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 [unchanged]
- model.h5 containing a trained convolution neural network
- writeup_report.pdf summarizing the results
- video.mp4 containing a video of the camera output while the car is driven around the tracks one after the other by the trained network

2. Submission includes functional code

Using the Udacity-provided simulator and the 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. The file shows the pipeline I used for training and validating the model, and it is organized in a way that it is easy to read. In my opinion, comments are not really necessary as the code is short and self-explanatory, and is formatted in a way such that different concerns are separated and easy to distinguish.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of a convolution neural network with 5 convolutional layers and 4 fully connected layers. Here is a description of the model:

Layer	**Description**
Cropping	Top 70 pixels and bottom 25 pixels shaved off
RGB-to-grayscale conversion	
Normalization	Range -1 to +1
Convolutional	5x5 filter, 24 channels
RELU	
Dropout	Drop probability of 0.25
Convolutional	5x5 filter, 36 channels
RELU	
Dropout	Drop probability of 0.25
Convolutional	5x5 filter, 48 channels
RELU	
Dropout	Drop probability of 0.25
Convolutional	3x3 filter, 64 channels
RELU	
Dropout	Drop probability of 0.25
Convolutional	3x3 filter, 64 channels
RELU	
Dropout	Drop probability of 0.25
Fully connected	Output size of 100
Fully connected	Output size of 50
Fully connected	Output size of 10
Fully connected	Output size of 1

The model uses a cropping layer to remove the top and bottom portions of the images which are not of interest since they contain landscape (top) and the front portion of the car (bottom).

The model uses Keras lambda layers to convert the images to grayscale, as well as to normalize the pixel values. These steps have been done within the model since they need to be done during testing as well as when the model is running on live images.

The model includes RELU layers after each convolutional layer to introduce nonlinearity.

2. Attempts to reduce overfitting in the model

The model contains dropout layers after each convolutional layer in order to reduce overfitting. The dropout rate is fixed at 25% for each dropout layer, which means that, randomly, 25% of the nodes are dropped in each dropout layer.

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 17). 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 uses an adam optimizer, so the learning rate was not tuned manually (model.py line 67).

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 and driving around the tracks in both the clock-wise and the anti-clockwise directions to provide sufficient opportunity to the model to learn good driving behavior as well as to allow it to learn how to recover from non-ideal positions. I trained the model on examples from both the tracks so that it can better learn how to generalize the learnings on any track.

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 make sure that the network had a sufficient amount of non-linearity for the complexity of the learning task at hand. Overall, I understood that a powerful network such as the NVIDIA network discussed in the final sections of the project lessons would work well, as long as the data it was learning from was good enough.

My first step was to go through project lessons to get familiar with the procedure to be used for model training and for running the model on the simulator. I first used the basic network discussed in the lessons, then started using LeNet, and finally switched to the NVIDIA architecture discussed in the final sections of the project lessons. Most of the NVIDIA architecture has been preserved in the final model I have created.

While experimenting with LeNet, I realized that I needed to collect enough examples of good driving as well as recovering behavior in order to make this work. While the network must be powerful enough to generalize driving behavior on at least the two tracks that were provided, the real challenge was to collect the right data in order to make it work.

Ironically, I was able to get good performance on track 2 before I could make my model work well with track 1. I realized that this was because of the quality of the data collected. On track 2, I was forced to drive slower so as to avoid falling off the track, and this led to more precise steering while I was driving. On the track 1, I could stay on the track while driving at the top speed of more than 30 mph, but this led to poorer quality of driving data.

Having realized the importance of good driving data, I proceeded to collect data using precision driving, i.e., driving slow enough to be able to make all turns smooth and precise. I drove precisely for a single lap in both directions on track 1, and used the resulting data to train the network. My model performed much better, but there were some spots where the vehicle was falling off the track.

To improve driving behavior in these cases, I recorded another lap full of examples of recovering from the left and from the right. I did this using very precise driving as well. At the end of this process, my model was able to drive without falling off the track on track 1. I was also able to get good results while letting the network drive the car in the opposite direction.

I then repeated the same process on track 2, and was finally able to get good results on this track as well. It was more challenging to record recovery behavior for track 2, especially in spots where the slope was too steep to demonstrate precise recovery behavior.

