# 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. An appropriate model architecture has been employed

My model is based off from the NVIDIA Self-Driving convolution neural network (CNN), (model.py lines 111-127).

The model includes RELU layers to introduce nonlinearity, and the data is normalized in the model using a Keras lambda layer (code line 105).

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

The model contains spatial dropout layers in order to reduce overfitting after each convolution layer. Regular dropout layers have been introduced before the first fully connected layer and before the last fully connected layer.

The model was trained and validated on different data sets to ensure that the model was not overfitting. Data was collected by driving clockwise and counter-clockwise. 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 130).

### 4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, recovering 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 start with the NVIDIA CNN architecture and modify as needed after experimenting with the results while running during autonomous mode.

My first step was to use a convolution neural network model similar to the NVIDIA CNN network. I thought this model might be appropriate because it has a 5 layers of convolutions, followed by fully connected layers. This is a very good regression model when using images as input.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. I generally always got similar training and validation loss values. However, the car did not drive properly. Best results were acquired when keeping the correction factor for the left and right steering measurements to 2, and by adding Spatial Dropouts and Regular Dropouts.

To combat the overfitting, I modified the model so that dropouts were used and minimal number of epochs.

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... to improve the driving behavior in these cases, I acquired more data from the turns and changed the steering correction factor for left and right images to 2 from 4.

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 111-127) consisted of a convolution neural network with the following layers and layer sizes:

1. Image Cropping Layer
2. Normalization Layer for Pixel Data
3. Convolution2D with 5x5 Filter and 24 Depth Output
4. ReLU Activation for Non-Linearity
5. Spatial Dropout for 2D Convolution
6. Convolution2D with 5x5 Filter and 36 Depth Output
7. ReLU Activation for Non-Linearity
8. Spatial Dropout for 2D Convolution
9. Convolution2D with 5x5 Filter and 48 Depth Output
10. ReLU Activation for Non-Linearity
11. Spatial Dropout for 2D Convolution
12. Convolution2D with 3x3 Filter and 64 Depth Output
13. ReLU Activation for Non-Linearity
14. Spatial Dropout for 2D Convolution
15. Convolution2D with 3x3 Filter and 64 Depth Output
16. ReLU Activation for Non-Linearity
17. Spatial Dropout for 2D Convolution
18. Flatten Layer
19. Dropout Layer
20. Fully Connected (output size=100) with ReLU Activation
21. Fully Connected (output size=50) with ReLU Activation
22. Fully Connected (output size=10) with ReLU Activation

23. Dropout Layer
24. Fully Connected (output size=1)

### 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. 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 recover in case it drifted to the either side of the road. These images show what a recovery looks like:

#### Starting Point

#### Recovering

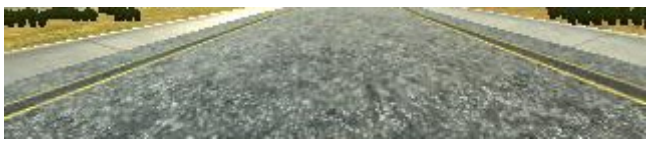
#### Recovered

To normalize, process, and augment the data set, the input images were cropped to remove distractions that could affect the network from learning, the pixel data was normalized to be in a range of -0.5 and 0.5 and to augment data, images and angles were flipped thinking that this would help the network generalize better... For example, here is an image that has been cropped and then flipped:

**Original Image**

**Cropped Image**

**Flipped Image**



After the collection, preprocessing and augmentation process described above, using all 3 cameras, I had 30,804 number of data points.

I finally randomly shuffled the data set and put 20% 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. The ideal number of epochs was 10 as evidenced by both the training loss and validation loss. Both loss values were not decreasing much after 10 epochs. I used an adam optimizer so that manually training the learning rate wasn't necessary.