#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

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

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.md: summarizing the results

####2. Submission includes functional code Using the Udacity provided simulator and drive.py file, the car can be driven autonomously around the track by executing

python drive.py model.h5

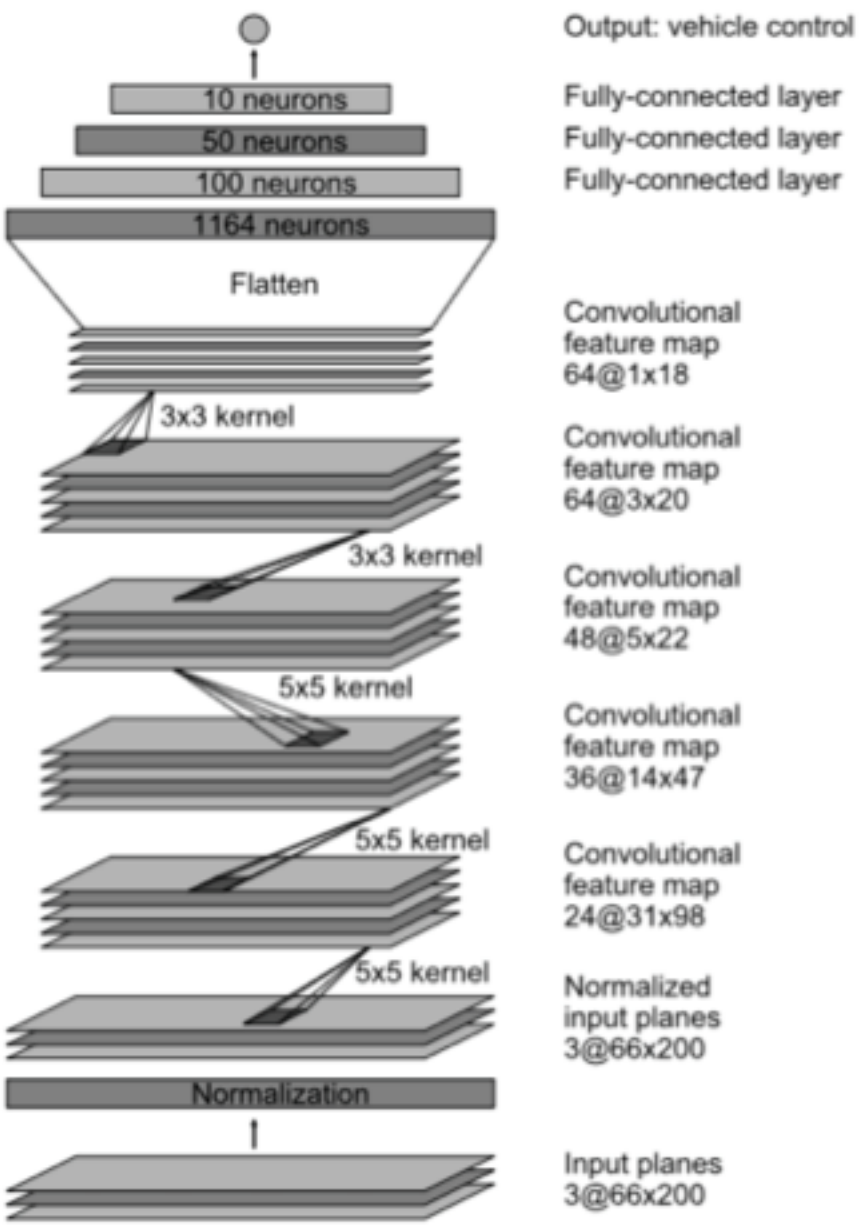
####3. Submission code is usable and readable

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

My model is based on Nvidia model. Image original size is 160 * 320 * 3 and cropped top 70*320*3 and bottom 20*160*3 pixels. Then resized cropped images to be the size of 66*200*3 in order to be consistent to Nvidia model. Resize image is normalized to between -1 and 1. Below is the Nvidia architecture:



