Behavioral Cloning

This is the Project 3 in Self Driving Car Nano degree from Udacity

The purpose of this project is using deep learning to train a deep neural network to drive a car automously in a simulator.

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 (https://review.udacity.com/#!/rubrics/432/view) individually and describe how I addressed each point in my implementation.

Files Submitted & Code Quality

My project includes the following files:

- model.py containing the script to create and train the model, data processing and tuning parameters.
- drive.py for driving the car in autonomous mode
- model1.h5, model2.h5 containing trained convolution neural network
- writeup_report.html or writeup_report.pdf summarizing the results
- run1.mp4, run2.mp4 capturing the simulation running in automonous mode with the selected model.

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
```

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. Model Architecture

My model is based on model proposed by the NVIDIA in this paper (http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf). The NVIDIA model is well-documented and not complicated and has been proven its effectiveness in self driving car control.

The NVIDIA model architecture is summarized as follows:

Layer	Description
Input	66x200x3 RGB image
Lambda	Normalized
Convolution 5x5x24	valid padding, subsample(2,2), activation ReLU, outputs 31x98x24
Convolution 5x5x36	valid padding, subsample(2,2), activation ReLU, outputs 14x47x36
Convolution 5x5x48	valid padding, subsample(2,2), activation ReLU, outputs 5x22x48
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 3x20x64
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 1x18x64
Flatten	outputs 1152
Fully connected	activation ReLU, outputs 100
Fully connected	activation ReLU, outputs 50
Fully connected	activation ReLU, outputs 20
Fully connected	outputs 1

My initial model is modified from NVIDIA model with some modifications as follows:

Layer	Description
Input	**160x320x3 RGB image**
Lambda	**Cropping, outputs 75x320x3**
Lambda	Normalized, outputs 75x320x3
Convolution 5x5x24	valid padding, subsample(2,2), activation ReLU, outputs 36x158x24
Convolution 5x5x36	valid padding, subsample(2,2), activation ReLU, outputs 16x77x36
Convolution 5x5x48	valid padding, subsample(2,2), activation ReLU, outputs 6x37x48
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 4x35x64
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 2x33x64
Dropout	**keeping probability 0.5, outputs 2x33x64**
Flatten	outputs 4224
Fully connected	activation ReLU, outputs 100
Fully connected	activation ReLU, outputs 50
Fully connected	activation ReLU, outputs 20
Fully connected	outputs 1

Total params: 559,419

- My model consisted of a convolution neural network with three 5x5 and 3x3 filter sizes and depths between 24 and 64.
- The convolution layers included RELU activations to introduce nonlinearity and downsample 2 times.
- The data was normalized in the model using a Keras lambda layer.
- Because the captured images have size of 160x320 and include redundant part such as sky and the front part of the car, a Keras lambda layers was added for cropping the redundancy.
- The model contained a dropout layer with keeping probability 0.5 in order to reduce overfitting.
- The model used an adam optimizer with default learning rate 0.001, so the learning rate was not tuned manually.

2. Training Strategy

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,
- driving counter-clockwise to help the model generalized,
- flipping the images to help the model generalized,
- collecting data from the second track can also help generalize the model

