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 or writeup_report.pdf summarizing the results
- environment_behavior.yml for anaconda environment used for the project

2. Submission includes functional code

The submission includes model.py and drive.py as code files. Model.py includes everything needed to train the model in my computer. I had to create an environment which uses tensorflow version 2.3. I

had to update to the latest version to utilize the GPU available in my computer.

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 only modification done to drive.py is to increase the vehicle set speed to 20mph from 9mph initial setting.

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

My model follows the NVIDIA model https://developer.nvidia.com/blog/deep-learning-self-driving-cars/. The input to the model is the 160x320x3 images that are acquired from the simulator provided by Udacity. The output of the model is the *steering angle* to be applied to the vehicle on the simulator.

The first layer of the model is a normalization layer applied via Lambda layer (model.py line 117). This normalization will be applied to each layer in the image, which represents Red, Green and Blue channels. Normalization layer is followed by a Cropping2D layer (model.py line 118), which crops top and bottom of the image.

The model is continued with convolutional layers with 3x3 filter sizes and depths of 24, 36, 48, 64 and 64 (model.py lines 119-124). The first 3 convolution layers have strides of 2 on each direction, and the last 2 layers have strides of 1. To introduce nonlinearity, I set activation of each layer to relu following the course material; however, this wasn't specified into NVIDIA article.