2. Final Model Architecture

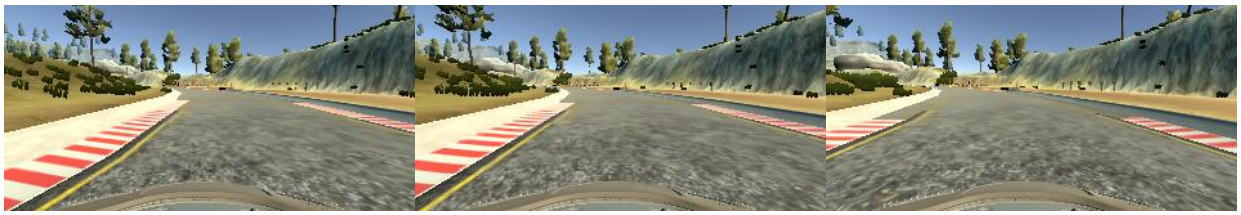
The final model architecture has been described above in the section on Model Architecture. More or less, I used the NVIDIA architecture that was described in the later lessons with just a few notable changes:

1. Addition of a pre-processing layer for conversion to grayscale to make it easier for the network to learn
2. Addition of dropout layers after each convolutional layer to combat overfitting

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:

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 to recover in case it starts falling to either side of the road. These images show what a recovery looks like starting from the left:



Then I repeated this process on track two in order to get more data points.

After the collection process, I had 43888 data points. I did not perform any stand-alone pre-processing on this data as all of my pre-processing steps [cropping, grayscale conversion and normalization] were a part of the model itself.

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 around 27 as evidenced by the training logs and the training and validation loss after each epoch [see below]. I used an adam optimizer so that manually training the learning rate wasn't necessary.

Training Logs:

```
Epoch 1/30
43888/43888 [==============================] - 106s - loss: 0.0454 - val_loss: 0.0262
Epoch 2/30
43888/43888 [==============================] - 104s - loss: 0.0270 - val_loss: 0.0228
Epoch 3/30
43888/43888 [==============================] - 104s - loss: 0.0231 - val_loss: 0.0191
Epoch 4/30
43888/43888 [==============================] - 103s - loss: 0.0209 - val_loss: 0.0170
Epoch 5/30
43888/43888 [==============================] - 105s - loss: 0.0194 - val_loss: 0.0156
Epoch 6/30
43888/43888 [==============================] - 105s - loss: 0.0180 - val_loss: 0.0154
```

```
Epoch 7/30
43888/43888 [==============================] - 104s - loss: 0.0174 - val_loss: 0.0135
Epoch 8/30
43888/43888 [==============================] - 103s - loss: 0.0167 - val_loss: 0.0132
Epoch 9/30
43888/43888 [==============================] - 103s - loss: 0.0159 - val_loss: 0.0143
Epoch 10/30
43888/43888 [==============================] - 103s - loss: 0.0157 - val_loss: 0.0130
Epoch 11/30
43888/43888 [==============================] - 104s - loss: 0.0151 - val_loss: 0.0116
Epoch 12/30
43888/43888 [==============================] - 103s - loss: 0.0148 - val_loss: 0.0117
Epoch 13/30
43888/43888 [==============================] - 100s - loss: 0.0142 - val_loss: 0.0120
Epoch 14/30
43888/43888 [==============================] - 102s - loss: 0.0140 - val_loss: 0.0110
Epoch 15/30
43888/43888 [==============================] - 100s - loss: 0.0135 - val_loss: 0.0112
Epoch 16/30
43888/43888 [==============================] - 101s - loss: 0.0134 - val_loss: 0.0105
Epoch 17/30
43888/43888 [==============================] - 104s - loss: 0.0131 - val_loss: 0.0102
Epoch 18/30
43888/43888 [==============================] - 103s - loss: 0.0128 - val_loss: 0.0101
Epoch 19/30
43888/43888 [==============================] - 103s - loss: 0.0127 - val_loss: 0.0102
Epoch 20/30
43888/43888 [==============================] - 102s - loss: 0.0125 - val_loss: 0.0097
Epoch 21/30
43888/43888 [==============================] - 104s - loss: 0.0122 - val_loss: 0.0107
Epoch 22/30
43888/43888 [==============================] - 105s - loss: 0.0122 - val_loss: 0.0102
Epoch 23/30
43888/43888 [==============================] - 102s - loss: 0.0117 - val_loss: 0.0108
Epoch 24/30
43888/43888 [==============================] - 101s - loss: 0.0118 - val_loss: 0.0108
Epoch 25/30
43888/43888 [==============================] - 103s - loss: 0.0117 - val_loss: 0.0099
Epoch 26/30
43888/43888 [==============================] - 103s - loss: 0.0115 - val_loss: 0.0092
Epoch 27/30
43888/43888 [==============================] - 103s - loss: 0.0114 - val_loss: 0.0090
Epoch 28/30
43888/43888 [==============================] - 102s - loss: 0.0112 - val_loss: 0.0095
Epoch 29/30
43888/43888 [==============================] - 101s - loss: 0.0110 - val_loss: 0.0093
Epoch 30/30
43888/43888 [==============================] - 103s - loss: 0.0109 - val_loss: 0.0095
```