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

Architecture and Training Documentation

1. Solution Design Approach

The overall strategy for deriving a model architecture was to use a well-known model that is sucessful in the field of self driving car and then twist model parameters as well as use appropriate image pre-processing and augmentation to the training dataset. The model was trained and validated on different data sets to ensure that the model was not overfitting. The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

My first step was to use a convolution neural network model similar to the NVIDIA model in this paper (http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf). I thought this model might be appropriate because the NVIDIA model is well-documented and not complicated and has been proven its effectiveness in self driving car control.

Then I focused on image pre-processing methods which including:

- Crop the sky (60 pixels) and steering wheel (25 pixels) parts in the image.
- Resize the images from 160x320x3 matrix to 66x200x3 matrix, which is the input size of the NVIDIA model. I tested without/with a Lambda layer for resizing the image to see if the resizing affect the performance of the model. Resizing the images helped reducing the training time but may decrease the model performance.

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.

To combat the overfitting, I modified the model so that the model contains a dropout layer with keeping probability 0.5 in order to reduce overfitting.

The final step was to run the simulator to see how well the car was driving around track one.

There were a few spots where the vehicle fell off the track. To improve the driving behavior in these cases, I collected more training data in the track one, as well as generate more training data using image augmentation methods (see details in the next part).

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 consisted of a convolution neural network with the following layers and layer sizes.

Model 1: without resizing: better performance, slower training time.

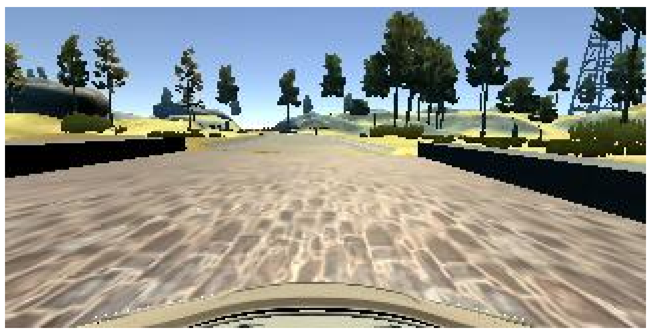
Layer	Description
Input	160x320x3 RGB image
Lambda	Cropping, outputs 75x320x3
Lambda	Normalized, outputs 75x320x3
Convolution 5x5x24	valid padding, subsample(2,2), activation ReLU, outputs 36x158x24
Convolution 5x5x36	valid padding, subsample(2,2), activation ReLU, outputs 16x77x36
Convolution 5x5x48	valid padding, subsample(2,2), activation ReLU, outputs 6x37x48
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 4x35x64
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 2x33x64
Dropout	keeping probability 0.5, outputs 2x33x64
Flatten	outputs 4224
Fully connected	activation ReLU, outputs 100
Fully connected	activation ReLU, outputs 50
Fully connected	activation ReLU, outputs 20
Fully connected	outputs 1

Model 2: with resizing: faster training time.

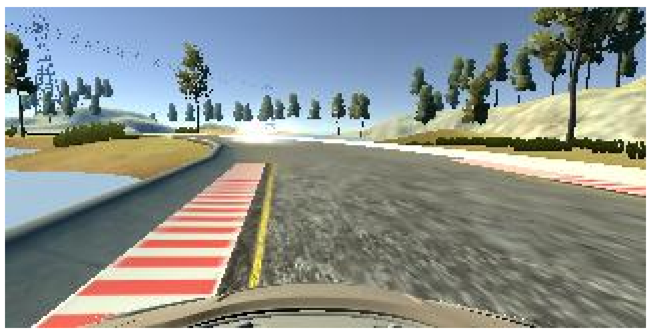
Layer	Description
Input	160x320x3 RGB image
Lambda	Cropping, outputs 75x320x3
Lambda	Resized, outputs 66x200x3
Lambda	Normalized, outputs 66x200x3
Convolution 5x5x24	valid padding, subsample(2,2), activation ReLU, outputs 31x98x24
Convolution 5x5x36	valid padding, subsample(2,2), activation ReLU, outputs 14x47x36
Convolution 5x5x48	valid padding, subsample(2,2), activation ReLU, outputs 5x22x48
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 3x20x64
Convolution 3x3x64	valid padding, subsample(2,2), activation ReLU, outputs 1x18x64
Dropout	keeping probability 0.5, outputs 1152, outputs 1x18x64
Flatten	outputs 1152
Fully connected	activation ReLU, outputs 100