After the convolutions, model has a Flatten layer (model.py line 125), and 4 fully connected layers with 100, 50, 10 and 1 neurons respectively, which last neuron represents the steering output.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 160, 320, 3)       0
_________________________________________________________________
cropping2d (Cropping2D)      (None, 90, 320, 3)        0
_________________________________________________________________
conv2d (Conv2D)              (None, 43, 158, 24)       1824
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 77, 36)        21636
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 37, 48)         43248
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 6, 35, 64)         27712
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 33, 64)         36928
_________________________________________________________________
flatten (Flatten)            (None, 8448)              0
_________________________________________________________________
dense (Dense)                (None, 100)               844900
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_2 (Dense)              (None, 10)                510
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 11
=================================================================
Total params: 981,819
Trainable params: 981,819
Non-trainable params: 0
```

2. Attempts to reduce overfitting in the model

To reduce overfitting, I trained the model on both center, left, and right images of the car with a steering correction. In addition to these, I also added flipped images to the training and validation dataset. Without the flipped images, the vehicle would go off the road regularly, via either applying too much steering or too less steering, see image below.

Finally, the model was tested by running it through the simulator and ensuring that the vehicle could stay on the track. I tried this with multiple speeds as well; however, the steering oscillation increases significantly as the speed increases.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 131). The other parameter that can be tuned was the correction applied to steering measurements for left and right camera images. Here, I chose `0.2`, which was also the default value that was selected in the coursework (model.py line 85).