Here is the model summary from Keras:

model = Sequential()

```
#resize image otherwise won't be loaded by drive.py
model.add(Cropping2D(cropping=((60, 25), (0, 0)),input_shape=(160, 320, 3)))

# Resize the data to become 66*200*3 format
model.add(Lambda(resize_comma))

# Normalize image to be within (-1, 1)
model.add(Lambda(lambda x: x/255.0 - 1.0))

# Add three 5x5 convolution layers (output depth 24, 36, and 48), each with 2x2 stride
# In all three layers, regulation (0.001) is applied
# Nonlinear active function ELU is applied to add nonlinearity
model.add(Convolution2D(24, (5, 5), strides=(2, 2), padding='valid',
kernel_regularizer=l2(0.001)))
model.add(ELU())
model.add(Convolution2D(36, (5, 5), strides=(2, 2), padding='valid',
kernel_regularizer=l2(0.001)))
model.add(ELU())
model.add(Convolution2D(48, (5, 5), strides=(2, 2), padding='valid',
kernel_regularizer=l2(0.001)))
model.add(ELU())

# Add dropout (0.5) to further reduce overfitting.
model.add(Dropout(0.50))

# Add two 3x3 convolution layers (output depth 64, and 64)
# Again, ELU function is applied to increase nonlinearity
# Again, regulation (0.001) is applied
model.add(Convolution2D(64, (3, 3), padding='valid', kernel_regularizer=l2(0.001)))
model.add(ELU())
model.add(Convolution2D(64, (3, 3), padding='valid', kernel_regularizer=l2(0.001)))
model.add(ELU())

# Dropout (0.5) is applied.
model.add(Dropout(0.50))

# Add a flatten layer
model.add(Flatten())

# Add three fully connected layers (depth 100, 50, 10), ELU activation (and dropouts)
model.add(Dense(100, kernel_regularizer=l2(0.001)))
model.add(ELU())
model.add(Dense(50, kernel_regularizer=l2(0.001)))
model.add(ELU())
model.add(Dense(10, kernel_regularizer=l2(0.001)))
model.add(ELU())
```

```
# Add a fully connected output layer
model.add(Dense(1))

#train model  by applying adam optimizer.
model.compile(loss = 'mse', optimizer='adam')
batch_size = 128
model.fit_generator(train_generator, steps_per_epoch= len(train_x)/batch_size,
                    validation_data=validation_generator, validation_steps=len(val_x)/batch_size,
                    epochs=12)
```

####2. Attempts to reduce overfitting in the model

The model contains dropout layers and regulation in order to reduce overfitting.

The model was trained and validated on different data sets to ensure that the model was not overfitting. The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

####3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually

####4. Appropriate training data

I created new images and combined them with Udacity data. This is one approach to augment data. After that, I used a combination of center lane driving, recovering from the left and right sides of the road. I tried different angle correction on left and right images from 0.08 to 0.27. 0.25 provides the best performance.

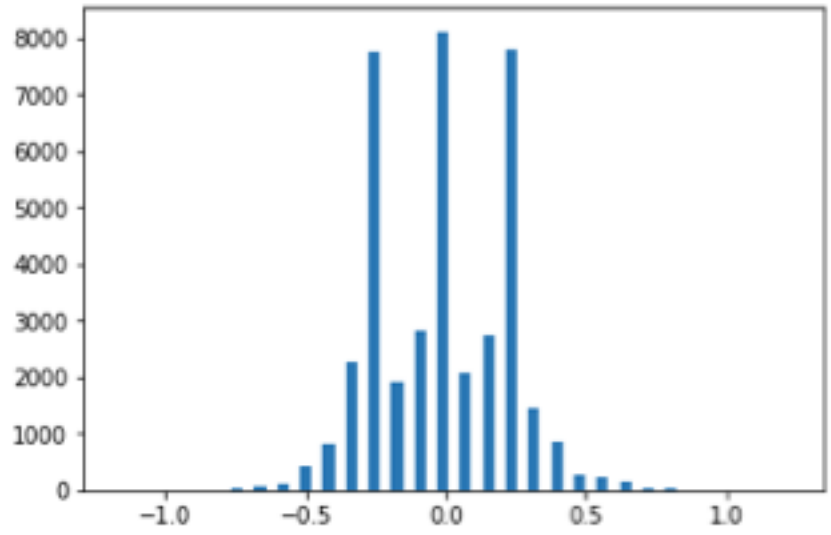
###Model Architecture and Training Strategy

