#**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.md or 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

```
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. Solution Design Approach

My first step was to use a convolution neural network model similar to the nvidia's CNNs I thought this model might be appropriate because they have used it to map the raw pixels from a front-facing camera to the steering commands for a self-driving car which similar to what I was doing.

I split my image and steering angle data into a training and validation set using sklearn. The first model always fell off the track and drove into valley where looks like a road but it's not. And the 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.

To combat the overfitting, I modified the model and added a Dropout layer. Then I tested Dropout parameter from 0.9 to 0.5. And 0.7 has the best accuracy.

However, while the accuracy was pretty good, the vehicle still drove into the river, that bothered me for a long time, and finally, I found out it's because of opencv read iamges as bgr and I fixed it.

At the end of the process, the vehicle is able to drive autonomously around the track and can drive back if it has fell off the track.

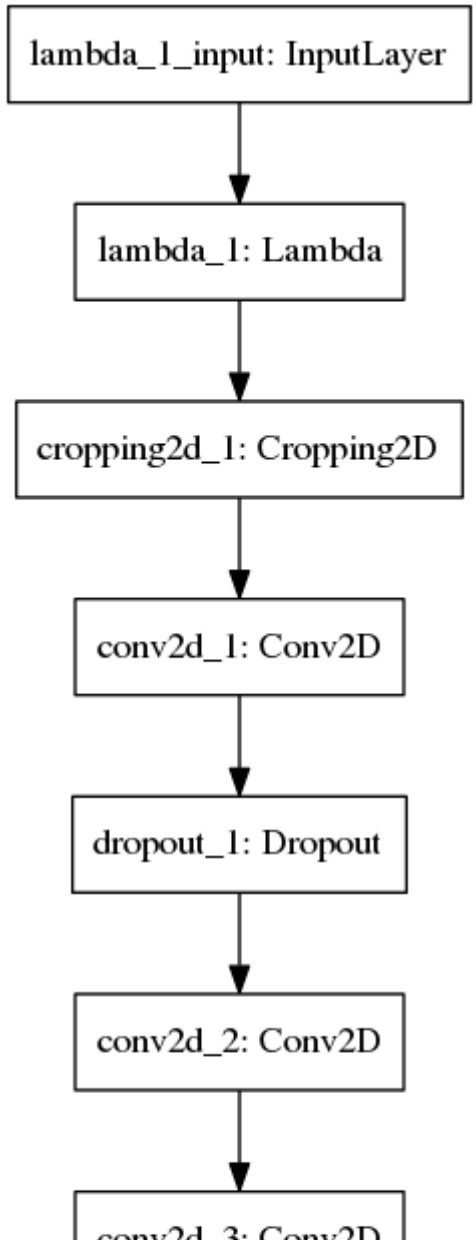
####2. Final Model Architecture

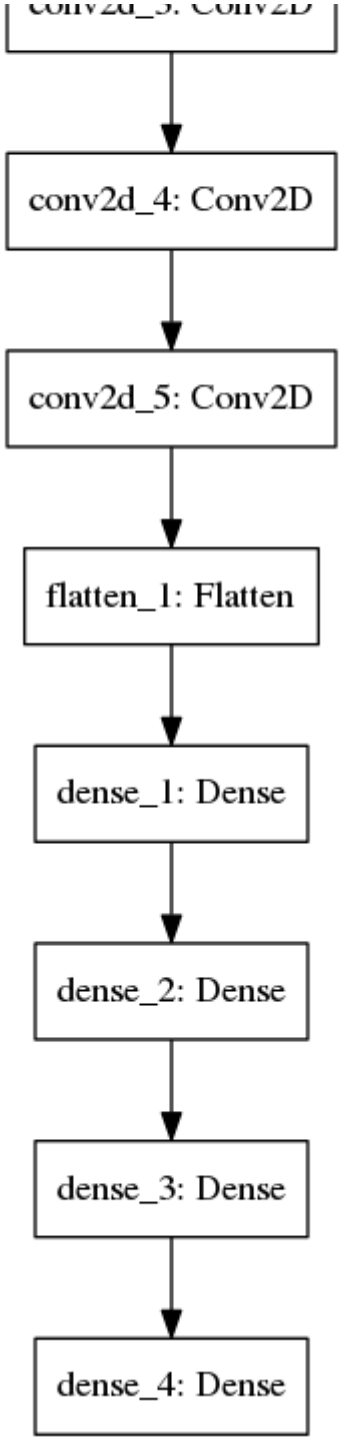
The final model architecture (model.py lines 72-85) consisted of of the following layers:

Layer	Description

Input	160x320x3 RGB image
Cropping2D	crop top 55px, botton 20px
Convolution 24x5x5	2x2 stride, VALID padding
RELU	
Dropout	avoid overfitting
Convolution 36x5x5	2x2 stride, VALID padding
RELU	
Convolution 48x5x5	2x2 stride, VALID padding
RELU	
Convolution 64x3x3	1x1 stride, VALID padding
RELU	
Convolution 64x3x3	1x1 stride, VALID padding
RELU	
Flatten	
densely-connected	outputs 100
densely-connected	outputs 50
densely-connected	outputs 10
densely-connected	outputs 1

Here is a visualization of the architecture

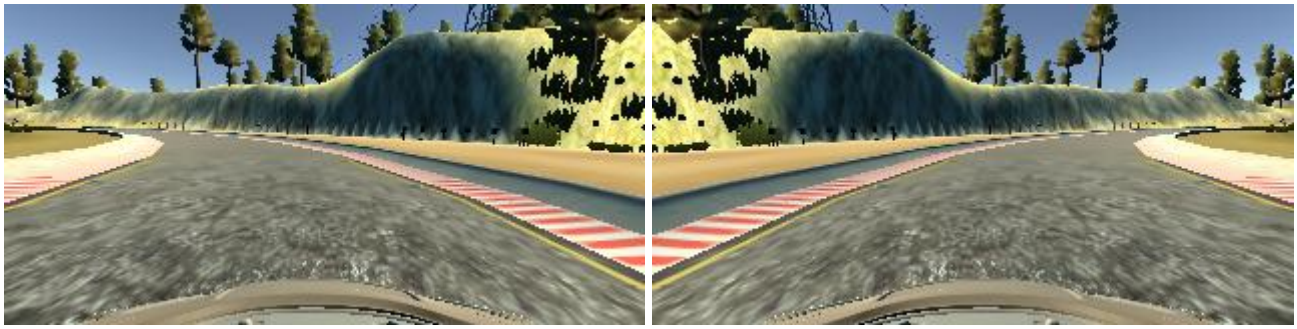




####3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded three laps on track one using center lane driving. To obtain a smoother data, I used a mouse and I drove very slow round the corner. Here is an example image of center lane driving:

To augment the data sat, I also flipped images and angles thinking that this would double the data set. For example, here is an image that has then been flipped:



After the collection process, I had X number of data points. I then preprocessed this data by lambda layer and toke each pixel in an image and ran it through following formulas:

pixel_normalized = pixel / 127.5

pixel_mean_centered = pixel_normalized - 1.0

I alse used Cropping2D layer to crop 60 rows pixels from the top of the image and 20 rows pixels from the bottom of the image.

Instead of storing the preprocessed data in memory all at once, I finally used a generator feed the data and randomly shuffled the data set with put 2% validation set.

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 3 as evidenced by the val_loss was not significantly decline. I used an adam optimizer so that manually training the learning rate wasn't necessary.