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 in the simulator (This is provided by Udacity, my only modification was to increase the car speed on code line 47 from 9 to 18)
- **model.h5** : Containing a trained convolution neural network.
- **writeup_report.pdf** : Summarizing the results. It explained the structure of your network and training approach. The writeup also include examples of images from the dataset in the discussion of the characteristics of the dataset.
- **video.mp4:** A video recording of my vehicle driving autonomously in one lap around the track.

2. Submission includes functional code

Using the Udacity provided simulator and my modified drive.py file, the car can be driven autonomously around the track by executing this command (the 4th argument, video, is a directory name):

```
Python drive.py model.h5 video
```

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 with 3x3 filter sizes and depths between 32 and 128 (model.py lines 112-125)

The model includes RELU layers to introduce nonlinearity (code line 115-119), and the data is normalized in the model using a Keras lambda layer (code line 113).

2. Attempts to reduce overfitting in the model

I tried using a drop out layer in order to reduce overfitting (model.py line 120).

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 130). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track. The easiest method to reduce overfitting is just watching the loss for training and validation and making sure they converge and stopping at that point.

3. Model parameter tuning

The model used an Adam optimizer, so the learning rate was not tuned manually (model.py line 129).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. Also, the data provided by Udacity, I used the first track data. The simulator provides three different images: center, left and right cameras. Each image was used to train the model.

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 essentially use models that were already created. I tried LeNet, Nvidia's, and also variations of the two.

My first step was to just get something working to give me the confidence to understand how the data was responding on the actual track.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. More often than not things were equal in terms of mean squared error. The starting point with LeNet provided a decent base and moving to Nvidia's model worked even better enough.

I found my main problem was at the first turn after the bridge where no lane lines were present and you could run off into the dirt. This problem wasn't solved by changing the model though. This problem was solved prior to the creation of the model in preprocessing of the data.

2. Final Model Architecture

The final model architecture is shown in the following image:

The final model architecture (model.py lines 123-136) consisted of a convolution neural network with the following layers and layer sizes.

Layer (type)	Output Shape	Param #	Connected to
lambda_1 (Lambda)	(None, 160, 320, 3)	0	lambda_input_1[0][0]
cropping2d_1 (Cropping2D)	(None, 65, 320, 3)	0	lambda_1[0][0]
conv2d_1 (Conv2D)	(None, 31, 158, 24)	1824	cropping2d_1[0][0]
conv2d_2 (Conv2D)	(None, 14, 77, 36)	21636	conv2d_1[0][0]
conv2d_3 (Conv2D)	(None, 5, 37, 48)	43248	conv2d_2[0][0]
conv2d_4 (Conv2D)	(None, 3, 35, 64)	27712	conv2d_3[0][0]
conv2d_5 (Conv2D)	(None, 1, 33, 64)	36928	conv2d_4[0][0]
dropout_1 (Dropout)	(None, 1, 33, 64)	0	conv2d_5[0][0]
flatten_1 (Flatten)	(None, 2112)	0	dropout_1[0][0]
dense_1 (Dense)	(None, 100)	211300	flatten_1[0][0]
dense_2 (Dense)	(None, 50)	5050	dense_1[0][0]
dense_3 (Dense)	(None, 10)	510	dense_2[0][0]
dense_4 (Dense)	(None, 1)	11	dense_3[0][0]

Total params: 348,219

Trainable params: 348,219

Non-trainable params: 0

Here is a visualization of the architecture.