Fully connected	activation ReLU, outputs 50
Fully connected	activation ReLU, outputs 20
Fully connected	outputs 1

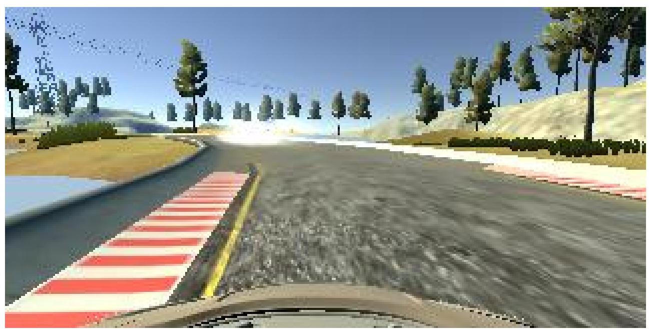
3. Creation of the Training Set & Training Process

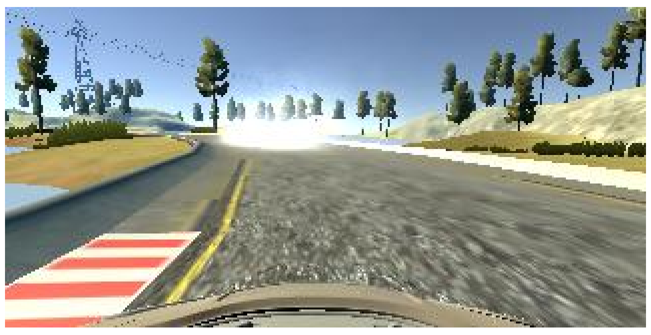
To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving on track one:



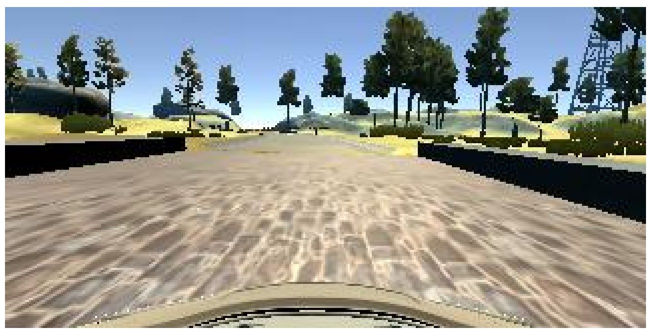
I then recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn how to go back the center of driving from the left side and right sides. These images show what a recovery looks like:

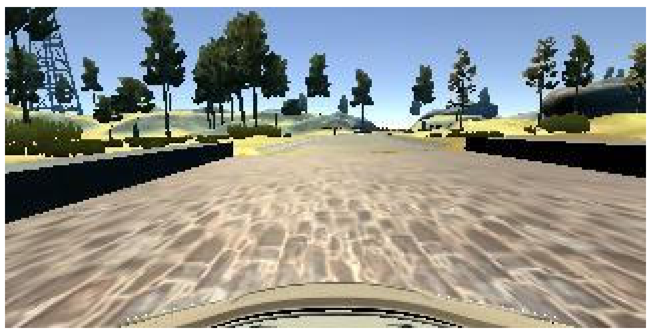






To augment the data sat, I also randomly flipped images and angles thinking that this would balance the left turn bias because the tracks are left-turn. For example, here is an image that has then been flipped:





I also recorded the vehicle running around the curves only so that the vehicle would learn how to go around the curves.

I also recorded the counter-clockwise driving on track one and track two for better model generalization.

After the collection process, I had 36,202 number of data points. I then preprocessed this data by cropping the sky and steering wheel parts in the images.

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.

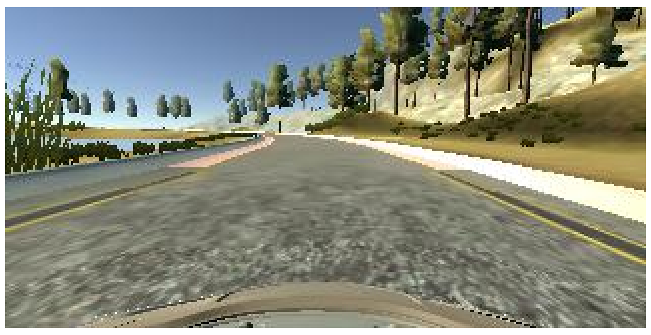
I used batch generator to create training and validation batches with batch size = 32 to help the training and validation process faster and without out-of-memory problem.

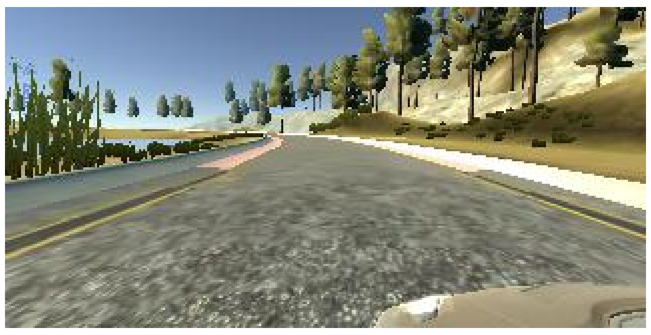
I used an adam optimizer so that manually training the learning rate wasn't necessary.

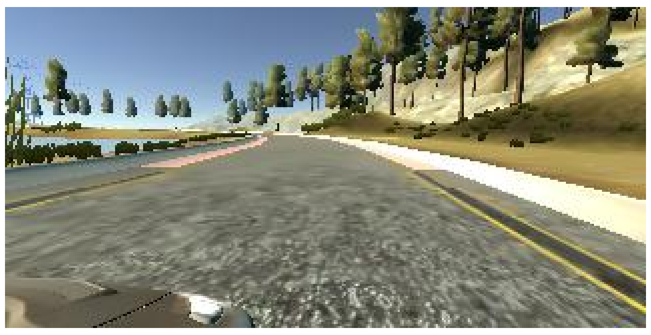
The simulation result after first training times was good at the straight road but sometimes the vehicle went of the track.

Hence, I continued improve the image augmentation with the left and right cameras. The simulator captured images from three cameras mounted on the car: a center, right and left camera to overcome the issue of recovering from being off-center. I randomly selected one image from three camera and added steering correction if the left or right image was selected as in the lecture. This kind of image augmentation was applied in the training phase only. Image from center camera always selected in the validation phase.

Here is images from the center, left, and right cameras:







I added a ModelCheckpoint() object to save all the model weights after each epoch of training and tested all models in the simulator with autonomous mode. The ideal number of epochs was 10 epochs as evidenced by learning curve plotted at the end of training.

At the end of the process, the vehicle is able to drive autonomously around the track one without leaving the road.

I also tested the same network with adding a Lambda layer for resizing the images to the size of 66 pixel height x 200 pixel width like NVIDIA network. At the epoch = 10, the vehicle is able to drive autonomously around the track one but sometimes it went off the track a little bit. So I went with the network without resizing image.

Code Scripts

In [1]:
```
import os
import csv
import cv2
import numpy as np
import sklearn
```

Image Augmentation

In [2]:
```
def flip_image(img, angle):
    """
    Randomly flip the image and adjust the steering angle.
    """
    if np.random.rand() < 0.5:
        img = cv2.flip(img, 1)
        angle = -angle
    return img, angle
```

Reading and spliting data

In [3]:
```
# Read the driving_log.csv and get paths of images as samples
samples = []
with open('../../P3_Data/driving_log.csv') as csvfile:
    reader = csv.reader(csvfile)
    for line in reader:
        samples.append(line)

