Required Files

My project includes the following files:

- model.py contains the script to create and train the model as well as some additional functionality for batching and preprocessing
- drive.py is the script that can be used to drive the car in autonomous mode by providing a trained model
- flip_raw_images.py contains the preprocessing of the data that creates a dictionary with all images and their corresponding steering angles
- model.h5 contains the trained convolutional neural network based on the comma.ai architecture
- writeup_report.md/.pdf contains the report for this project
- video.mp4 is a recording of my model driving course 1 in autonomous mode

Quality of Code

1. Code functionality

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`

2. Code usability and readability

The `model.py` file contains the code for training and saving the convolutional neural network. In order to process the training data I chose to use a generator that yields the relevant parts of the data everytime it is called (instead of having to hold all image data in memory).

Model Architecture and Training Strategy

1. Model architecture

After initial attempts with a custom architecture and the NVIDIA architecture, I found that the comma.ai architecture gave me pretty good results also with regard to the complexity of the model. I added two layers, one for normalizing the images (Lambda layer) and one for cropping the images (Cropping) to remove the parts of the image like sky and background that are not of importance for the model to predict the steering angle.

Layer	Description
Input	160x320x3 RGB image
Lambda	Normalization layer, outputs 160x320x3
Cropping	outputs 65x320x3
Convolution 8x8	4x4 stride, same padding, outputs 17x80x16
ELU	
Convolution 5x5	2x2 stride, same padding, outputs 9x40x32
ELU	
Convolution 5x5	2x2 stride, same padding, outputs 5x20x64
Flatten	outputs 1x6400
Dropout	Rate: 0.2
ELU	
Fully connected	outputs 1x512
Dropout	Rate: 0.5
ELU	
Fully connected	outputs 1x1

Overall my model consists of three convolutional layers with 8x8 and 5x5 filters. Filter depths range thereby from 16 to 64. (`model.py` lines 63-67) The final layers are two dense layers (code lines 71 and 74) with a depth of 512 and 1 (output layer) respectively.

In general, the model includes ELU layers to introduce nonlinearity (code lines 64, 66, 70, 73) and prevent the vanishing gradient problem. The data is normalized in the model using a Keras lambda layer (code line 61) and cropped (code line 62).

2. Reducing overfitting

The model contains dropout layers in order to reduce overfitting (`model.py` lines 69 and 72).

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 18) with a train/test split size of 80 to 20. The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Parameter tuning

The model used an adam optimizer. However, I also experimented myself with different learning rates [1e-1, ..., 1e-5] and found that a learning rate of 1e-4 yielded the best results for me (`model.py` lines 57 and 75).

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, as well as re-recording of road parts that were particularly difficult for the model to manage (e.g. the first two curves after the bridge).

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

Model Architecture and Training Strategy

1. Solution Design Approach

The overall task of this assignment was to create a neural network that is able to take in an image as an input and be able to predict a steering angle associated with that image. Performed on a sequence of images this should allow to drive a car around a track.

Due to the spatial input, my first step was to use a convolutional neural network model. I normalized the input data (`model.py` code line 61) in order to allow the optimizer to converge faster to an optimal solution. Also, I introduced a cropping layer (`model.py` code line 62) in order to remove parts of the image that yield no additional information the network (i.e. the parts of the image with skies and background above the road) and generally make the training faster (smaller input). Apart from experiments I did myself to get a feeling for the problem (e.g. with regard to the explosion of parameters if filter sizes are chosen to small on a large image), I chose to resort to proven architectures for this particular problem. As pointed out above I first employed the NVIDIA architecture but later on chose the architecture of comma.ai. Overall I have to say the process for model selection was guided by the experiments I performed. Since I had severe memory issues in the beginning even though I used a generator (I chose smaller filter sizes in the first layers of the traditional NVIDIA architecture), I then experimented with the comma.ai architecture. Since I only later realized the memory problem was related to the lower filter sizes and found that comma.ai yielded pretty good results for training and validation I decided to stick with this architecture.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. Thereby, I had to consecutively refine the training data. Since my model had severe issues even with large amounts of training data (including additional recovery data), I found that one critical issue was the loading of the image. OpenCV loads images in BGR and not RGB (as in `drive.py`). So my model was trained on BGR data but then trying to predict RGB images which yielded absolutely unreasonable results. After correcting for that I found that after three epochs my validation loss was increasing with the training loss further decreasing. It was only then I reintroduced the Droput layers from

the comma.ai architecture. Since the loss was still somewhat "jumpy" (both on training and validation set) I experimented with different values for the learning rate (as described above) and found that a manual learning rate of 1e-4 actually yielded more stable results. Even though I then had to increase the number of epochs from 5 to 10.

