to reduce overfitting in the model

I did not find dropouts to help the accuracy so I did not use them. My effort to reduce overfitting was focused to collect different data sets. I collected from 2000 to 10000 images (center images, so with accompanying left and right images the size of data was three times these values), and trained model on each collection. Different strategies for data collection is described in section 4.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 106).

Number of Epoch, I tried three values, 5, 7 and 10. I found out that 7 is reasonable for this model. The model was converging after 7 Epochs, with validation accuracy close to training accuracy. Training for 5 Epochs was also showing reasonable training and validation accuracy, however I could see that model could benefit from couple more Epochs. More than 7 Epochs did not result any improvement.

Batch size was set to 32. I tried 64 and 128, which made the model converge faster, but I noticed that accuracy is suffering, so I decided to set the batch size to 32.

Data was split to 80% training and 20% validation, and loss function was set to MSE to minize the mean square error.

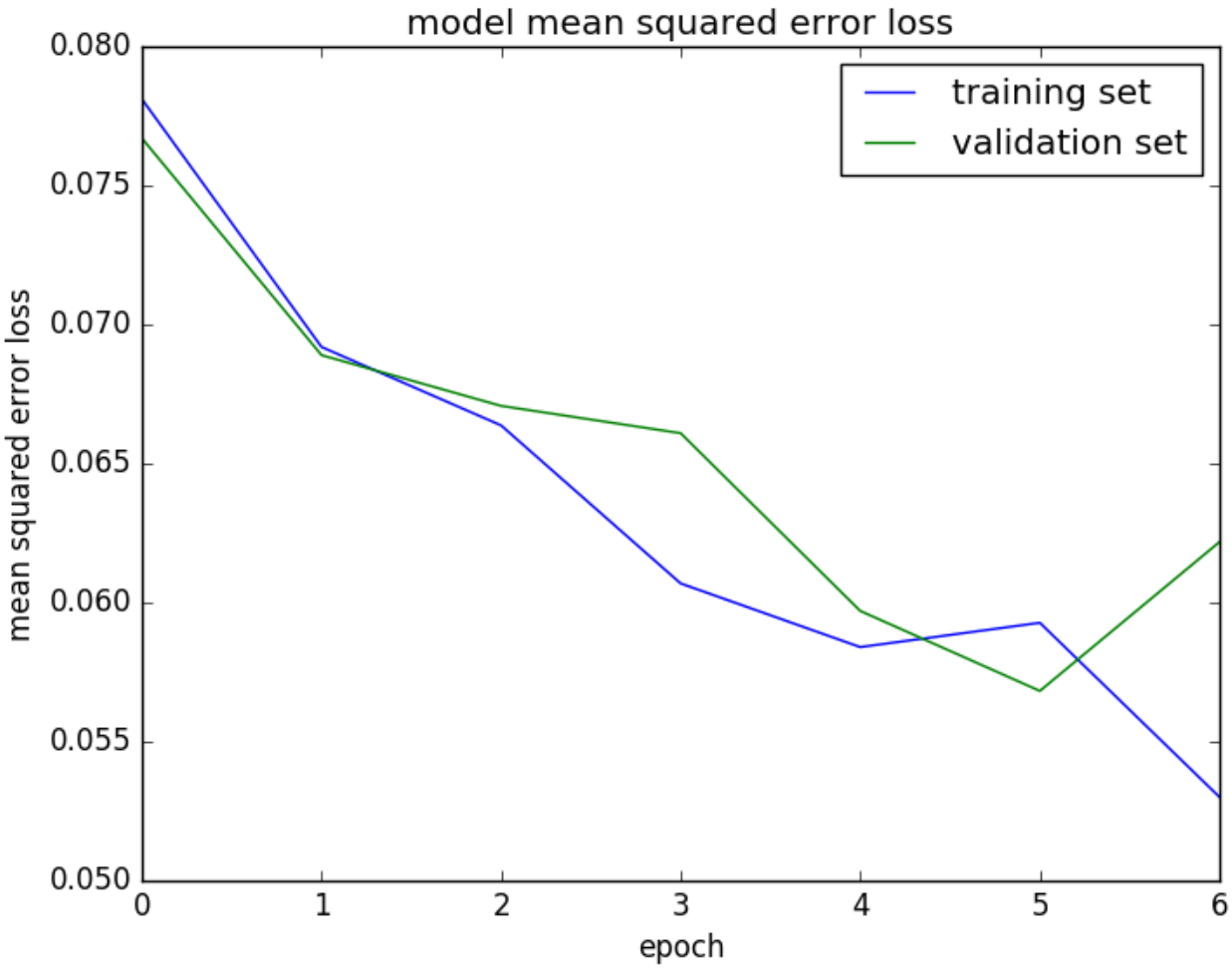
One of the early issues that I had was with dimension ordering of the Keras backend. After couple of days, I found out that I had to set the backend to tensorflow rather than Theano which was the Keras default value.