####1. Solution Design Approach

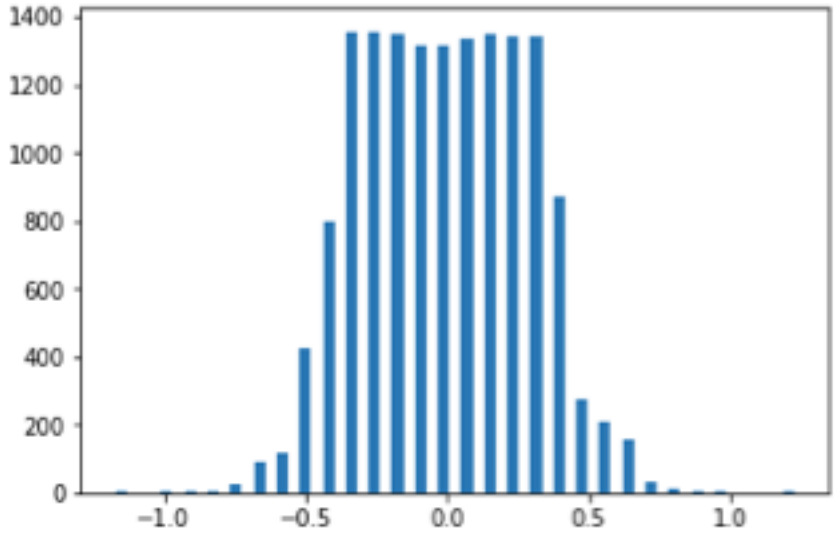
The key thing that I learnt from this project is the importance of data. More specifically, the data balance is important. In the very first several trials, I didn't apply any specific processing to deal with the

unbalance input data. As the result, even though model converged abnormally fast, as a result, in the autonomous simulation, the car constantly steered off the road and most of the time, it steered toward to the right side of the road. The initial steering angles were heavily dominated by both the center and the right side drivings. Then I applied statistically analysis and randomly removed the data sets whose samples were larger than the average. I compared the histogram before and after removing redundant measurements:

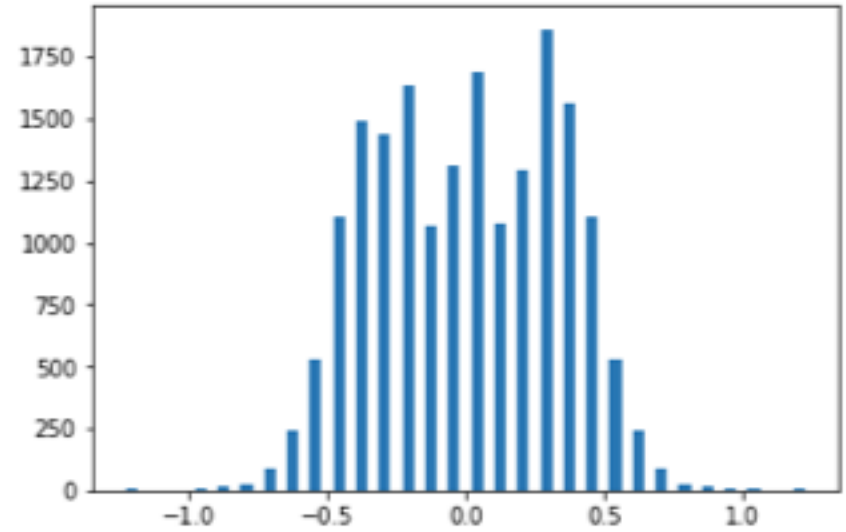
Before:



After:



As we can see, the measurements beyond +-0.35 are still not well balanced. In order to overcome this unbalance, I flipped images in this particular range. Then below is the final histogram:



My understanding toward the data balancing is giving measurements fair opportunities to be trained in the model, or in other words, at least symmetric around 0 measurement. Then the samples provide relatively the same left and right measurements to the training model. The model can learn the behavior of left and right steering equally. Otherwise, the car will always be steering to one side as I had at the very beginning. The other thing I want to emphasize here is the above

histogram filtering is completely data driven. In the other words, we should always check on the data quality and based on it to determine any adjustment factors, which will be more efficient than just randomly trying parameters. The necessary data analysis is very important. Understanding data properly is the first key step!

In oder to make the data set represent reality better and help the model learning, I applied random bigness and random shading to better augment data sets. Additionally, in the Nvidia model, it requires YUV scale instead of RGB. Images were converted to YUV.

