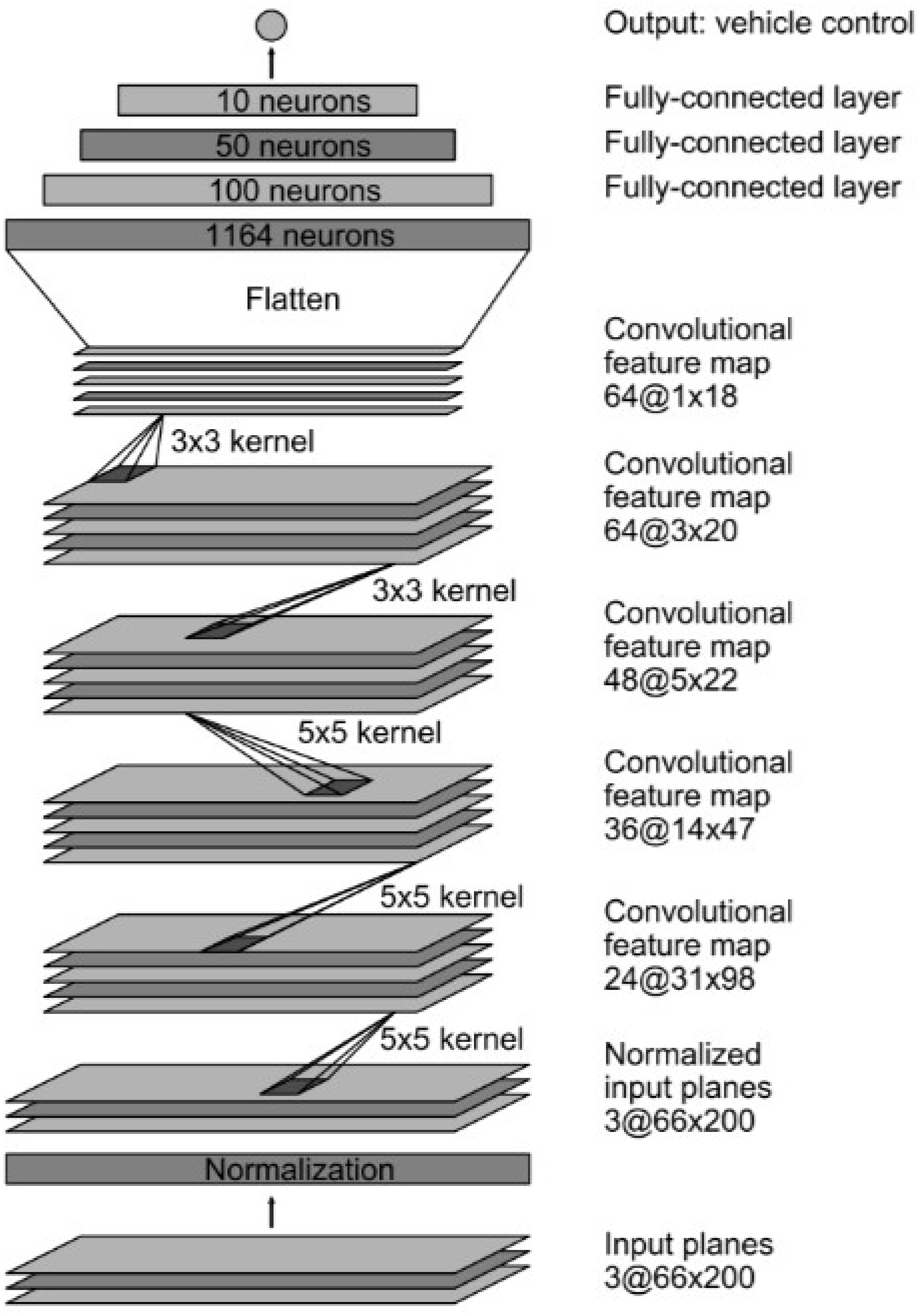
#**Behavioral Cloning**

##Writeup Template

###Model Architecture and Training Strategy

####1.

My model consists of a convolution neural network as shown below.



