Behavioral Cloning

Overview

In this Udacity's Carnd-term1 behavioral cloning project, I used what you've learned about deep neural networks and convolutional neural networks to clone driving behavior. I will train, validate and test a model using Keras.

The model will learn form labeled pictures of lack track and output a steering angle to an autonomous vehicle. It is a supervised regression problem between the car steering angles and the road images in front of a car. We use images from center, left and right cameras simultaneously.

The architecture I used is based on the architecture published in NVIDIA's "End to End Learning for Self-Driving Cars" paper, which has been proven to work well in this kind of project and computational efficient.

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
- **model.h5** containing a trained convolution neural network
- **writeup_report.md**/writeup_report.pdf summarizing the results

2. Submission includes functional code

Using the Udacity provided simulator and drive.py file, the car can be driven autonomously around the track by executing

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

- Udacity Self-Driving Car Simulator: https://github.com/udacity/self-driving-car-sim

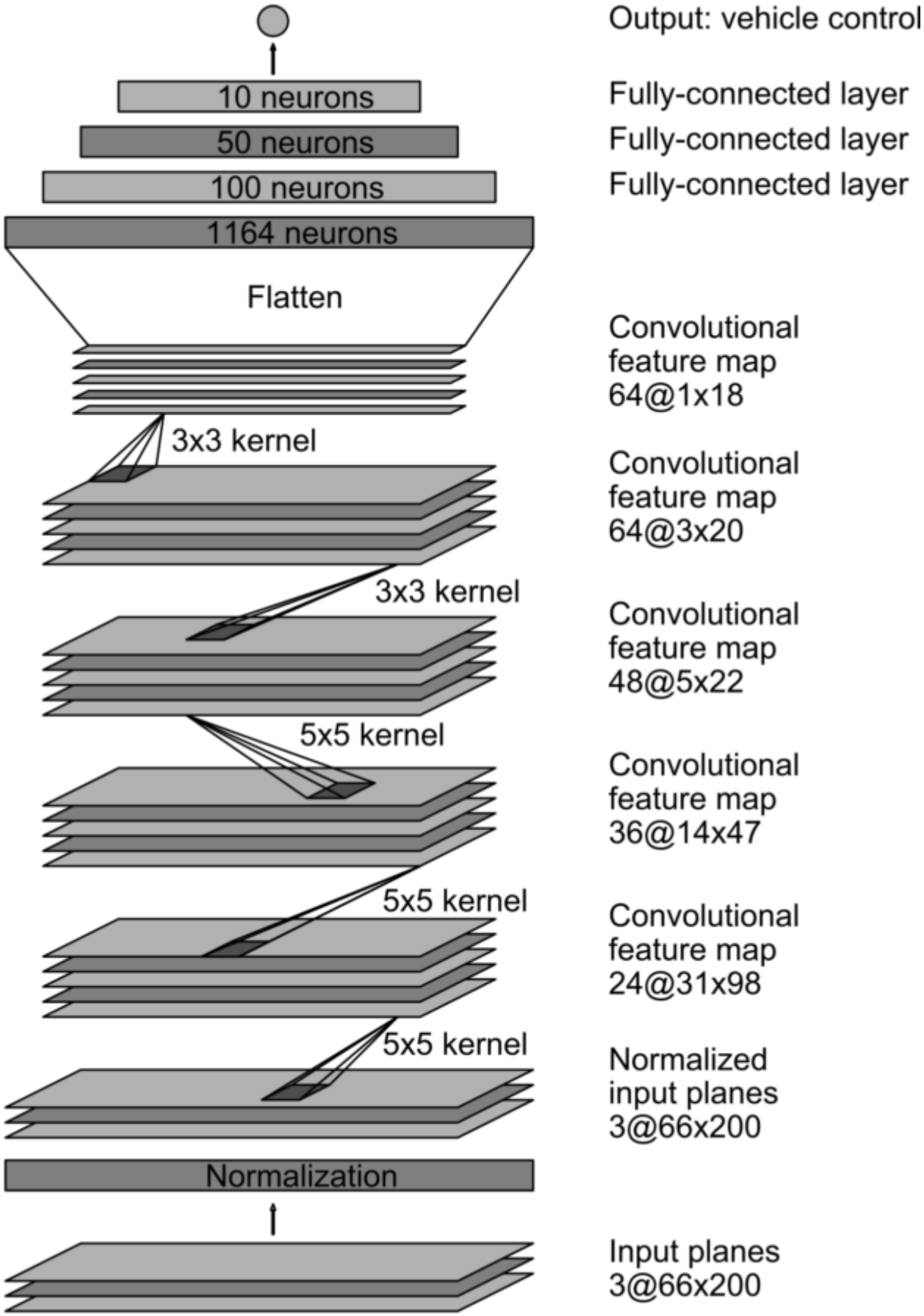
3. Submission code is usable and readable