4. Appropriate training data

I performed several sets of training data collection using the simulator. At the beginning to make sure that the model is functioning properly, I drove the car for one lap on the center of the track using a relatively low and constant speed. After I ensured that the model is functional, I started to collect more data, first by driving a few laps around the track and keeping the in the center, and then driving the car with sharp turns and sudden changes in the speed, going to the sides of the road and driving back to center, and finally driving the car backward on the track. Thanks to AWS EC2, I was able to train the model fast using all these different combinations, although the time to transfer the training data to the AWS EC2 was significant, up to 20 minutes. For each data, I transferred back the generated model and tested it on the simulator. Interestingly, the second set of data alone with sharp turns and sudden changes of speed had the worst performance. The car was moving off track not long from the start line. The best performance was from the combining all training set into one which the car was driven several laps on the center of the track, driven backward and driven simulating driver mistakes. The model generated from this set was comparable to the one created by the training data released by Udacity and was used for final training and submission. None of the models trained with these data were successful

in the second track, and I did not collect data from the second track for training. The following figure show the accuracy for data collected using the simulator. The training and validation accuracy are loss: 0.0222 - val*loss: 0.0226 for collected data which are very similar to training and validation accuracy for Udacity released data, loss: 0.0299 - val*loss: 0.0293.

Below is accuracy for the model trained only on the data from sharp turns and driver mistake simulation. As it can be seen, this data by itself not only had higher MSE, but it did not converge.



