Writeup for Behavioral Cloning

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](https://review.udacity.com/#!/rubrics/432/view) 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
* writeup_report.pdf summarizing the results

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

```sh
python CarND-Behavioral-Cloning/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

I build my network based on the NVIDIA's self driving car network as discussed in the 'even more powerful network' lesson. As shown in image1, lambda layer is added to normalize images, cropping layers is used to crop useless pixels from images, then a convolutional layer with 5*5 filters. A maxpooling layer is added to reduce width and height dimension, so less parameters means less overfitting.

After five convolutional layers, I used a flatten layer to flat image to be a vector, then a fully connected layer with activation function of relu. A dropout layer is followed to reduce overfitting.

after four fully connected layer, I use the final dense layer with one output to predict the steer angle.

```
Using TensorFlow backend.
```

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 160, 320, 3)	0
cropping2d_1 (Cropping2D)	(None, 65, 320, 3)	0
conv2d_1 (Conv2D)	(None, 61, 316, 24)	1824
max_pooling2d_1 (MaxPooling2	(None, 30, 158, 24)	0
conv2d_2 (Conv2D)	(None, 26, 154, 36)	21636
conv2d_3 (Conv2D)	(None, 22, 150, 48)	43248
max_pooling2d_2 (MaxPooling2	(None, 11, 75, 48)	0
conv2d_4 (Conv2D)	(None, 9, 73, 64)	27712
conv2d_5 (Conv2D)	(None, 7, 71, 64)	36928
flatten_1 (Flatten)	(None, 31808)	0
dense_1 (Dense)	(None, 240)	7634160
dropout_1 (Dropout)	(None, 240)	0
dense_2 (Dense)	(None, 50)	12050
dropout_2 (Dropout)	(None, 50)	0
dense_3 (Dense)	(None, 24)	1224
dense_4 (Dense)	(None, 10)	250
dense_5 (Dense)	(None, 1)	11

```
Total params: 7,779,043
Trainable params: 7,779,043
Non-trainable params: 0
```

Image1

2. Attempts to reduce overfitting in the model

The model contains dropout layers in order to reduce overfitting, at first I my network has much larger dense layer, with dropout rate 0.4, I reduce the number of perceptron in dense layers and change dropout rate from 0.4 to 0.5, my model seems to perform better.
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 116. And I have used ‘relu’ as activation function to introduce nonlinearity into my model.

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, I have set the correction number to be 0.22, and I used numpy's fliplr function to generate symmetrical images for training, it may help the model to generalize better.

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 build a deep neural network the starts with some convolutional layers to extract information from the images, the first layer may just see lines, second one may see shapes, with more convolutional layers, the model can extract higher information. Then some fully connected layers are added to use the information to finally predict a turning angle.

My first step was to use a convolution neural network model similar to the NVIDIA's self driving car network as discussed in the 'even more powerful network' lesson. I thought this model might be appropriate because they both solve similar problems

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, as shown in image 2.

```
model=Sequential()
model.add(Lambda(lambda x: x/255.0-0.5, input_shape=(160,320,3)))
model.add(Cropping2D(cropping=((70,25),(0,0))))
model.add(Conv2D(filters=24, kernel_size=5, padding='valid', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=36, kernel_size=5, padding='valid', activation='relu'))
#model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=48, kernel_size=5, padding='valid', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, padding='valid', activation='relu'))
#model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, padding='valid', activation='relu'))

model.add(Flatten())
model.add(Dense(1164,activation='relu'))
model.add(Dense(100,activation='relu'))
model.add(Dense(50,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(1))
model.compile(loss='mse',optimizer='adam')
history_object=model.fit(pre_X_train,pre_y_train,validation_split=0.2,shuffle=T
rue,nb_epoch=10,verbose = 1,batch_size=128)
```

Image2

To combat the overfitting, I decrease the number of neuros in dense layer and add some dropout layers, as shown in image3

```
model=Sequential()
model.add(Lambda(lambda x: x/255.0-0.5, input_shape=(160,320,3)))
model.add(Cropping2D(cropping=((70,25),(0,0))))
model.add(Conv2D(filters=24, kernel_size=5, padding='valid', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=36, kernel_size=5, padding='valid', activation='relu'))
model.add(Conv2D(filters=48, kernel_size=5, padding='valid', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, padding='valid', activation='relu'))
model.add(Conv2D(filters=64, kernel_size=3, padding='valid', activation='relu'))