The pipe is as shown below:

1. Load data sets

2. Check histogram

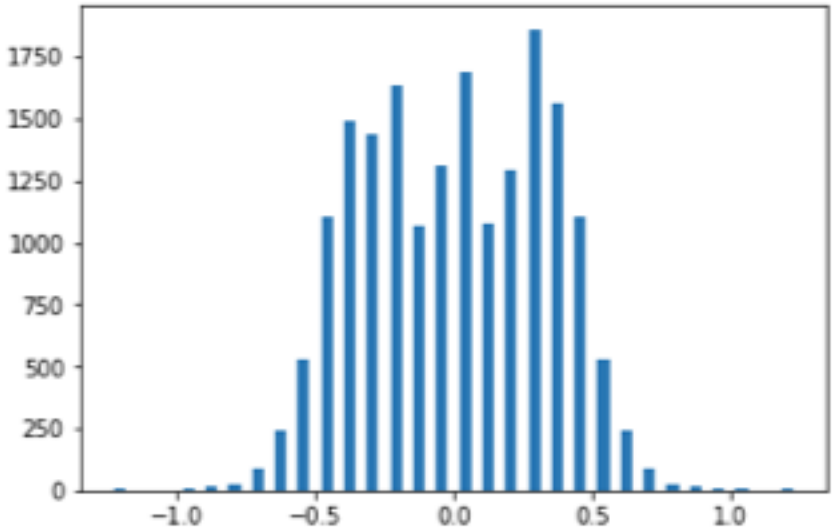
3. Filter redundant/strong based measurements

4. Convert images from RGB to YUV

5. Apply random bigness and shading processing

6. Crop images and resize to 66*200*3

7. Create generator and generator loads patches.



8. Nvidia model training

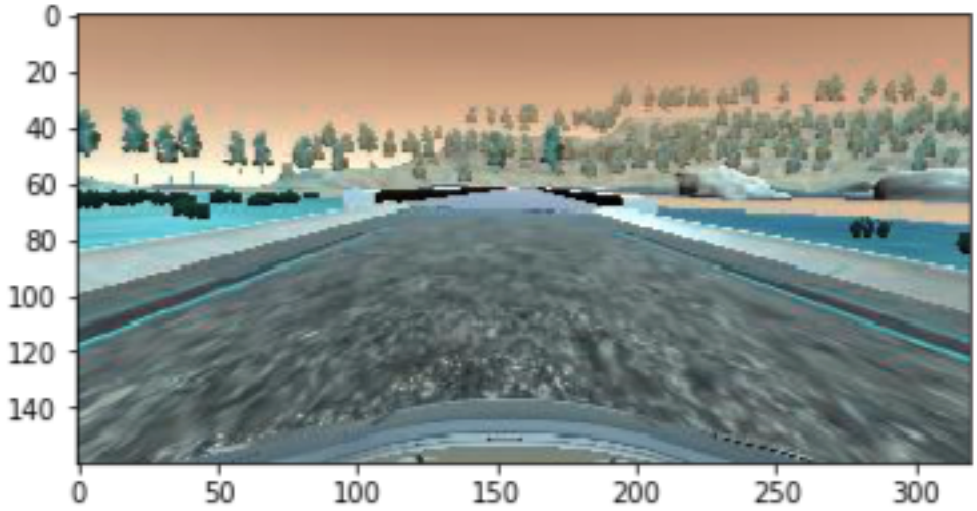
occasionally tried 128, it worked! So I think it is the same theory here: apply proper amount data as always!

####2. Final Model Architecture

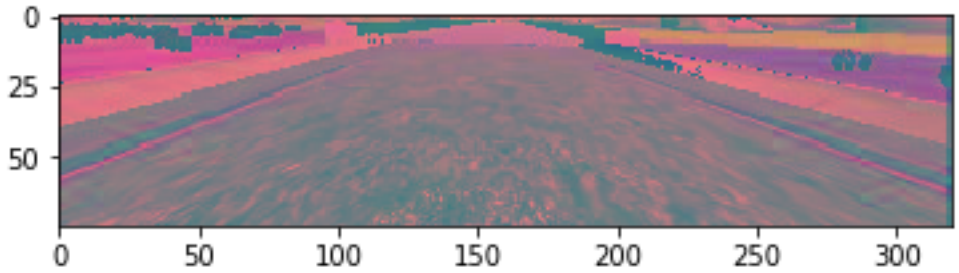
The final model is Nvidia. I added regulation and two dropout layers (0.5).

####3. Creation of the Training Set & Training Process

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



As I mentioned above, I applied random brightness, shading processing, scale conversion and cropping:



I randomly shuffled the data set and put 10% of the data into a validation set.

I used this training data for training the model. The validation set helped determine if the model was over or under fitting. I used an adam optimizer so that manually training the learning rate wasn't necessary. The validation error is always close to training loss, so overfitting overcomes. The finally autonomous simulation works! The car stays in the driving region for a complete lap!