Introduction

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

The project is graded according to these Rubric Points, which have been addressed in my project. A table of contents for the rubric point is below:

Rubric Point Table of Contents

i) Required Files

The project submission includes all required files:

- Model.py
- Drive.py
- Model.h5
- Writeup report (this)
- Vdeo.mp4.

ii) Quality of Code

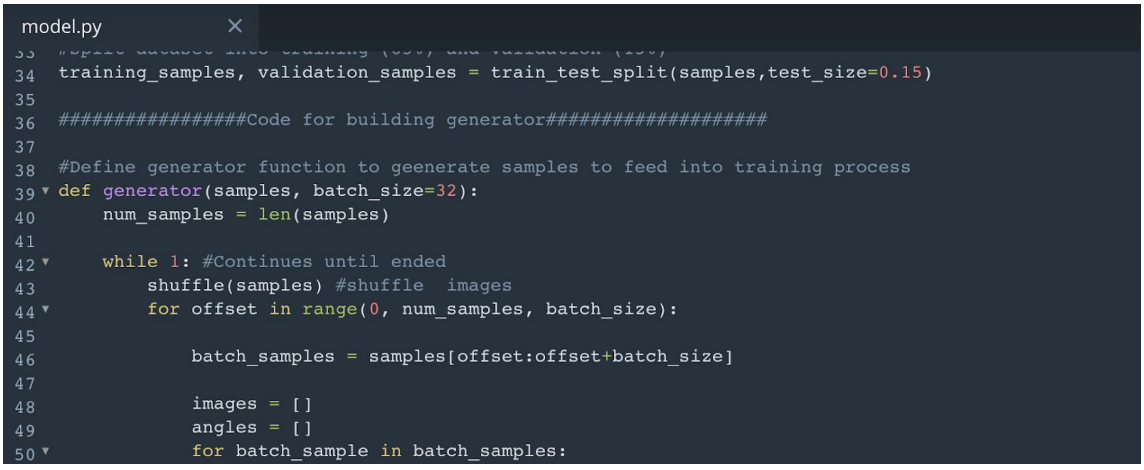
Is the code functional?

The model provided can be used to successfully operate the simulation. The code to drive the car autonomously around the track is:

```
python drive.py model.h5 run1
```

Is the code usable and readable?

The code in model.py uses a Python generator to generate data for training rather than storing the training data in memory. The model.py file contains my code for defining, training and saving my model. The file has my pipeline for training and validating. There are comments to explain how the code works.



Example code documentation from model.py

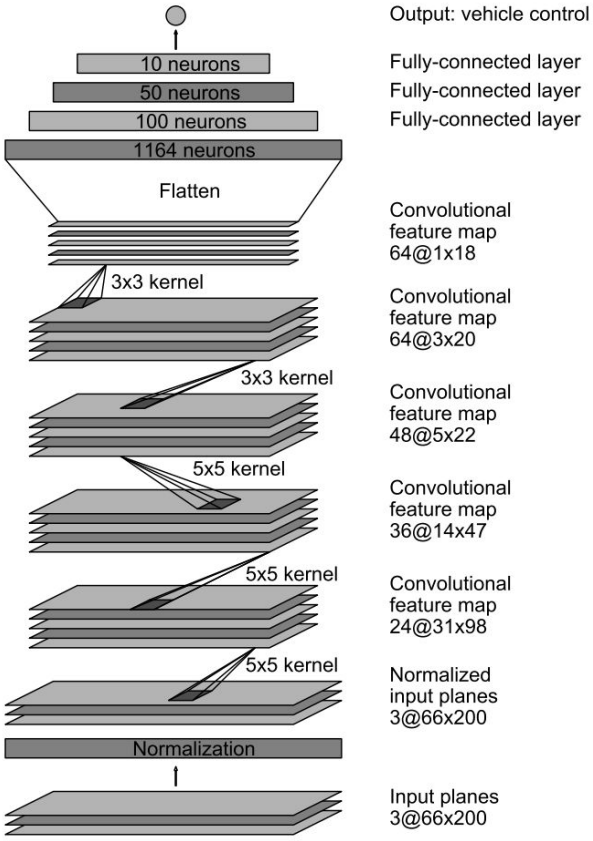
iii) Model Architecture and Training Strategy

1. Has an appropriate model architecture been employed for the task?

The neural network uses convolution layers with appropriate filter sizes. Layers exist to introduce nonlinearity into the model. The data is normalized in the model.

I adapted the suggested Nvidia end-to-end CNN model's architecture (see below). This model takes as an input image (with shape 60 x 266 x3). However, our camera images are 160 x 320 x 3. Hence, I adapted the model to take in a different input image shape.

I will explorate more on the model design later in the model architecture subsection of this readme.



2. Has an attempt been made to reduce overfitting of the model?

Train/validation/test splits have been used, and the model uses dropout layers or other methods to reduce overfitting.