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. Model architecture

My model is based on the architecture published in NVIDIA's "End to End Learning for Self-Driving Cars" paper. NVIDIA's network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers. Following picture is this CNN architecture



It is a **deep convolutional neural network** which has proved works well with end-to-end self-driving car problems, I just make use of it to solve my problem and fine tuning the model to avoid overfitting and adding non-linearity to improve the prediction.

As the NVIDIA model has done, the input images is **normalized** in the model using a Keras **lambda layer**. Additionally, I add a **cropping layer**, choosing area of interest that excludes the sky(Upper 60 pixels) & the hood of the car(20 pixels), this will help accelerate the training process.

```
#Data Preprocessing: Use a lambda layer to normalize input images
model.add(Lambda(lambda x: x/255.0-0.5, input_shape = (160,320,3)))
#Cropping Images: adding a cropping layer, choosing area of interest that excludes the sky & the
hood of the car.
model.add(Cropping2D(cropping=((60,25), (0,0))))
```

2. Attempts to reduce overfitting in the model

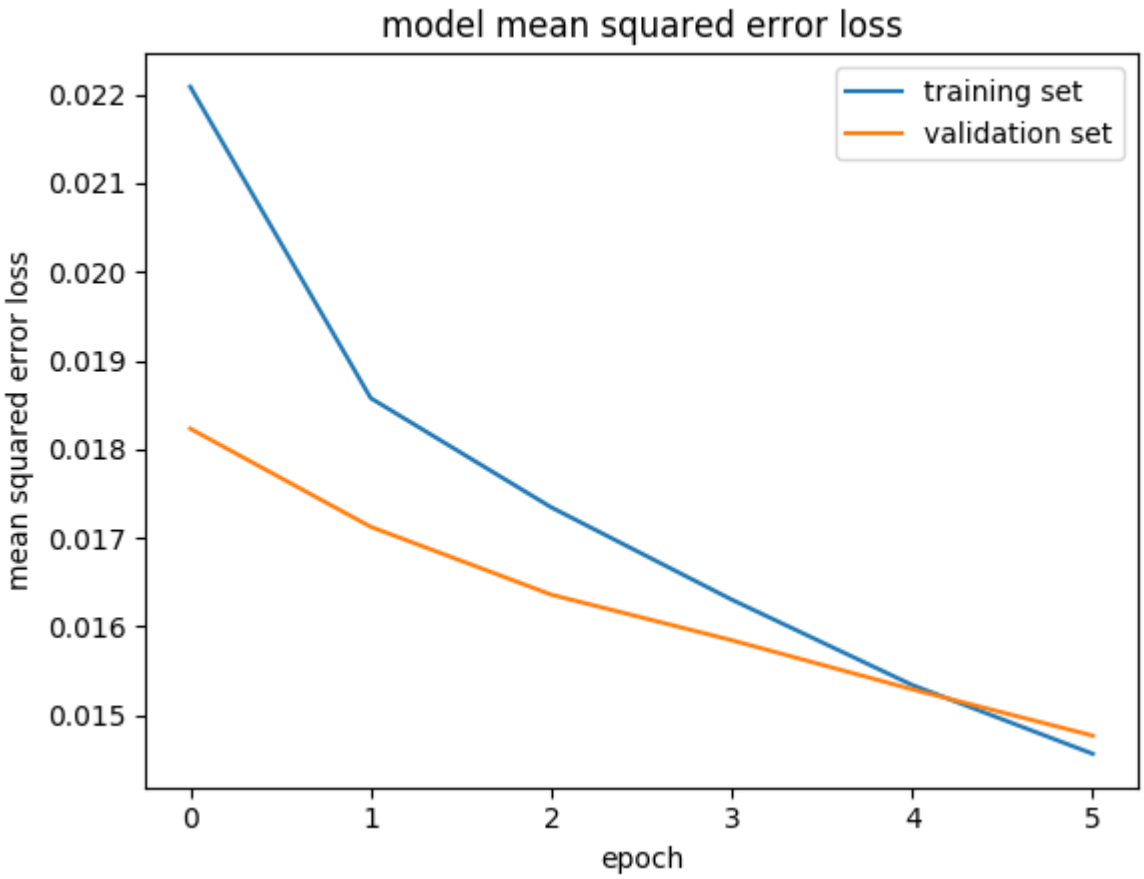
The model contains 1 **dropout layers**(between convolutional layer and Full-Connected layer) in order to reduce overfitting.