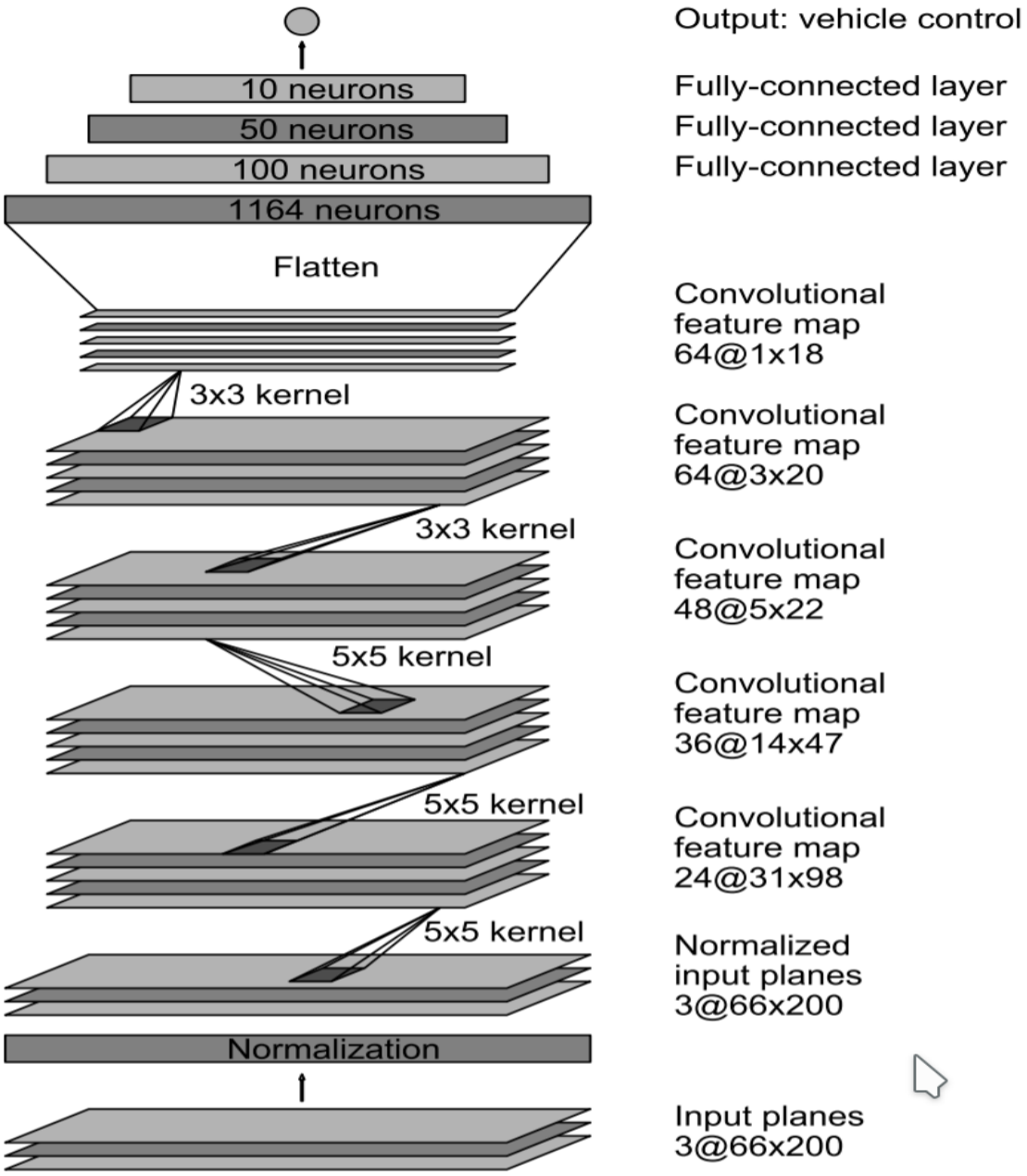
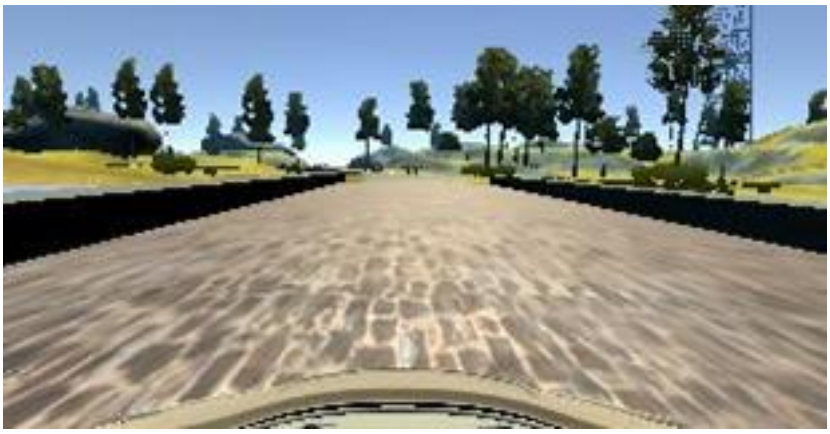


