# Behavioral Cloning

## 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
- trained video file Track1.mp4

#### 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 convolutional 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 parallels that described in the Nvidia end-to-end learning paper with the following differences:

- 200x70x3 input rather than 200x66x3
- Batch normalization prior to every layer to make sure that the model remains well scaled throughout
- No fixed input normalization (not required with batch normalization)
- ELU activation functions rather than RELU activation functions since ELU learning performance is generally slightly better (though computational complexity may be higher)
- Explicit glorot_uniform (He) initialization
- Dropout with rate of 0.5 for all dense layers
- Bias suppression for layers fed by batch normalization

## 2. Attempts to reduce overfitting in the model

I used several approaches to reduce overfitting in the model:

- All densely connected layers are preceded by a dropout layer.
- 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.
- The model was trained on augmented data from both tracks, and therefore a large amount of varied data was available.

The validation scores during 20 epochs of training decreased more or less monotonically, indicating that the model did not over-fit.

## 3. Model parameter tuning

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

The principal hyper-parameters requiring tuning had to do with the augmentation methods. These were specifically:

- The cropping area for the image presented to the network: (x=60:260, y=60:130)
- The maximum lateral translation of camera images: (40 pixels)
- The maximum angular rotation (about the vertical axis) of camera images: (20 degrees)

- The scaling of corrections to the steering angle for lateral translations: (0.05)
- The scaling of corrections to the steering angle for angular rotations: (0.20)

**4. Appropriate training data**

I experimented with many different kinds of data including data from forward and reverse directions, data illustrating recovery driving, data from trouble spots like corners and earthen berms, and more. I trained the final model using a much smaller data set that contained one forward lap, and a handful of specific trouble areas for each track. I used image augmentation to expand this limited data significantly in ways reflective of a much larger data pool.

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 choose a model that would be sufficiently powerful for the task, make any changes necessary to comply with current best practices, and work on training data and augmentation processes that would efficiently produce a capable model.

My first step was to use a convolutional neural network model similar to the model described in the Nvidia end-to-end learning paper. I thought this model might be appropriate because the task for which it was designed and to which it was successfully applied is almost exactly like the simulator driving task.

I modified that model by adding a dropout layer prior to each densely connected layer in order to reduce any tendency to over-fit. I added batch normalization layers because these often help deep network architectures to train more rapidly by keeping all model layers operating at reasonable dynamic ranges. I explicitly used a glorot_uniform initializer because this too, can help keep signal scaling consistent throughout the network and has been shown to produce more rapid training and superior trained network performance. I changed activation functions to ELU from RELU because these exhibit more rapid training and often produce superior final results, in part by avoiding the "dead neuron" problem in which neurons become inactive and there is no gradient pressure to draw them back into use.

The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where the vehicle fell off the track. There appeared to be various reasons for these failures, but at root many appeared to be the result of inadequate training pressure to learn the desired behavior from the training data. I attempted to correct these problems by gathering more training data to illustrate the necessary behavior, and by designing augmentation methods for a similar effect.

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 consisted of a convolutional neural network with the following layers and layer sizes:

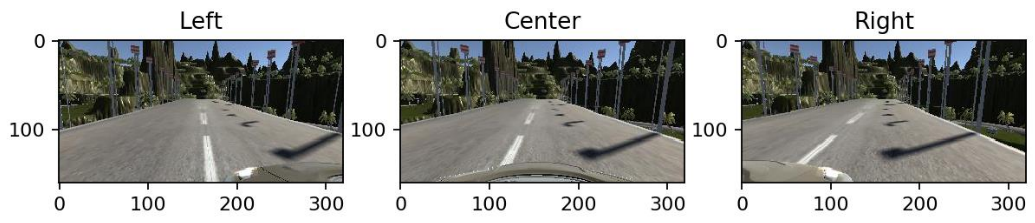
- input_shape=(70, 200, 3)
- BatchNormalization()->Convolution2D(24, 5, 5, subsample=(2,2))
- BatchNormalization()->Convolution2D(36, 5, 5, subsample=(2,2))
- BatchNormalization()->Convolution2D(48, 5, 5, subsample=(2,2))
- BatchNormalization()->Convolution2D(64, 3, 3)
- BatchNormalization()->Convolution2D(64, 3, 3)
- BatchNormalization()->Flatten()->Dropout()->Dense(1154)
- BatchNormalization()->Dropout()->Dense(100)
- BatchNormalization()->Dropout()->Dense(50)
- BatchNormalization()->Dropout()->Dense(10)
- BatchNormalization()->Dense(1, activation='linear')