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` code lines 60-75) consisted of a convolutional neural network as in more detail described in the previous section.

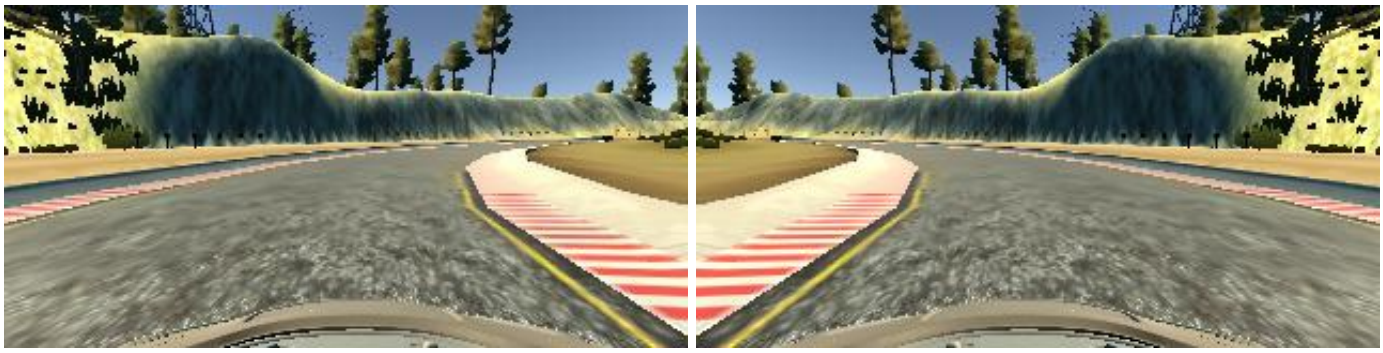
3. Creation of the Training Set & Training Process

To capture good driving behavior, I overall recorded three 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 right side and left sides of the road back to center so that the vehicle would learn to recover when it ends up too far on the side of the road. These images show what a recovery looks like starting from the left, then moving back to center:

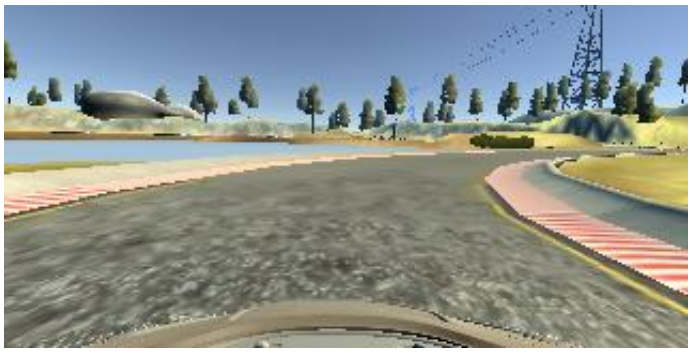
To augment the data sat, I also flipped images and angles thinking that this would help the model to learn better. For example, here is an image that has then been flipped:



Since there were a few spots where the vehicle fell off the track I collected additional training data to improve the driving behavior in these cases. Examples of these cases are the following:

1. No line marking on the right

1. Sharp right turn



For training I used all images (center, left, and right) as well as their flipped equivalents. Left and right images were corrected with a steering angle +- 0.2, in order to compensate for the relative position to the left or to the right of the center camera. All preprocessing has been done in `flip_raw_images.py`.

I finally randomly shuffled the data set and put 20% of the data into a validation set (`model.py` code line 18).

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 a decreasing training and validation loss up to that epoch. I used an adam optimizer but with a manual learning of 1e-4.