Project: Behavioral Cloning

Student: Soroush Razmyar

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 submission includes the following files:

1. **model.py** containing the script to create and train the model
2. **drive.py** for driving the car in autonomous mode.
3. **model.h5** containing a trained convolution neural network
4. **report.pdf** summarizing the results (this file)
5. **capture.mp3** simulation video file.

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
```

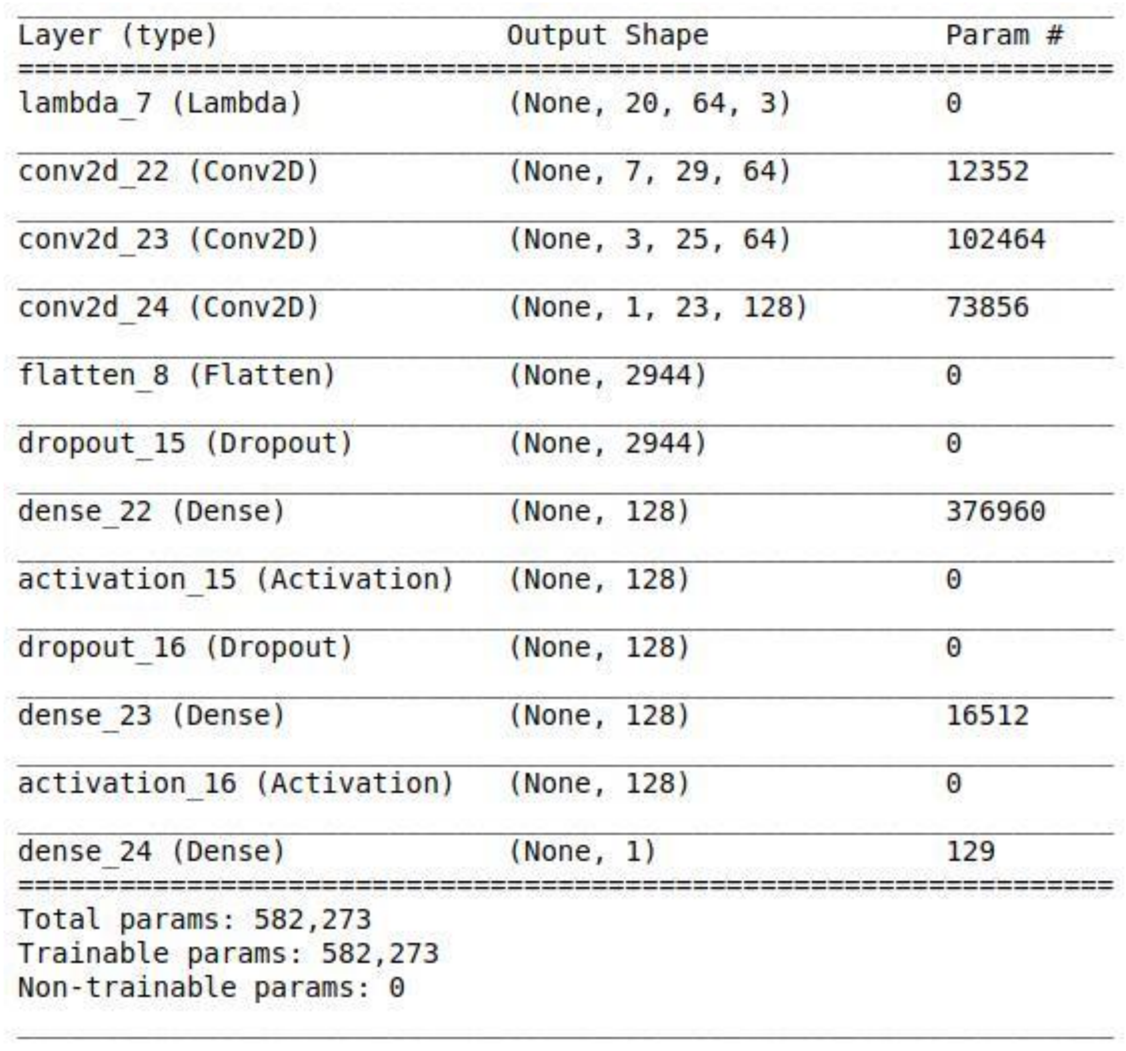
3. Submission code is usable and readable

The model.py file contains the code for training and saving the convolution neural network (model.py, 99). 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

Table 1 represents the summary of my model's architecture. This model (lmodel.py 83 – 96) has three convolutional layers as well as two dense (fully connected) layers. The model includes ELU layers to introduce nonlinearity, and the data is normalized in the model using a Keras lambda layer (model.py, 84).



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_7 (Lambda)            (None, 20, 64, 3)         0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 7, 29, 64)         12352
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 3, 25, 64)         102464
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 1, 23, 128)        73856
_________________________________________________________________
flatten_8 (Flatten)          (None, 2944)              0
_________________________________________________________________
dropout_15 (Dropout)         (None, 2944)              0
_________________________________________________________________
dense_22 (Dense)             (None, 128)               376960
_________________________________________________________________
activation_15 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_16 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_23 (Dense)             (None, 128)               16512
_________________________________________________________________
activation_16 (Activation)   (None, 128)               0
_________________________________________________________________
dense_24 (Dense)             (None, 1)                 129
=================================================================
Total params: 582,273
Trainable params: 582,273
Non-trainable params: 0
_________________________________________________________________
```

Table 1: Model Architecture Summary

2. Attempts to reduce overfitting in the model

The model contains dropout layers to reduce overfitting (model.py lines 89, 92). The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 143 - 149). 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 103).