I looked at Udacity's training dataset and was happy with the data. It contained 9 laps driving around the first track and had recovery maneuvers. I split the dataset into a training and validation set according to a 85%-15% ratio.

3. Have the model parameters been tuned appropriately?

Learning rate parameters are chosen with explanation, or an Adam optimizer is used.

In the final model, I used a dropout step after the first fully connected layer to prevent overfitting. I used a dropout rate of 0.25. I also used the Adam optimizer.

4. Is the training data chosen appropriately?

Training data has been chosen to induce the desired behavior in the simulation (i.e. keeping the car on the track).

I primarily used Udacity's provided training dataset. I shuffle training images so that the order of the images doesn't impact the CNN.

I augmented the training data by flipping the image horizontally and adjusting (i.e., change the sign) steering angle accordingly. This approach allowed me to double the number of camera images for each entry in driving_log file.

iv) Architecture and Training Documentation

1. Is the solution design documented?

The README thoroughly discusses the approach taken for deriving and designing a model architecture fit for solving the given problem.

As mentioned above, I started with the Nvidia E2E CNN model as it is trained for a similar task (i.e., take in a camera image and outputs the steering angle). Formally, the steps of my solution pipeline - and the corresponding logic - are shown below:

Data Selection and Loading
As mentioned, I used the sample training dataset provided by Udacity. I used OpenCV to load these images. Since OpenCV reads images in BGR by default, I converted these images to RGB as drive.py processes it in this format (see below).

```
angles = []
for batch_sample in batch_samples:
        for i in range(0,3): #3 images: first -> center, second -> left and third -> right

            name = './data/data/IMG/'+batch_sample[i].split('/')[-1]
            center_image = cv2.cvtColor(cv2.imread(name), cv2.COLOR_BGR2RGB) #CV2 reads img in BGR, so we  convert it to

            steering_angle = float(batch_sample[3]) # steering angle
            images.append(center_image)
```

Image loading and color conversion

I corrected the steering angle associated with the left and right image by introducing a correction factor of 0.2 to these images. This accounts for the fact that the camera is not at the center. For the left camera, I added 0.2 to the steering angle. For the right camera, I subtracted 0.2.

Here is an example of a center camera image from the sample training dataset:

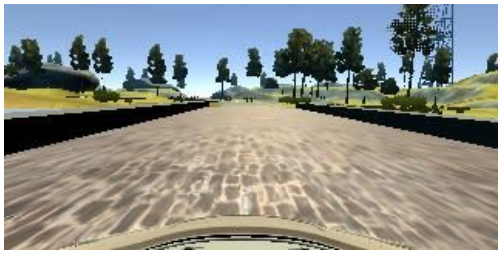


Image Preprocessing Pipeline

I shuffled the sample training images so that the order of the images doesn't impact the CNN. I then augmented the training data by flipping the image horizontally and adjusting (i.e., change the sign) steering angle accordingly. This approach allowed me to double the number of camera images for each entry in driving_log file.

```
###Augment data: flip image and change sign (+ --> - ) of steering angle
images.append(cv2.flip(center_image,1))
if(i==0):
    angles.append(steering_angle*-1)
elif(i==1):
    angles.append((steering_angle+0.2)*-1)
elif(i==2):
    angles.append((steering_angle-0.2)*-1)
```

Data augmentation step in pipeline

Training and Validation Set

The example dataset from Udacity seems to be sufficient for the purpose of this project. The dataset contains 9 laps of the first track and seems to contain recovery manvers. The dataset is split into training and validation at a 85%-15% using sklearn preprocessing library.

```
##################Create training and validation sets#############

#split dataset into training (85%) and validation (15%)
training_samples, validation_samples = train_test_split(samples,test_size=0.15)
```

I used a generator to create the data as the training process runs, rather than loading all the images from memory. Training examples are generated at runtime in batches of 32 images. The augmented images are also created by the generator during runtime.

Final Model Architecture

The final model architecture is discussed in the subsection below.

2. Is the model architecture documented?

The README provides sufficient details of the characteristics and qualities of the architecture, such as the type of model used, the number of layers, the size of each layer. Visualizations emphasizing particular qualities of the architecture are encouraged.

My final model architecture (see below) is an adaptation of [Nvidia's end-to-end CNN model].