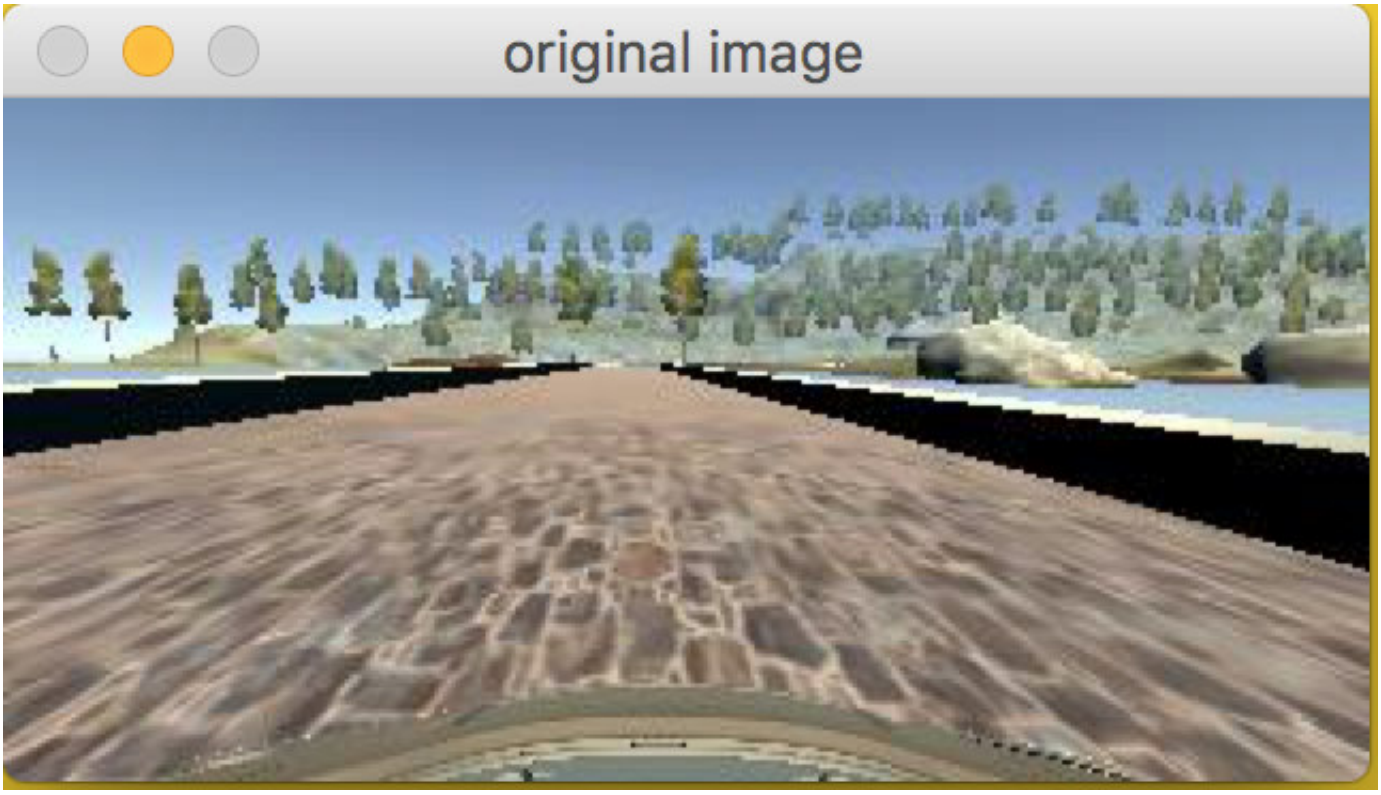
Model Architecture and Training Strategy

1. Solution Design Approach and Final Model

Through a comprehensive literature survey, I found out that the Nvidia model is one of the better candidates for this exercise. Beside having all the components and the being tested for self-driving, its input/output structure matched well with the project specifications. As it was described during the course, the choice of network architecture is mainly empirical. It must contain a convolutional layer to extract feature and it must contain the fully connected layers to function as a controller. The choice of kernels and depths in convolutional layers should gradually encode more image features into more condensed information that with the a flattening layer could be classified into output values.

2. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving:



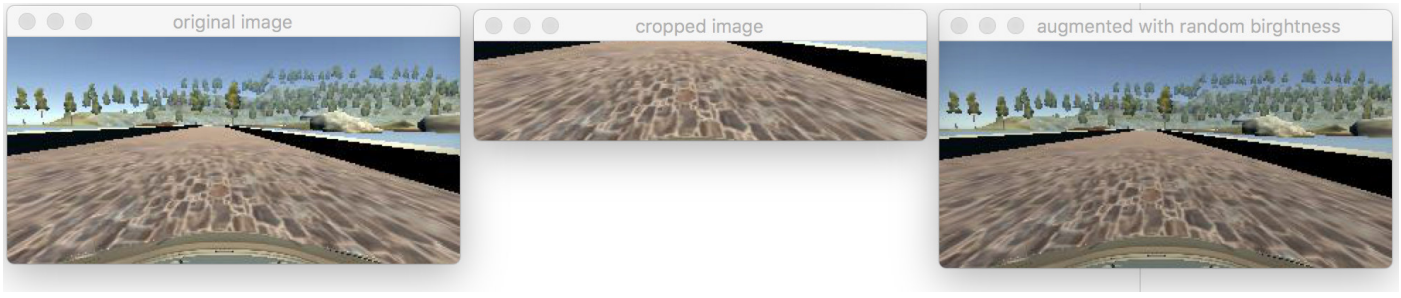
I then recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn to perform well if the driver makes mistakes. An image showing this recovery is shown below:

Finally to collect more data, I recorded the vehicle driving backward on the track trying to keep the car in the center of the lane. Images below are showing how I used the shoulder area to make a u-turn and then drive the car backward:


self_driving_car_nanodegree_program
ANGLE
0.00°
RECORD
0.01
MPH
MENU

3. Preprocessing and Data Augmentation

After collecting around 10000 images (counting center, left and right = 30000 images), I performed preprocessing and data augmentation. Preprocessing included normalization and cropping the image. A sample image along with the cropped image is shown below.



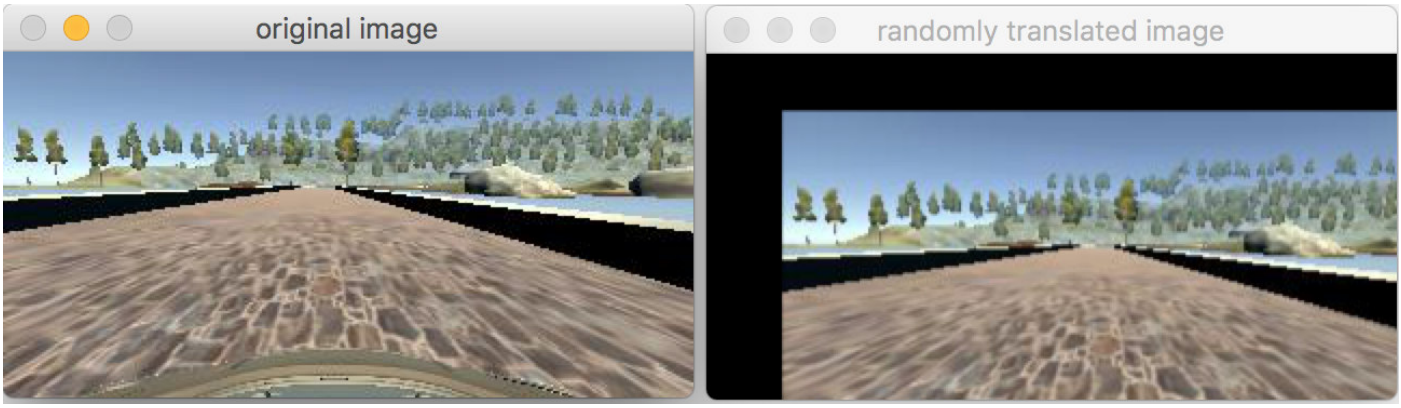
To further assist the model learning none-zero steering angles, I randomly dropped 80% of the zero steering angles. This helped with both the speed and the network weights.

Data augmentation included several steps:

1- I flipped the center image and inverted the steering angle. This helps with balancing the data

2- I randomly added brightness to the center image. This was performed in HSV colorspace as RGB is not suitable to augment the image brightness.

3- For each of left and right images, I randomly translated the image and adjusted the steering angle accordingly. The amount of steering angle adjustment was a function of number of pixel shift, however I found out model is not very sensitive to this parameter as long as the value is small. I scaled the pixel shift by 0.002.