Activation functions were 'ELU' for all but the final dense layer.

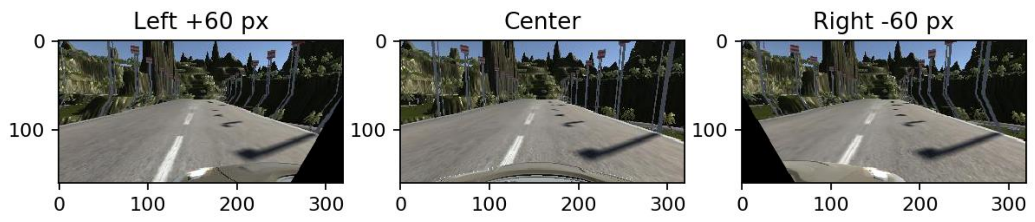
**3. Creation of the Training Set & Training Process**

My final training set consisted of data from both tracks. It included one complete lap around each track as well as several passes over trouble spots that I had observed. All of this training data was collected while driving along a natural path near the center of the track. For track one, the only trouble spot was after the bridge where there is a left turn with an earthen berm. For track two, there were numerous areas of difficulty including several sharp turns and several areas in which strong shadows or poor lighting confused the model.

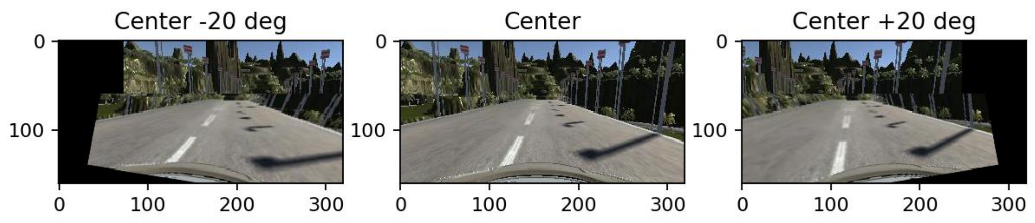
Each sample contains images from a left, center, and right camera as illustrated in the images below:



If these images are compared, the bottom edge of the left and right cameras is offset by roughly 60 pixels from left to center and from right to center. Perspective transformation can alter the perspective of an image as though it had been taken from a laterally offset position. Such transformations can adjust left or right camera images to appear as though they had come from the center camera as illustrated in below:



Perspective transformation can also alter the perspective of the image as if the camera image plane had been rotated about the vertical axis. Examples of this rotation processing are illustrated below:



These perspective transformations introduce blank areas at the image edges and the rotation transformation introduces distortion – particularly at the horizon, that I did not attempt to correct for these figures. For modest transformation parameters, the blank areas fall outside the cropping region.

As mentioned in the Nvidia end-to-end learning paper, I used these perspective transformations to allow my generator to synthesize images with random amounts of translation and rotation, and modified the steering angle correspondingly. In this case the 320x160 pixel input image was cropped to 200x70. Translations of up to +/-40 pixels were applied along with rotations of up to +/-20 degrees. When translating the image, the steering angle was adjusted by .00125 per pixel. When rotating the image, the steering angle was adjusted by .01 per degree. These augmentations dramatically improved the driving results and made it possible to train the network with very little data and no explicit recovery training data.