4. Appropriate training data

Training data was created by using two sets of recorded data: 1) The SamleData provided by Udacity (size: 8036), 2) Data captured through simulator (size: 7628). I manually joined these two data sets into one set and used it for the training. The training and validation sets was created by using the train_test_split function from the sklearn framework (line 73)

Architecture and Training Documentation

1. Solution Design Approach

I started of by making a simple model similar to the model that was used in training videos to familiarize myself with the process of data collection and simulator. During the data collection, I used a PS4 controller to drive the car. Once the data was captures, I trained the model, and test it with the simulator. I started implementing the details after this initial step.

I used the Numpy and Pandas libraries to read the csv files. Images for center, left, and right cameras and the steering angle data was read. I resized and crop images right after reading them from the disk. The drive.py code was also modified to work with cropped images. Although I could have use a batch_reader to read the images in batch, that would introduce some unnecessary overhead to our code. Batch generators are useful when dealing with large datasets. However, here the dataset is not large. So, I read the images into memory, and used Keras **ImageDataGenerator** function to send batches of data from memory into the GPU to be used by the model (this will be explained shortly). After resizing and cropping, the three types of the images data were joined together (line 66). The steering angles was also corrected to correspond to the left and right camera (line 69).

In order to develop a CNN model, I started with a simple CNN model, and gradually add additional layers. My objective was to drive the car with the smallest possible network. In order to decrease overfitting, I used two Droup_out layers with probability of 0.75. However, initial observation showed that model will learn better with lower (0.5) drop_out probability.

In order to train the model, batches of data were generated by the **ImageDataGenerator** function. This function works in parallel to the model and feed data into the model. Through this process, some real-time image augmentation was also applied to the images.

The final step was to run the simulator to see how well the car was driving around track one. There were some “critical” spots where the vehicle fell off the track. To improve the driving behavior in these cases, I re-recorded training data with more emphasize on driving on those critical spots. I also learned that the best approach in creating a useful training data is to minimize using steering angle as few times as possible. Following this approach significantly decreased the “off-spot” incidences. At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

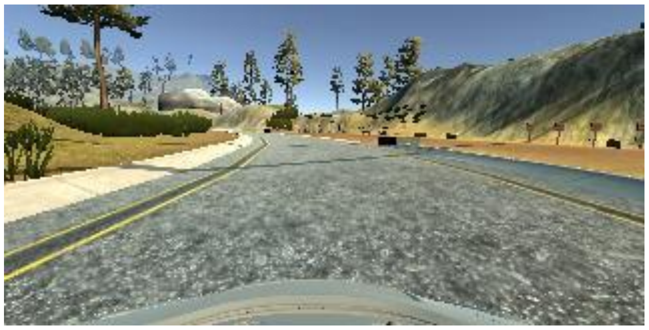
The final model architecture is displayed in the table 01 in previous section. Here is some additional info for the convolutional layers.

Layer	Depth	Filter Size	Activation
Conv 1	64	8x8	-
Conv 2	64	5x5	elu
Conv 3	128	3x3	elu

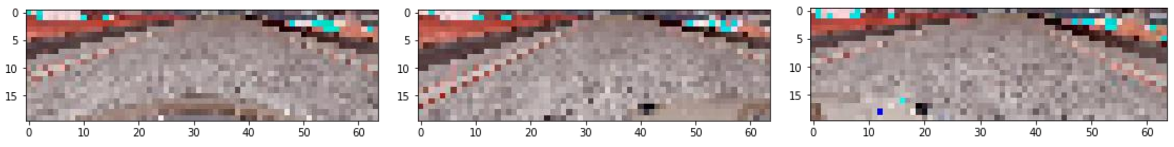
There were also three dense layers with size of 128, activation elu. Check table 1 for more information.

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:



Here is example of the images after resizing and cropping:



After the collection process and merging it with the Udacity sample data, I had 15664 number of data points. I used Sklean *train_test_split* function to shuffle and split the dataset. This process put 25% of the data into a validation set. During the training model with the lowest loss were saved. I trained the model for 3, 5, 10, 15 and 30 epochs. The ideal number of epochs was 15. After 15 epochs, the model will suffer from over fitting.