4. Appropriate training data

As mentioned above, I used all of center, left and right camera images for my training data. The goal of the project was to keep the car within the boundaries. In order to do this, I drove one lap counter-clockwise direction, and one lap clockwise direction, while trying to keep the vehicle centered in these runs. I also did a few recovery runs; however, the model was already performing well, so I kept these to a minimum.

Model Architecture and Training Strategy

1. Solution Design Approach

I initially followed the steps suggested by Udacity, where the first step was to deploy a very simple model. As expected, this model wasn't able to keep the vehicle in lane in the curves, and had high loss. Then, I continued with the NVIDIA model, which was also suggested in the class. I also read the NVIDIA article, and they initially used a simulator to evaluate their model before going to the vehicle. Therefore, I thought it fit this application well.

In addition to the normalization layer, I added a cropping layer to remove the unnecessary information that the camera sees, similar to the masking we applied in the first and second projects of the Nanodegree.

I must note that due to the limited number of scenarios the vehicle faces in the track; I still think this model is a bit of an *overkill*. There are nearly 1 million parameters, which I believe is causing model to memorize the track and overfit. Although the loss values are very low for both training and validation sets, it seems like the model is doing *too much*.

To test the model, I first split the dataset into training and validation using `sklearn.model_selection` library's `train_test_split()` function (model.py line 106). I used mean squared error as the loss (model.py line 131).