from sklearn.model_selection import train_test_split
train_samples, validation_samples = train_test_split(samples, test_size=0.2, random_state=0)
```

Using generator function to compile and train the model

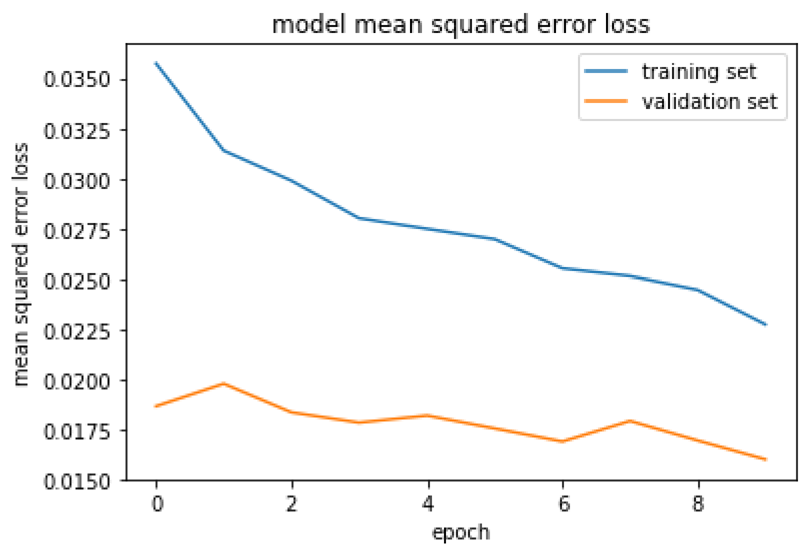
In [4]:
```python
def select_image(batch_sample, is_training=False):
    """
    Randomly select an image among the center, left or right images, and adjust the steering angle.
    This way, we can teach your model how to steer if the car drifts off to the left or the right.
    """
    if is_training == True:
        choice = np.random.choice(3)
    else:
        choice = 0