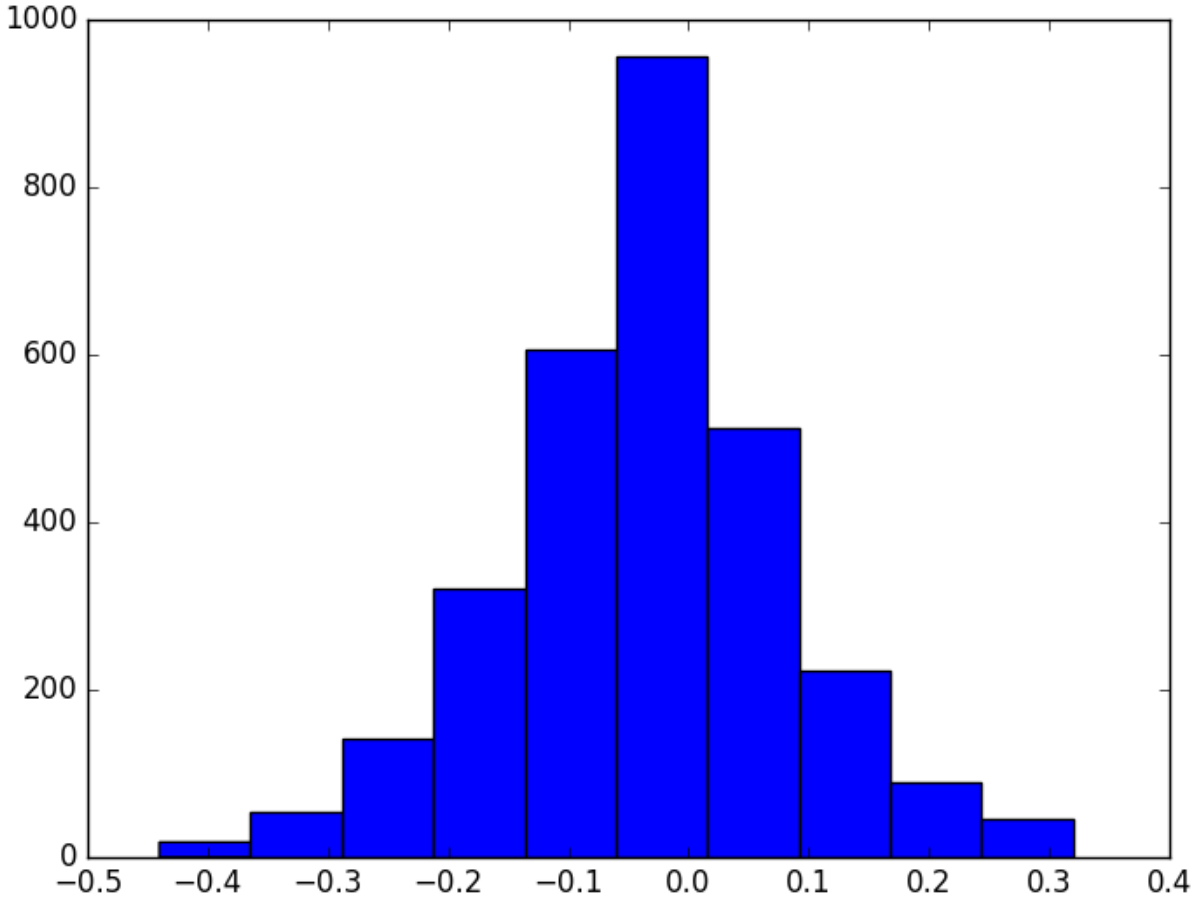


4- Adding left and right images require the steering angle correction. Without having a proper 3D scene information to find the target point and the distance between cameras it is not possible to computer accurate correction. Therefore empirically I chose value 0.3 to add and subtract from steering angles to adjust the left and right images.

After the collection and data augmentation process, I had around 70000 number of data points.

4. Training and Running the Model on Simulator

Training was performed on AWS EC2 for 80% of the collected and augmented data (80% x 700000 = 56000). Then model was validated on the rest of data. The accuracy of the training and validation was loss: 0.0222 - val_loss: 0.0226, which is comparable to what I obtained from data released by Udacity. The vehicle could drive the in the first track without leaving it. The steering angles histogram from driving the car in autonomous mode in first track is shown below. The histogram distribution is relatively normal with zero mean.



Data smoothing of the steering angles in real-time using butterworth filter was suggested in one of the posts in the medium platform. As it can be seen from the submitted video, I believe this technique would have helped the jitter artifact. However, I could not gather enough information to implement this filter. I tried to best guess some values, for example to set the sampling frequency to 10 Hz, but the car did not move.