**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
- video.mp4 showing the driving of the car in track 1
- im2video.py file that was used to generate video from the run images
- writeup_report.pdf summarizing the results

2. Submission includes functional code Using the Udacity provided simulator and my drive.py file. The drive.py file was modified to reflect some preprocessing of the image data that were performed outside the model in the model.py file. Those preprocessing include cropping and flipping. No normalization was done inside drive.py as the

normalization was done inside the model in model.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 consists of a convolution neural network based on the suggested NVIDIA model as referred in the class.

Initially we normalize and center the image. Then I added three 5x5 convolution layers (output depth 24, 36, and 48), with 2x2 stride. Next I added two 3x3 convolution layers (output depth 64, and 64). Then we flatten the layer and add three fully connected layers (depth 100, 50, 10) and dropouts. And then I added a fully connected output layer.

The model includes ELU layers to introduce nonlinearity and the data is normalized in the model using a Keras lambda layer (code line 180).

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

The model contains 3 dropout layers in order to reduce overfitting (model.py lines 204, 207, 210 etc.).

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 (model.py line 216).

### 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

I started with using a simple one layer model, trained the network and was able to run the car a little while. The loss on those training was greater than 2. Then I preprocessed the images inside the model by normalizing and mean centering (model.py line 180).

Although the model improves, it was not enough. So I move to a more complex model - LeNet. I then use data augmentation by Flipping Images and Steering Measurements (model.py line 133-138)

So far the model was trained using only the center images. Later I included more images - that is left and right camera images to better train the model. At this point, the car was driving much better but still not got enough. The car was more prone to be close to the left lane as the training dataset had more images on the left. So I used a correction factor (*correction* and *correctionRight* parameter in model.py) parameter to keep the car closer to the center lane of the road.

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 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.

I removed the unnecessary scenes from the images that are not part of the road by cropping the top and bottom portions of the data set.

Finally, I used even more complex model as suggested by Udacity - the NVIDIA model

To combat the overfitting, I modified the model to include dropout.

## 2. Final Model Architecture

The final model architecture I used is the NVIDIA model

Initially we normalize and center the image. Then I added three 5x5 convolution layers (output depth 24, 36, and 48), with 2x2 stride. Next I added two 3x3 convolution layers (output depth 64, and 64). Then we flatten the layer and add three fully connected layers (depth 100, 50, 10) and dropouts. And then I added a fully connected output layer.

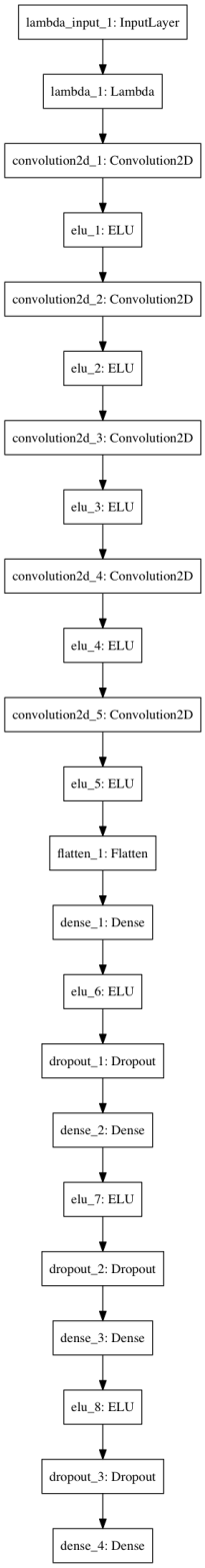


Figure: Final Model Architecture

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

I used the training data set provided by Udacity.

To read the data, I used cv2.imread which returns images in BGR format while drive.py sends images in RGB format. So in model.py, after reading the images using cv2.imread, I convert them to RGB format.

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 5 as after 5, the loss seems to increase. I used an adam optimizer so that manually training the learning rate wasn't necessary.

We augmented the dataset by flipping the image. Below is a sample image before and after augmentation (flipping). You will see how the trees in the image has flipped from right side of the image to the left side.

Figure: Before augmentation(flipping)

Figure: After augmentation (flipping)

**Thoughts and Learnings:**

I used NVIDIA model. Another option can be to use comma.ai model

This helpful guide from Paul was useful specially in terms of avoiding the RGB and BGR pitfalls.

Another helpful guide on changes that need to be applied to drive.py is here

Note to self: spend more time with generators

I had some issues with keras version. Always make sure you are using the CarND Term1 Starter Kit

Using the Udacity provided video.py file, I couldn't generate the mp4 video. So I used im2video.py file found here to generate video from images.