The model was **trained and validated on different data sets** to ensure that the model was not overfitting (code below). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

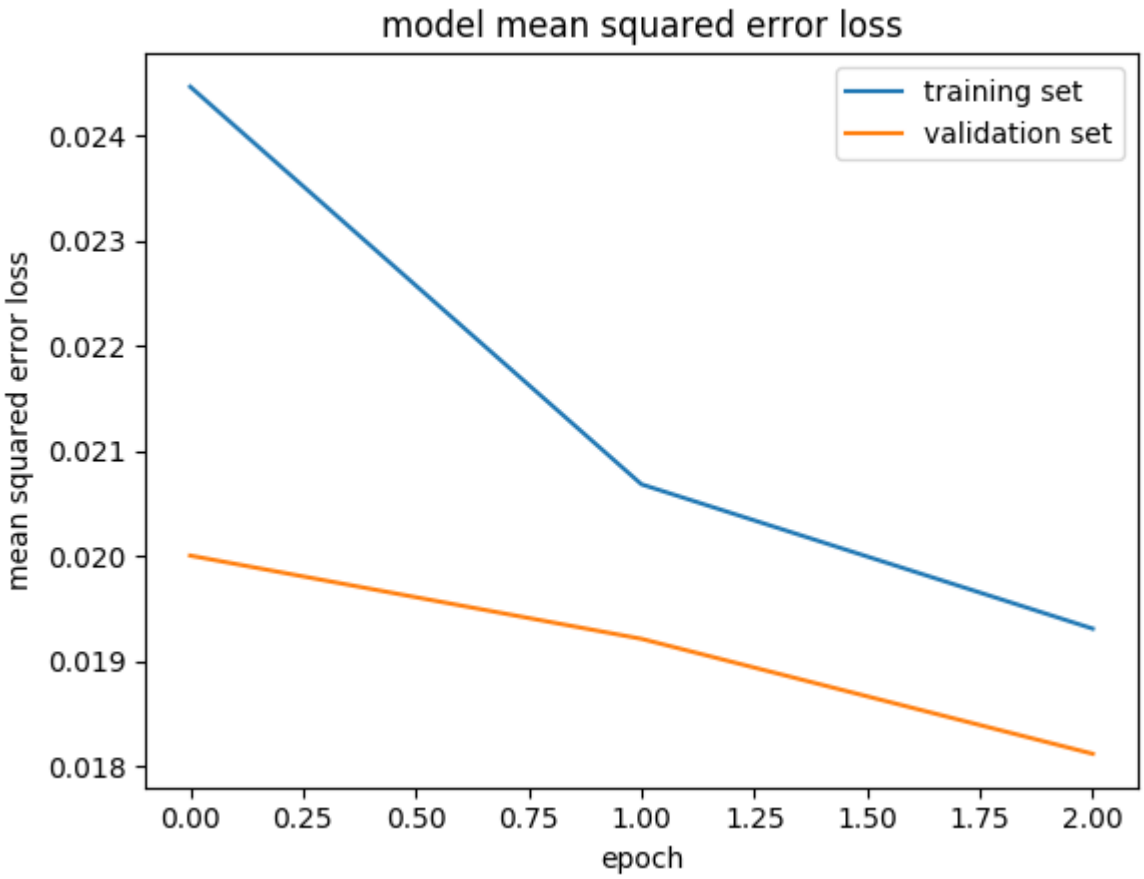
```
from sklearn.model_selection import train_test_split
train_samples, validation_samples = train_test_split(samples, test_size=0.2)
```

When training the model, I find that number of epochs to training the model at 3~5 will acquire good performance when testing on simulator. what's more, when testing on simulator, model with **epochs=3** perform stable than epochs=6 which with low training/validation loss, so it's better to **early stop training** to reduce overfitting. I think the reason is that the measurement angles are sparse(Keyboard input), a little underfitting with low variance will help the self-driving car drive more smoothly.

