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

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
- video.mp4 - recording of the vehicle driving autonomously two laps around the track
- 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

```
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 contains comments to explain how the code works.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed.

My model consists of:
a) Preprocessing within the model ("on the fly"): image resizing, cropping and data normalization. Each convolutional layer is followed by a 2x2 max pooling layer.
b) Two convolutional layers with filter sizes 5x5 and 3x3 and depths 32 and 64
c) Three fully connected layers with sizes 128, 64, 32 + final output layer

The model includes RELU layers to introduce nonlinearity. I have also tested ELU activation function, but my model performed worse with it.

I have tested different architectures, starting with LeNet and up to NVIDIA. Finally I came up with my own approach, which is something between LeNet and the NVIDIA model.

My final model consisted of the following layers:

Layer	Description
Input	160 x 320 x 3 RGB image
Lambda resize	Resizing the image to 30 x 60 x 3
Cropping	Cropping top 9 and bottom 3 pixels
Lambda normalize	Data normalization to zero mean
Convolution	Depth 32, filter 5x5
Max pooling	2x2 stride
RELU	
Convolution	Depth 64, filter 3x3
Max pooling	2x2 stride
RELU	
Flatten	
Fully connected	Output = 128
RELU	Keep probability = 80%
Fully connected	Output = 64
RELU	
Fully connected	Output = 32
RELU	
Output layer	Output = 1 (predicted steering angle)

2. Attempts to reduce overfitting in the model

The model contains max pooling layers in order to reduce overfitting. I have also tested the usage of dropout, but with dropout the model's performance deteriorated.

The model was trained and validated on different data sets to ensure that the model was not overfitting (test / validation split).

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.

4. Appropriate training data

I'm using a combination of Udacity's "sample training data" and additional "recovery driving" data recorded by myself. For details about how I created the training data, see the next section.

5. Solution design approach

Overall, solution design was a long iterative process, with many "unfruitful" changes and rollbacks to previous architecture and/or data:

a) Architecture tuning

- In the first step I adapted the LeNet architecture for the purposes of this project. I observed quite high training and validation MSEs (in the range of 0.1+, which taking into account that the predictions are between -1 and 1 was quite a high error rate).
- In the second step, I tried the NVIDIA architecture. With this architecture I observed a significant overfitting of the model.
- I tried to combat the overfitting with dropout and even L2 regularization, but in the end just removing a couple of layers altogether yielded the best result.
- In the whole process I tried a lot of different architectures, but in the end it turned out that data preprocessing / augmentation (see below) was more powerful in improving the models performance than architecture fine-tuning.

a) Data preprocessing / augmentation

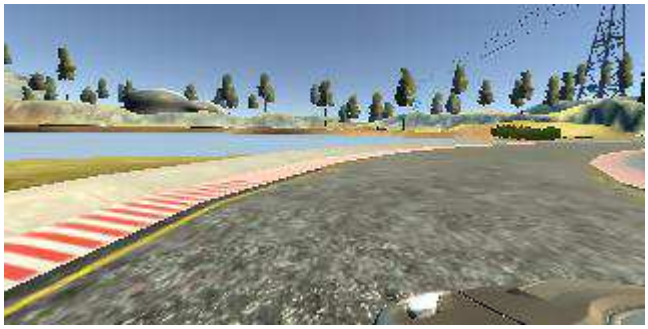
- At the beginning I just used the images from the center camera. With this data the model would get into trouble very quickly, even before the first major turn.
- Adding images from the left and right cameras has significantly improved driving "stability" on the less demanding parts of the track.
- Still, the model was not able to steer the car through the curve after the bridge and the following right turn with water on the left side. In both cases the car would veer of the road.
- Therefore, I recorded additional "recovery" driving data targeted at these "difficult" spots.
- While adding this data allowed the car to pass the difficult turns, it also led to unexpected trouble in other parts of the road. For instance, the car was not able to drive through the bridge - it veered off to the left right before the bridge. It seems to me that the model wrongly recognized the bridge as the dirt after the bridge (similar brown surface and black sides).
- As a result, I decided to record even more "recovery driving" data, this time to make sure that the car enters the bridge successfully.
- After this step, the car finally started driving around the entire track

Examples of dataset images:

Example 1: Center lane driving
- from Udacity's "sample training data" set:

Example 2: Recovery driving dirt after the bridge (image from the right camera)
- from my own additional training data set:

Example 3: Recovery driving sharp left turn (image from the left camera)
- from my own additional training data set:



The car drives successfully around the entire track. On 2 or 3 occassions it briefly starts veering off to one side - but in all these cases it successfully recovers to the middle of the road.

I spent a significant amount of time trying to fine tune the model and data so that the car always stays in the middle of the lane, but I was not able to improve it. Probably it would require many more iterations, but I now need to dedicate my time to the next Udacity SDC projects.