model.add(Flatten())
model.add(Dense(240,activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(50,activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(24,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(1))
model.compile(loss='mse',optimizer='adam')
history_object=model.fit(pre_X_train,pre_y_train,validation_split=0.2,shuffle=True,n
b_epoch=10,verbose = 1)#batch_size=1024
```

Image3

After 30 epochs of training, the valid loss is about 0.035, and the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

The final model architecture is shown in image1.

3. Creation of the Training Set & Training Process

I used the data provided by udacity and I added the left and right images with correction number of 0.2, and I fliped the images to get more data so my model may generalize better. The original image and flipped image is shown in image4.

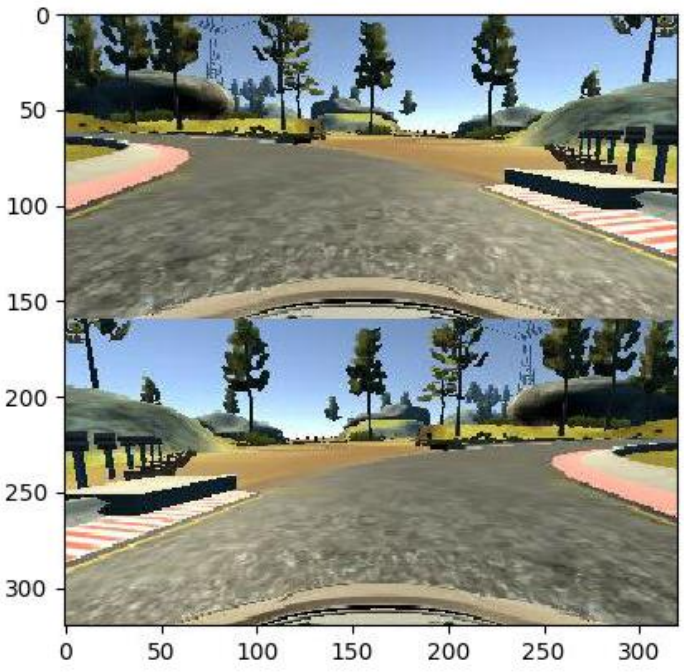

Image4

After the collection process, I had X number of data points. I then preprocessed this data by ‘preprocess’ function(in line36 of model.py).
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 30, as shown in image4, the validation imcreased after 30 epochs, so I chose 30 as epochs for my model. I used an adam optimizer so that manually training the learning rate wasn't necessary.

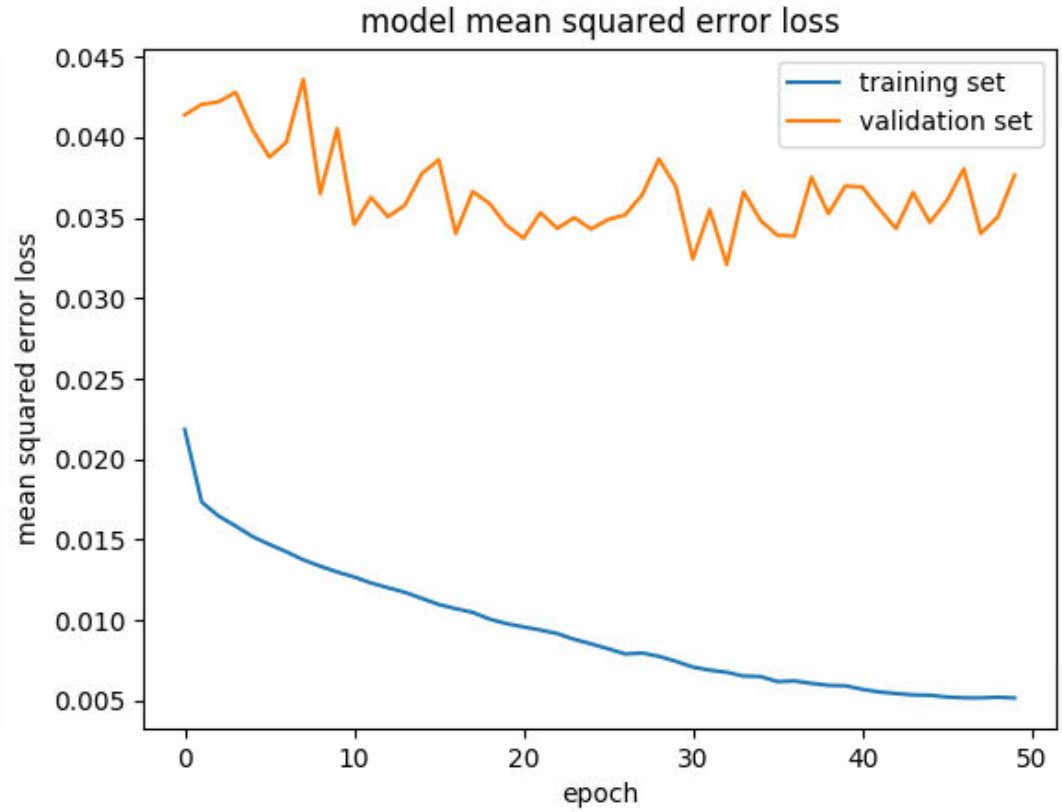

Image4

Modification：

I have added a generator to generate data to feed my model(line 19 in model_2.py), and I used the fit_generator function(line 122 in model_2.py) to train my model. After 5 epochs of training, the validation loss is less than 0.016, as shown in image 5, then I tested this model on the simulator, the video is named as ‘run_2’. My modified model.py file is named as ‘model_2.py’, the H5 file is named as ‘model_2.H5’, I did not change the drive.py file.

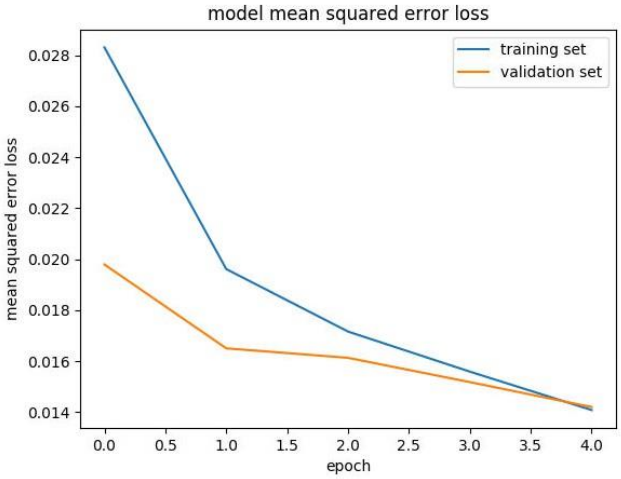
model mean squared error loss
training set
validation set
mean squared error loss
0.028
0.026
0.024
0.022
0.020
0.018
0.016
0.014
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch

Image5