    name = '../../P3_Data/IMG/'+batch_sample[choice].split('/')[-1]
    image = cv2.imread(name)
    steering_center = float(batch_sample[3])

    # create adjusted steering measurements for the side camera images
    correction = 0.2 # this is a parameter to tune
    steering_left = steering_center + correction
    steering_right = steering_center - correction

    if choice == 0:
        return image, steering_center
    elif choice == 1:
        return image, steering_left
    return image, steering_right

def generator(samples, batch_size=32, is_training=False):
    num_samples = len(samples)
    while 1: # Loop forever so the generator never terminates shuffle(samples)
        for offset in range(0, num_samples, batch_size):
            batch_samples = samples[offset:offset+batch_size]

            images = []
            angles = []
            for batch_sample in batch_samples:
                image, angle = select_image(batch_sample, is_training=is_training)
                images.append(image)
                angles.append(angle)

            # Get training data
            X_train = np.array(images)
            y_train = np.array(angles)

            # Randomly flip image if in training mode
            if is_training == True:
                X_train_augmented, y_train_augmented = [], []
                for x, y in zip(X_train, y_train):
                    x_augmented, y_augmented = flip_image(x, y)
                    X_train_augmented.append(x_augmented)
                    y_train_augmented.append(y_augmented)

                X_train_augmented = np.array(X_train_augmented)
                y_train_augmented = np.array(y_train_augmented)

                yield sklearn.utils.shuffle(X_train_augmented, y_train_augmented)

            else:
                yield sklearn.utils.shuffle(X_train, y_train)

# compile and train the model using the generator function
train_generator = generator(train_samples, batch_size=32, is_training=True)
validation_generator = generator(validation_samples, batch_size=32, is_training=False)
```

Train the network

In [5]:
```
# Build network architecture
# for a regression network (need only 1 neuron at output)
from keras.models import Sequential
from keras.layers.core import Flatten, Dense, Dropout, Lambda
from keras.layers.convolutional import Convolution2D, Cropping2D
from keras.optimizers import Adam
import matplotlib.pyplot as plt
from keras.backend import tf as ktf
from keras.callbacks import ModelCheckpoint

row, col, ch = 160, 320, 3  # image format
input_shape = (row,col,ch)

def resize(image):
    from keras.backend import tf as ktf
    resized = ktf.image.resize_images(image, (66, 200))
    return resized
```

```
# Create the Sequential model
model = Sequential()

## Set up lambda layers for data preprocessing:

# Set up cropping2D layer: cropping (top, bottom) (left, right) pixels
model.add(Cropping2D(cropping=((60,25), (0,0)), input_shape=input_shape))

# Add Lambda layer for resizing image (image, height, width, data_format)
#model.add(Lambda(resize, input_shape=(75, 320, 3), output_shape=(66, 200, 3)))

# Add Lambda layer for normalization
model.add(Lambda(lambda x: (x / 127.5) - 1.0))

## Build a Multi-layer feedforward neural network with Keras here.

# 1st Layer - Add a convolution layer
model.add(Convolution2D(24, 5, 5, subsample=(2,2), activation='relu'))

# 2nd Layer - Add a convolution layer
model.add(Convolution2D(36, 5, 5, subsample=(2,2), activation='relu'))

# 3rd Layer - Add a convolution layer
model.add(Convolution2D(48, 5, 5, subsample=(2,2), activation='relu'))

# 4th Layer - Add a convolution layer
model.add(Convolution2D(64, 3, 3, activation='relu'))

# 5th Layer - Add a convolution layer
model.add(Convolution2D(64, 3, 3, activation='relu'))

# 6th Layer - Add a convolution layer
model.add(Dropout(0.5))

# 7th Layer - Add a flatten layer
model.add(Flatten())

# 8th Layer - Add a fully connected layer
model.add(Dense(100, activation='relu'))

# 9th Layer - Add a fully connected layer
model.add(Dense(50, activation='relu'))

# 10th Layer - Add a fully connected layer
model.add(Dense(10, activation='relu'))

# 11th Layer - Add a fully connected layer
model.add(Dense(1))

model.summary()