The model includes RELU layers to introduce nonlinearity (model.py, line 139 -158), and the data is normalized in the model using a Keras lambda layer (model.py line 133).

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

The model contains regularization layers in order to reduce overfitting (model.py lines 152-158).

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 (model.py line 160).

####4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of solely center lane driving provided by Udacity, without recovery data from the left and right sides of the road.

For details about how I created the training data, see the next section.

###Model Architecture and Training Strategy

####1. Solution Design Approach

The overall strategy for deriving a model architecture was to document myself about this kind of approach, so I came across an interesting article suggested also in a few Udacity posts and named " End to End Learning for Self Driving Cars ". Ever since I retraced my model to those cited in the paper, it was that kind of approach I was searching for in order to solve the problem.

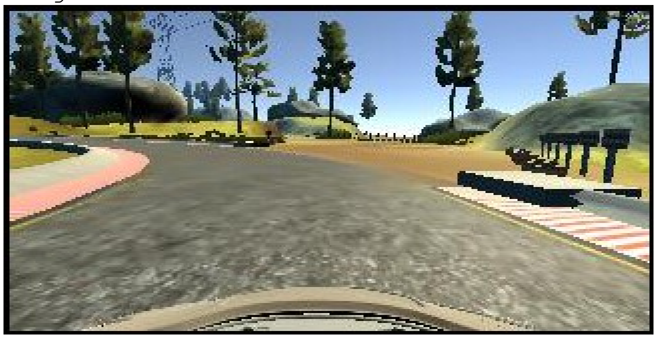
I thought this model might be appropriate because worked on an a real road in the real world, and therefore because it was a good alternative to the classical LeNet. Further the model seemed extremely robust for these type of application.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set (model.py line 123).I found that my first model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting.

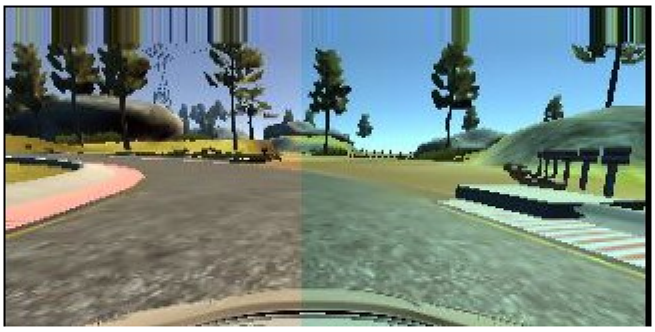
Non linearity was introduced at first with some activation functions of various types, but my final choice consisted in a RELU function for every layer on the Net. Before that all my attempts were ineffective, the car drove badly, struggling to stay in the middle of the track.

Knowing that the images produced by the simulator were 160x320, subsequently they were cropped to the dimension of 66x200 introducing a Cropping Layer in order to focus the model massively on the road excluding useless details as suggested in the project intro and shown in article before cited (model.py line 136).

Another method to minimize model's tendency to overfit was jittering the images by means of random distortion, which consisted of a randomized brightness adjustment as well as randomized shadow and randomized horizon shift to mimic the conditions of a hilly environment or a cloudy day. Below an example of distorted image produced.(model.py lines 36-64) original



distorted



After that I tried to introduce dropout layers with different keep probability but unsuccessfully, since the car punctually finished off the road after the bridge. The turning point was when I used for each fully connected layer a L2 regularization equals to 0.00001 (model.py lines 152 - 158)

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

####2. Final Model Architecture

The final model architecture (model.py lines 130-160) consisted of a convolution neural network with the following layers and layer sizes ...