Following is the visualization of mean squared error loss.



"MSE" Loss with Sample Training Data



Final "MSE" Loss with Added Training Data

3. Model parameter tuning

The model used an adam optimizer(use mean squre error "mse" to evaluate it's loss). In practise I follow the rule that always lower the learning rate(here I set the learning rate at 1e-4) and find it's convergence slightly faster than default adam optimizer.

For dropout layer, I just set the keep probability keep_prob = 0.5, a common used value. It is used to combat overfitting and not influence much if I stop training early.

4. Final Model Architecture

The final model architecture consisted of a convolution neural network with the following layers and layer sizes:

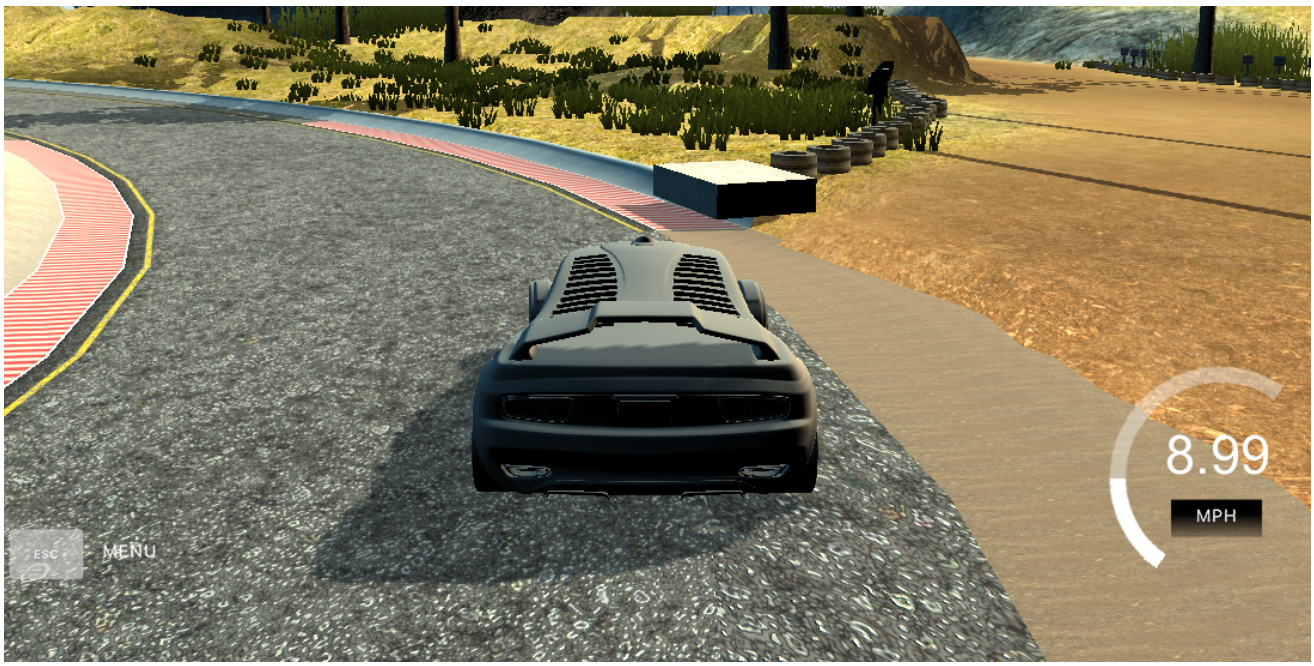
Layer	Description
Input	160x320x3 RGB image
Lambda	lambda x: x/255.0-0.5, normalize input images
Cropping2D	Crop upper 60 pixels and lower 25 pixels
Convolutional layer (Conv2D)	24 5×5 filters, subsample: 2x2, activation: RELU
Convolutional layer (Conv2D)	36 5×5 filters, subsample: 2x2, activation: RELU
Convolutional layer (Conv2D)	48 5×5 filters, subsample: 2x2, activation: RELU
Convolutional layer (Conv2D)	64 3×3 filters, activation: RELU
Convolutional layer (Conv2D)	64 3×3 filters, activation: RELU
Flatten	Transform into 1-D tensor (Transition from convolutional layer to fully-connected layer)
Dropout layer	Avoid overfitting, keep_prob = 0.5
Fully-connected layer(Dense)	100 filters
Fully-connected layer(Dense)	50 filters
Fully-connected layer(Dense)	10 filters
Fully-connected layer(Dense)	output