While collecting data, I was aware of how model might struggle with training. Therefore, I initially collected training data from clockwise and counter-clockwise driving around the track. While driving, I was trying to be as smooth as possible and avoid sudden changes and too busy steering. However, this was still not enough; as when I run the model on the simulator, it struggled with curves especially at high speeds. Therefore, I recorded some recovery runs. For examples, please see last section of this document.

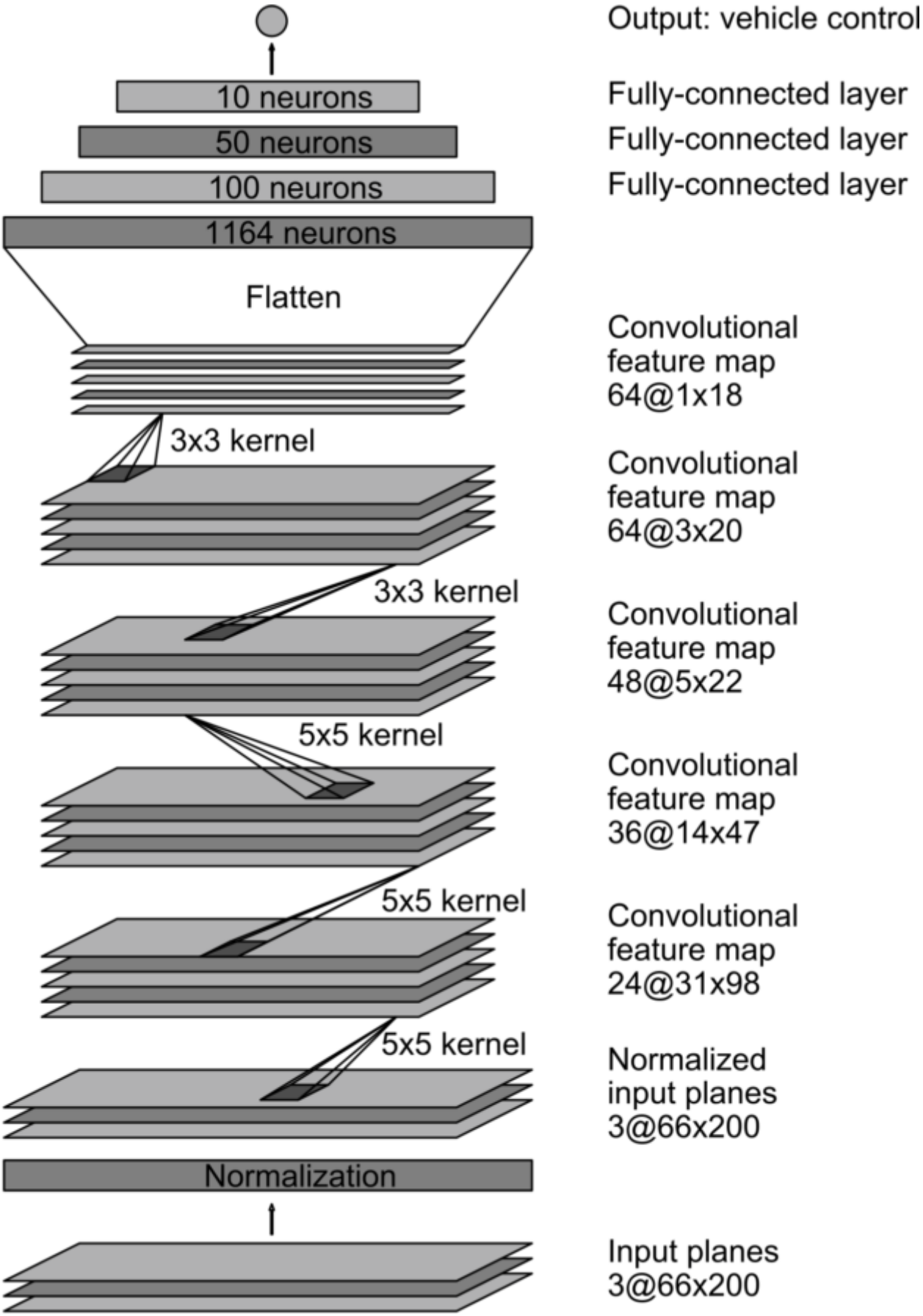
Finally, I ran the model on the autonomous mode and it was able to stay on track. In some situations, it was even impressive that it could have very smooth behavior in the curves.

One other note. In traffic sign classification, we didn't use the test dataset until we were sure with the model to avoid overfitting. However, in this project, running model gave useful feedback as the vehicle did went out of the road (see image above). This was a dilemma for me; however, as the model is intended to only run on this track, I still moved forward with testing the model more frequently, than just once.

2. Final Model Architecture

For final model architecture please see sections above which displays the summary from the console.

An image of the model architecture from NVIDIA website is below.



3. Creation of the Training Set & Training Process

As mentioned above, to create the dataset, I did two laps around the track, one counter-clockwise and the other clockwise. Within these laps, I tried to center the vehicle and keep the steering as smooth as possible. Here is an example image of center lane driving from each direction of track.

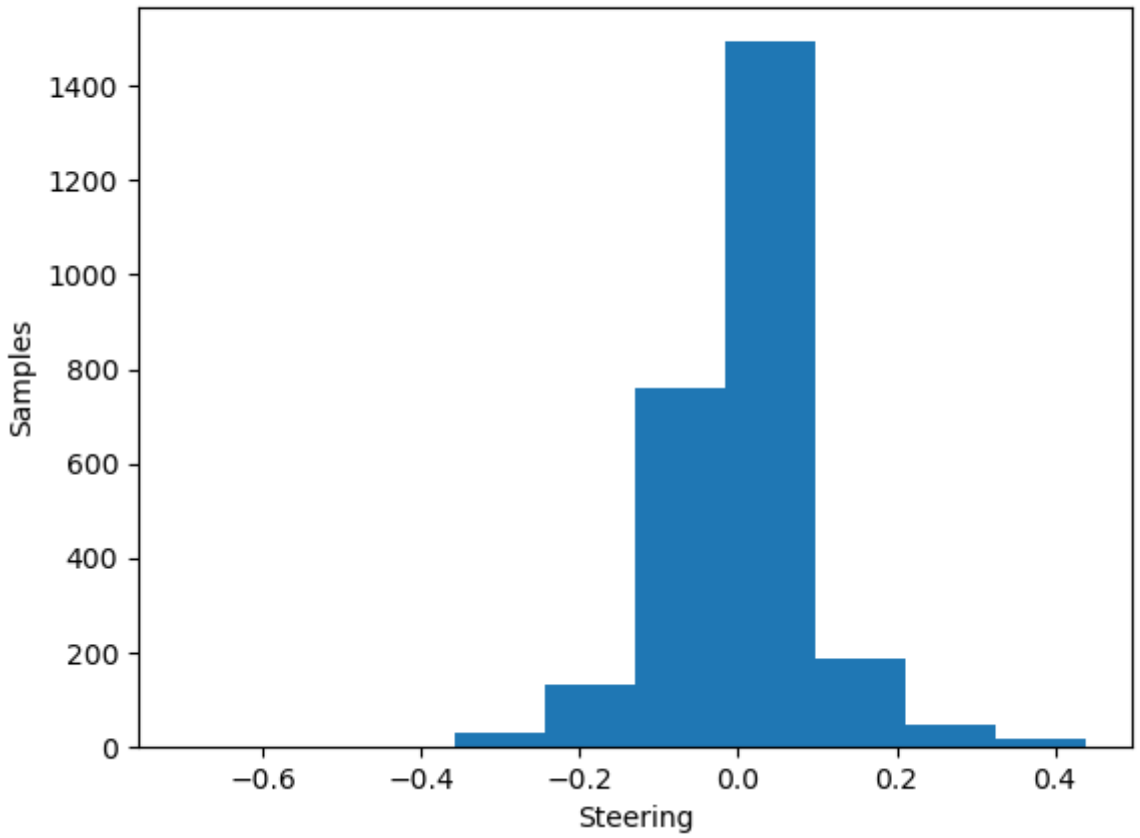
Clockwise:

Counter clockwise:

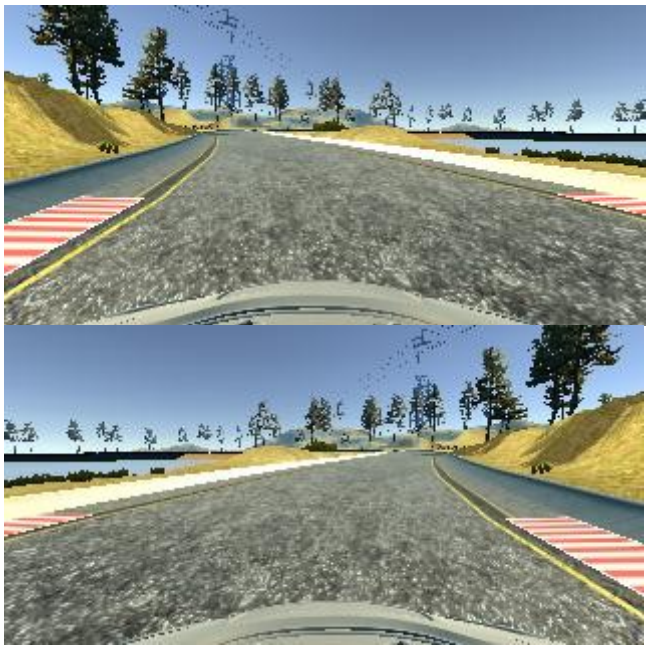
Then, the model was struggling a bit, so I also recorded recovery scenarios. In these scenarios, I drove the vehicle to the edge of the road, and then started recording as I manuevered the vehicle back to center. Here are some examples for recoveries.

The initial dataset had 2664 data points. Below are the speed, and steering distributions for the dataset.

	Steering	Throttle	Brake	Speed
count	2664.000000	2664.000000	2664.0	2664.000000
mean	-0.000330	0.850313	0.0	29.178026
std	0.090430	0.335313	0.0	3.554774
min	-0.700000	0.000000	0.0	0.000002
25%	-0.038123	1.000000	0.0	30.189030