Layer (type)	Output Shape	Param #
lambda_3 (Lambda)	(None, 160, 320, 3)	0
cropping2d_3 (Cropping2D)	(None, 65, 320, 3)	0
convolution2d_11 (Conv2D)	(None, 31, 158, 24)	1,824
activation_17 (Activation)	(None, 31, 158, 24)	0
convolution2d_12 (Conv2D)	(None, 14, 77, 36)	21,636
activation_18 (Activation)	(None, 14, 77, 36)	0
convolution2d_13 (Conv2D)	(None, 5, 37, 48)	43,248
activation_19 (Activation)	(None, 5, 37, 48)	0
convolution2d_14 (Conv2D)	(None, 3, 35, 64)	27,712
activation_20 (Activation)	(None, 3, 35, 64)	0
convolution2d_15 (Conv2D)	(None, 1, 33, 64)	36,928
activation_21 (Activation)	(None, 1, 33, 64)	0
flatten_3 (Flatten)	(None, 2112)	0
dense_9 (Dense)	(None, 100)	211,300
activation_22 (Activation)	(None, 100)	0
dropout_3 (Dropout)	(None, 100)	0
dense_10 (Dense)	(None, 50)	5,050
activation_23 (Activation)	(None, 50)	0
dense_11 (Dense)	(None, 10)	510
activation_24 (Activation)	(None, 10)	0
dense_12 (Dense)	(None, 1)	11

Total params: 348,219
Trainable params: 348,219
Non-trainable params: 0

The first stage of the final model is to apply normalization input images.

```
# Preprocess data: center to zero with small standard deviation
model.add(Lambda(lambda x: (x / 255.0) - 0.5, input_shape=(160,320,3)))
```

Code snippet

The second stage is to crop the image by removing the top 70 pixels and bottom 25 pixels. The top is cropped to not distract the model elements above the roadline (e.g., trees, mountain, sky) The bottom is dropped to remove the hood of the car.

Example cropped image

The third stage is three convolution layers followed by ELU activation functions. These three layers are:

1. 1st convolutional layer with filter depth of 24 and 5x5 filter size and 2x2 stride
2. 2nd convolutional layer with filter depth of 36 and 5x5 filter size and 2x2 stride
3. 2nd convolutional layer with filter depth of 48 and 5x5 filter size and 2x2 stride

```
#Layer 1 - Convolution: 24 filters, 5x5 filter size, 2x2 stride
model.add(Convolution2D(24,5,5,subsample=(2,2)))
model.add(Activation('elu'))

#Layer 2 - Convolution: 36 filters, 5x5 filter size, 2x2 stride
model.add(Convolution2D(36,5,5,subsample=(2,2)))
model.add(Activation('elu'))

#Layer 3 - Convolution: 48 filters, 5x5 filter size, 2x2 stride
model.add(Convolution2D(48,5,5,subsample=(2,2)))
model.add(Activation('elu'))
```

Code snippet

The fourth stage is two convolutional layers with filter depth of 64 and 3x3 filter size and 1x1 stride, each followed by an ELU activation function.

```
#Layer 4- Convolution: 64 filters, 5x5 filter size, 1x1 stride
model.add(Convolution2D(64,3,3))
model.add(Activation('elu'))

#Layer 5 - Convolution: 64 filters, 3x3 filter size, 1x1 stride
model.add(Convolution2D(64,3,3))
model.add(Activation('elu'))
```

Code snippet

The fifth stage is to flatten the output from 2D a 1D (i.e., side by side values)

The sixth stage is the first fully connected layer - with 100 outputs - followed by an ELU activation function. This is then followed by a dropout layer (rate of 0.25) to reduce overfitting.

```
#Layer 6 - Fully connected layer
model.add(Dense(100))
model.add(Activation('elu'))

#Dropout layer - 25%
model.add(Dropout(0.25))
```

Code snippet

The seventh stage is two fully connected layers, each followed by ELU activations. The second fully connected layer has 50 outputs while the third fully connected layer has 10 outputs.

```
#Layer 7 - Fully connected layer
model.add(Dense(50))
model.add(Activation('elu'))


#layer 8- Fully connected layer
model.add(Dense(10))
model.add(Activation('elu'))
```

Code snippet

The final stage is a fully connected layer with one output since we only need a single steering angle as the prediction output.

Model parameters

The parameters used to train my model are:

- Epochs: 5
- Optimizer: Adam Optimizer
- Learning Rate: 0.001 (default)
- Batch size: 32 images
- Loss Function: Mean Squared Error

```
#Use fit generator as the # of images are generated by the generator

model.fit_generator(train_generator, samples_per_epoch= len(training_samples), validation_data=validation_generator,
nb_val_samples=len(validation_samples), nb_epoch=5, verbose=1)
```

Code snippet

Model output:

I ran my final model with the drive.py script to record the image frames and ran video.py to create a video of the car driving itself around the track. Note: I increased the speed in drive.py for time efficiency's sake

Video of car driving around track

3. Is the creation of the training dataset and training process documented?

The README describes how the model was trained and what the characteristics of the dataset are. Information such as how the dataset was generated and examples of images from the dataset must be included.

v) Simulation

1. Is the car able to navigate correctly on test data?

In my final video, we see that no tire leaves the drivable portion of the track surface, nor does the car pop up onto ledges or roll over any surfaces that would otherwise be considered unsafe.