Here is a visualization of the architecture

Layer	Function	Regularization
Input layer (160 x 320 x3)		
Lambda : Normalization		
Cropping: (66x200x3)		
Convolution: 5x5 kernel, 2x2 stride, 24 depth	Activation: RELU	
Convolution: 5x5 kernel, 2x2 stride, 36 depth	Activation: RELU	
Convolution: 5x5 kernel, 2x2 stride, 48 depth	Activation: RELU	
Convolution: 3x3 kernel, 64 depth	Activation: RELU	
Convolution: 3x3 kernel, 64 depth	Activation: RELU	
Flatten		
Fully Connected: 100 depth	Activation: RELU	L2 regularization, L2 = 0.00001
Fully Connected: 50 depth	Activation: RELU	L2 regularization, L2 = 0.00001
Fully Connected: 10 depth	Activation: RELU	L2 regularization, L2 = 0.00001
Fully Connected: 1 depth	Activation: RELU	L2 regularization, L2 = 0.00001

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

Due to lack of time and without a GPU I tried since from the beginning to train the Net with just the samples data provided by Udacity. So I didn't recorded any further data around the track by means of the simulator.

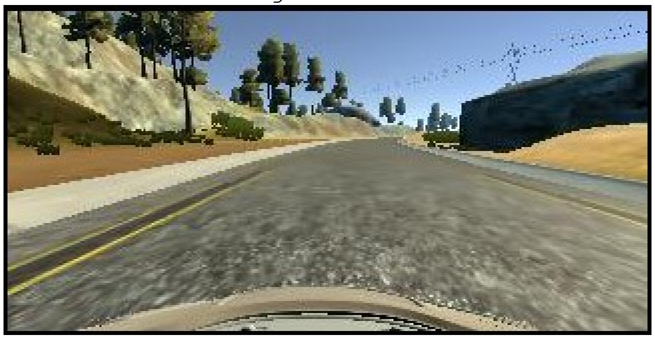
Preprocessing the images consisted to apply a Gaussian blur filter in order to reduce the noise and converts the images in BGR to RGB seeing the simulator works with this type of images.
(model.py lines 29-34)

To augment the data set, I also flipped images and angles thinking that this would reduced bias toward low and zero steering angles. Furthermore another benefit that came with flipped images was about balancing out higher angles in each direction so that neither left or right turning angles become in somehow overrepresented. To do that I applied a threshold that if overcame subsequently another image was produced, which was the exact mirror of the original one and consequently the associated angle was inverted.

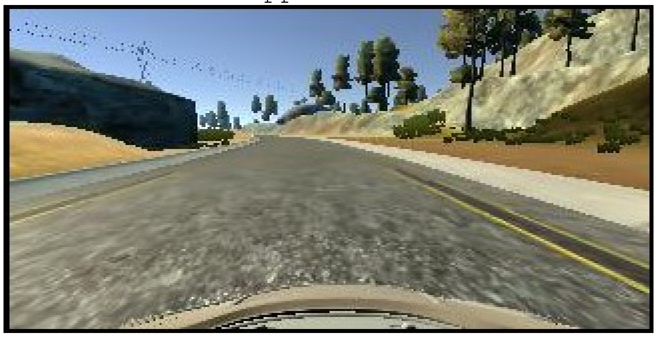
To sum up, if the magnitude angle is higher than 0.3, which was to me a significant angle and worth of being taken into account, the image was flipped.

For example, here is a couple of images that have been flipped:

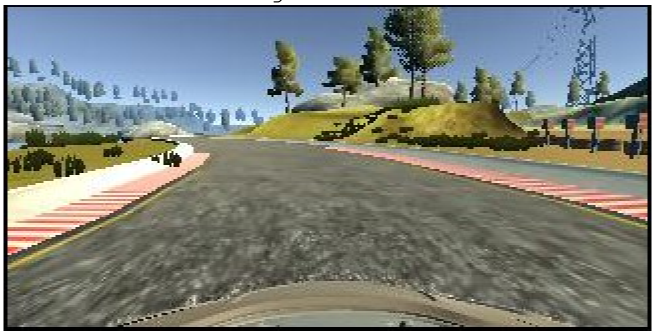
0.3488 original



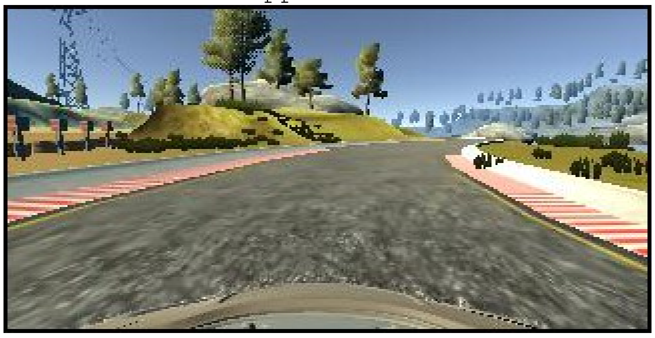
- 0.3488 flipped



0.3523 original



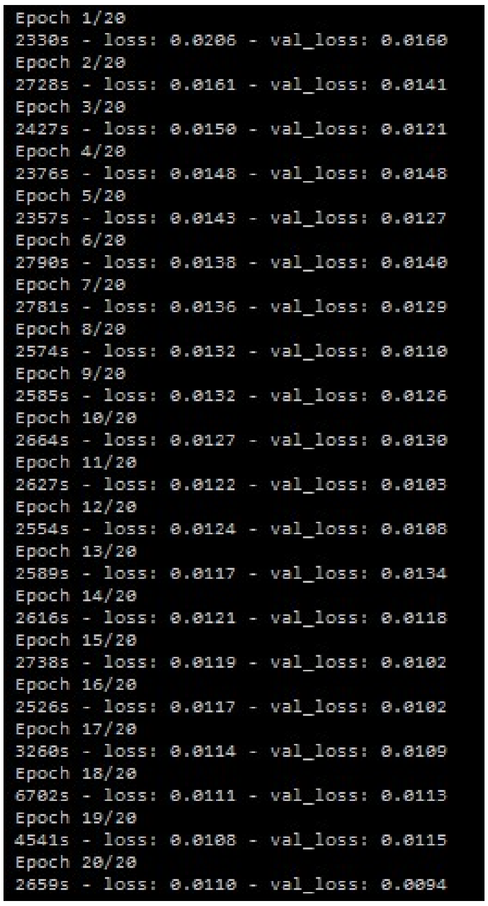
- 0.3523 flipped



After the collection process, I had 7634 number of data points.

I finally randomly shuffled the data set and put 20% of the data into a validation set. (model.py line 23)

I used this training data for training the model. The validation set helped determine if the model was over or under fitting. The ideal number of epochs was 10 as evidenced by the image below. When the validation loss became minor than the training loss the car sometimes lost the control of the road with huge speed, so I chose the model saved at ten epoch, which represents the best fit between the two parameters.



```
Epoch 1/20
2330s - loss: 0.0206 - val_loss: 0.0160
Epoch 2/20
2728s - loss: 0.0161 - val_loss: 0.0141
Epoch 3/20
2427s - loss: 0.0150 - val_loss: 0.0121
Epoch 4/20
2376s - loss: 0.0148 - val_loss: 0.0148
Epoch 5/20
2357s - loss: 0.0143 - val_loss: 0.0127
Epoch 6/20
2790s - loss: 0.0138 - val_loss: 0.0140
Epoch 7/20
2781s - loss: 0.0136 - val_loss: 0.0129
Epoch 8/20
2574s - loss: 0.0132 - val_loss: 0.0110
Epoch 9/20
2585s - loss: 0.0132 - val_loss: 0.0126
Epoch 10/20
2664s - loss: 0.0127 - val_loss: 0.0130
Epoch 11/20
2627s - loss: 0.0122 - val_loss: 0.0103
Epoch 12/20
2554s - loss: 0.0124 - val_loss: 0.0108
Epoch 13/20
2589s - loss: 0.0117 - val_loss: 0.0134
Epoch 14/20
2616s - loss: 0.0121 - val_loss: 0.0118
Epoch 15/20
2738s - loss: 0.0119 - val_loss: 0.0102
Epoch 16/20
2526s - loss: 0.0117 - val_loss: 0.0102
Epoch 17/20
3260s - loss: 0.0114 - val_loss: 0.0109
Epoch 18/20
6702s - loss: 0.0111 - val_loss: 0.0113
Epoch 19/20
4541s - loss: 0.0108 - val_loss: 0.0115
Epoch 20/20
2659s - loss: 0.0110 - val_loss: 0.0094
```

I used an adam optimizer so that manually training the learning rate wasn't necessary.