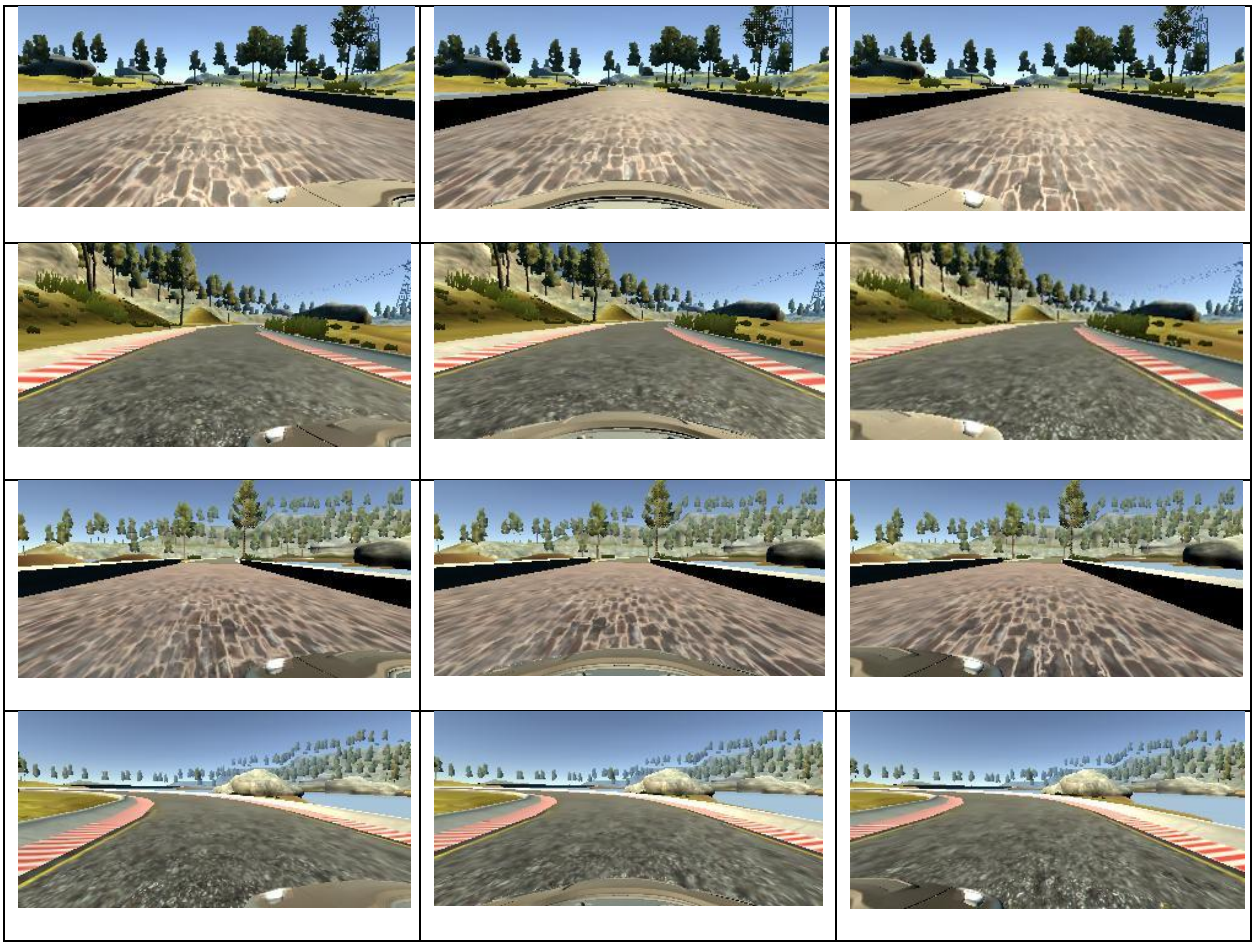
Image taken from - http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf