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Lambda, Cropping2D, Dropout
from keras.layers.convolutional import Convolution2D
from keras.optimizers import Adam

model = Sequential()
#Data Preprocessing: Use a lambda layer to normalize input images
model.add(Lambda(lambda x: x/255.0-0.5, input_shape = (160,320,3)))
#Cropping Images: adding a cropping layer, choosing area of interest that excludes the sky & the hood of the car.

model.add(Cropping2D(cropping=((60,25), (0,0))))
```

```
model.add(Convolution2D(24, 5, 5, subsample=(2, 2), activation='relu'))
model.add(Convolution2D(36, 5, 5, subsample=(2, 2), activation='relu'))
model.add(Convolution2D(48, 5, 5, subsample=(2, 2), activation='relu'))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(Flatten())
# Dropout layer with keep_prob = 0.5
model.add(Dropout(keep_prob))
model.add(Dense(100))
model.add(Dense(50))
model.add(Dense(10))
model.add(Dense(1))
```

5. Appropriate training data

Training data is the most important part of modeling a end-to-end self-driving car. At the begining of construct my model, Sample Training Data was used. However it failed to turn the following curve and collide with sideline, so I add data of this curve and it perform better.



To keep the vehicle driving on the road, images from center, left and right cameras are also used simultaneously.

- For left image, steering angle is adjusted by +0.2
- For right image, steering angle is adjusted by -0.2

I did use some image augumentation and image enhancement technology in the begining, but it turns out improve little, and computational expensive(I have no GPU to train the model), so I just use the simplest method, add more data with the simulator to augment the dataset.

6. Compile and training the model

Following well commented codes is used for training my model.

```
# Compile the model, use "mse"(mean squre error) as loss function to indicate how well the model
predicts to the given steering angle for each image.
# Use Adam optimizer with lower learning rate 1e-4 rather than default 1e-3
model.compile(loss='mse', optimizer=Adam(lr=learning_rate))
#Traing the model
# model.fit(X_train,y_train, validation_split=0.2, shuffle=True, nb_epoch=epochs)
history_object = model.fit_generator(train_generator, samples_per_epoch=len(train_samples)*3,
                    validation_data=validation_generator,
nb_val_samples=len(validation_samples)*3, nb_epoch=epochs, verbose=1)
#Save the model I have trained.
model.save("model.h5")
```

At the begining, I feed in all preprocessed data into memory all at once, whenever I operate my computer, especially watch vedio, it get stucked and result in memory exception. Use a **generator** to load data and preprocess it on the fly, in batch size portions to feed into Behavioral Cloning model is crucial important. Here I use the `fit_generator` function provided by Keras.

The final step is **training the model**. I set the epochs=3, and it's good enough to produce a well trained model for the lake side track. Also, I set batch_size=16 (multiply by 3 becaause I use center, left and right images), and It works well on my computer(Not watch vedios at the same time!).

Conclusions

In this Behavioral Cloning Project,

1. We build up a **deep convolutional neural network** to work on the regression problem in the context of self-driving cars;
2. Use a generator to feed batch training data into Behavioral Cloning model;
3. Reduce overfitting in the model through add dropout layer and early stop training, fine tuning model parameters;
4. Collect data of good driving behavior;
5. Compile and training the model, and then test the model on Udacity's simulator.

The result shows that this model works well on the Lake track. It's to capable of driving in Jungle track since I haven't collect it's Jungle dataset(Hardwork for Keyboard to collect dataset)