50%	0.000000	1.000000	0.0	30.190210
75%	0.032258	1.000000	0.0	30.190300
max	0.439883	1.000000	0.0	30.269300



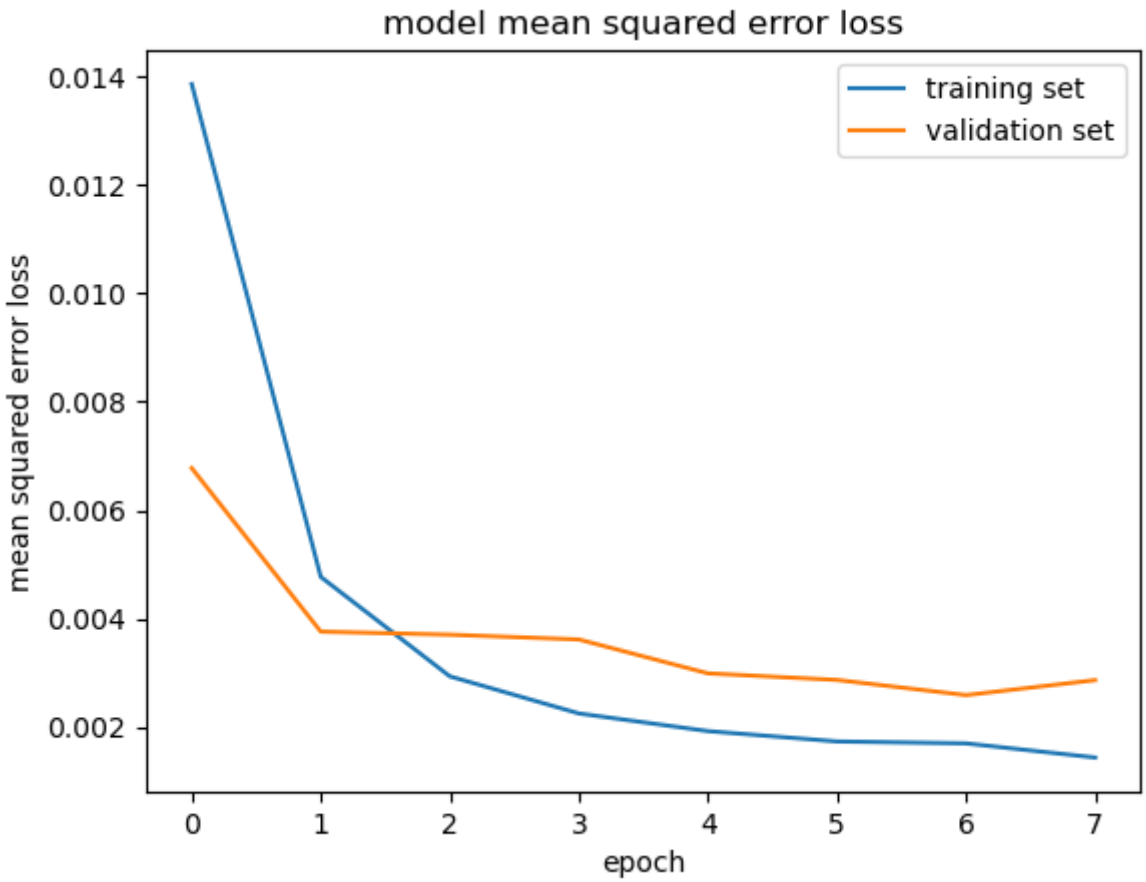
To increase the number of images to be trained on, I used left and right images and applied a correction factor of `0.2` to the steering. This multipled the dataset size by 3. For further data augmentaiton, I flipped images and angles for the whole dataset, which essentially doubled the dataset. Here is an example for an original, and flipped image.



The final dataset had ... data points after the augmentations. This data has been preprocessed by the normalization layer, which puts the image layers between `-0.5` and `0.5`, and a cropping layer, which crops `50 pixels` from top of the image, and `20 pixels` from bottom of the image.

The dataset is split to training and validation sets, with 20% of data being in the validation set (model.py line 106); and shuffled within the generator before batches are created (model.py line 23). Then, after a batch is created, the batch is shuffled again within the generator (model.py line 51)

I trained the model with 10 epochs; but I created callbacks for `EarlyStopping` and `ModelCheckpoint`. I set the `patience` value for `EarlyStopping` to 1 epoch. In my training, the model stopped earlier than 10 epochs as the validation loss didn't improve further.




```
400/400 [==============================] - 26s 65ms/step - loss: 0.0139 - val_loss: 0.0068
Epoch 2/10
400/400 [==============================] - 25s 62ms/step - loss: 0.0048 - val_loss: 0.0038
Epoch 3/10
400/400 [==============================] - 25s 62ms/step - loss: 0.0029 - val_loss: 0.0037
Epoch 4/10
400/400 [==============================] - 25s 62ms/step - loss: 0.0023 - val_loss: 0.0036
Epoch 5/10
400/400 [==============================] - 25s 62ms/step - loss: 0.0019 - val_loss: 0.0030
Epoch 6/10
400/400 [==============================] - 24s 61ms/step - loss: 0.0017 - val_loss: 0.0029
Epoch 7/10
400/400 [==============================] - 25s 62ms/step - loss: 0.0017 - val_loss: 0.0026
Epoch 8/10
400/400 [==============================] - 21s 53ms/step - loss: 0.0014 - val_loss: 0.0029
```