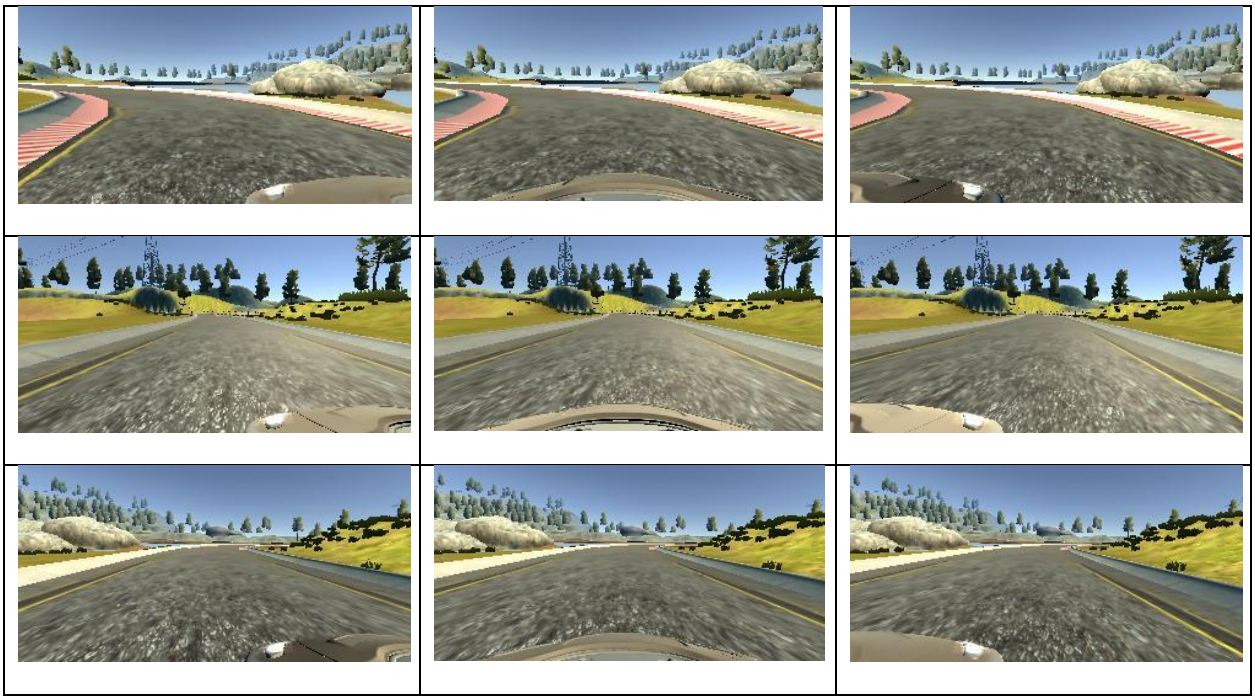
3. Creation of the Training Set & Training Process

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 move back to center. These images show what a recovery looks like starting from





To augment the data set, I also flipped the image along its vertical axis and multiply the steering angle by -1 to reflect the change in the curve direction.

I finally randomly shuffled the data set and put 10% 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 10/10 (so, no over-fitting) as evidenced by following results from log file:

Train on 43394 samples, validate on 4822 samples

Epoch 1/10

43394/43394 [==============================] - 78s - loss: 0.0184 - acc: 0.1803 - val_loss: 0.0170 - val_acc: 0.1860

Epoch 2/10

43394/43394 [==============================] - 62s - loss: 0.0170 - acc: 0.1803 - val_loss: 0.0154 - val_acc: 0.1860

Epoch 3/10

43394/43394 [==============================] - 62s - loss: 0.0166 - acc: 0.1803 - val_loss: 0.0150 - val_acc: 0.1860

Epoch 4/10

43394/43394 [==============================] - 63s - loss: 0.0164 - acc: 0.1803 - val_loss: 0.0148 - val_acc: 0.1860

Epoch 5/10

43394/43394 [==============================] - 62s - loss: 0.0162 - acc: 0.1803 - val_loss: 0.0148 - val_acc: 0.1860

Epoch 6/10

43394/43394 [==============================] - 62s - loss: 0.0158 - acc: 0.1803 - val_loss: 0.0143 - val_acc: 0.1860

Epoch 7/10

43394/43394 [==============================] - 62s - loss: 0.0153 - acc: 0.1803 - val_loss: 0.0143 - val_acc: 0.1862

Epoch 8/10

43394/43394 [==============================] - 62s - loss: 0.0149 - acc: 0.1803 - val_loss: 0.0138 - val_acc: 0.1864

Epoch 9/10

43394/43394 [==============================] - 62s - loss: 0.0144 - acc: 0.1803 - val_loss: 0.0136 - val_acc: 0.1864

Epoch 10/10

43394/43394 [==============================] - 62s - loss: 0.0139 - acc: 0.1804 - val_loss: 0.0135 - val_acc: 0.1864

I used an adam optimizer so that manually training the learning rate wasn't necessary.

Deep learning is an exciting field and we're lucky to live in these times of discovery.