# saves the model weights after each epoch if the validation loss decreased
checkpointer = ModelCheckpoint('model-{epoch:02d}.h5',
                                 monitor='val_loss',
                                 verbose=0,
                                 save_best_only=False,
                                 mode='auto')
# Compile and train the model
model.compile(optimizer='adam', loss='mse', verbose = 1)
# history_object = model.fit(X_train, y_train, validation_split=0.2, shuffle=True, nb_epoch=7, batch_size=12
8)
history_object = model.fit_generator(train_generator, samples_per_epoch=len(train_samples), \
                                     validation_data=validation_generator, nb_val_samples=len(validation_samp
les), \
                                     nb_epoch=10, callbacks=[checkpointer], verbose=1)

### print the keys contained in the history object
print(history_object.history.keys())

### plot the training and validation loss for each epoch
plt.plot(history_object.history['loss'])
plt.plot(history_object.history['val_loss'])
plt.title('model mean squared error loss')
plt.ylabel('mean squared error loss')
plt.xlabel('epoch')
plt.legend(['training set', 'validation set'], loc='upper right')
plt.show()
```

```
Using TensorFlow backend.

____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
cropping2d_1 (Cropping2D)        (None, 75, 320, 3)    0           cropping2d_input_1[0][0]
____________________________________________________________________________________________________
lambda_1 (Lambda)                (None, 75, 320, 3)    0           cropping2d_1[0][0]
____________________________________________________________________________________________________
convolution2d_1 (Convolution2D)  (None, 36, 158, 24)   1824        lambda_1[0][0]
____________________________________________________________________________________________________
convolution2d_2 (Convolution2D)  (None, 16, 77, 36)    21636       convolution2d_1[0][0]
____________________________________________________________________________________________________
convolution2d_3 (Convolution2D)  (None, 6, 37, 48)     43248       convolution2d_2[0][0]
____________________________________________________________________________________________________
convolution2d_4 (Convolution2D)  (None, 4, 35, 64)     27712       convolution2d_3[0][0]
____________________________________________________________________________________________________
convolution2d_5 (Convolution2D)  (None, 2, 33, 64)     36928       convolution2d_4[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 2, 33, 64)     0           convolution2d_5[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)              (None, 4224)          0           dropout_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 100)           422500      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: 559,419
Trainable params: 559,419
Non-trainable params: 0
____________________________________________________________________________________________________
/Users/phuongpham/anaconda3/anaconda/envs/carnd-term1/lib/python3.5/site-packages/keras/backend/tensorflow_b
ackend.py:1917: UserWarning: Expected no kwargs, you passed 1
kwargs passed to function are ignored with Tensorflow backend
  warnings.warn('\n'.join(msg))

Epoch 1/10
9653/9653 [==============================] - 145s - loss: 0.0357 - val_loss: 0.0187
Epoch 2/10
9653/9653 [==============================] - 147s - loss: 0.0314 - val_loss: 0.0198
Epoch 3/10
9653/9653 [==============================] - 150s - loss: 0.0299 - val_loss: 0.0184
Epoch 4/10
9653/9653 [==============================] - 146s - loss: 0.0280 - val_loss: 0.0179
Epoch 5/10
9653/9653 [==============================] - 148s - loss: 0.0275 - val_loss: 0.0182
Epoch 6/10
9653/9653 [==============================] - 148s - loss: 0.0270 - val_loss: 0.0176
Epoch 7/10
9653/9653 [==============================] - 153s - loss: 0.0255 - val_loss: 0.0169
Epoch 8/10
9653/9653 [==============================] - 156s - loss: 0.0252 - val_loss: 0.0179
Epoch 9/10
9653/9653 [==============================] - 157s - loss: 0.0245 - val_loss: 0.0170
Epoch 10/10
9653/9653 [==============================] - 155s - loss: 0.0228 - val_loss: 0.0160
dict_keys(['val_loss', 'loss'])
```


model mean squared error loss
training set
validation set
mean squared error loss
epoch