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

Files Submitted & Code Quality

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

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
- model_with_generator.py containing the script to create and train the model using fit generator
- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- model_track1_old.h5 containing an early version of trained convolution neural network
- model_track2.h5 containing a trained convolution neural network for track 2
- this writeup_report.pdf summarising 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.

The model_with_generator.py file contains the code for training and saving the convolution neural network by using Python generator to generate data for training rather than storing the training data in memory. This file is submitted for educational purposes in hope that the reviewers could point me to the implementation errors. The code runs but it takes 10 times longer to train the model by using this approach on my machine. I expect either the problem is with Keras 2.0 API mismatch that I had to use (the script throws warnings about it but I couldn't figure out how to fix that and there are no examples provided for API 2.0) or it is just the price for using less memory.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of a convolution neural network with 3x3 and 5x5 filter sizes and depths between 24 and 64 (model.py lines 58-62)
The model includes RELU layers to introduce nonlinearity (code line 58), and the data is normalized in the model using a Keras lambda layer (code line 56).

2. Attempts to reduce overfitting in the model

The model contains dropout layers in order to reduce overfitting (model.py lines 64-69).
The model was trained and validated on different data sets to ensure that the model was not overfitting. The datasets were collected by appending image paths to the driving_log.csv file. The set used to train the submitted solution contains 17651 triple images which including the augmentation results in 17651*3*2 = 105906 images. 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 (model.py line 73).

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, several runs on problematic curves. I used touchpad for driving which is better than a keyboard but with an analog joystick one could achieve a smoother behaviour.

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 start with the model described in the Nvidia paper. Additionally Paul Heraty created a helpful guide stating that this model works.
My first step was to use the original convolution neural network model described in the paper simply because it was proven that it works and from the previous experience we learned how convolution network are capable of extracting meaningful features and fully connected layers allow us to use those features to achieve the desired behavior.

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 in several ways starting from including dropouts after each layer of the networks. I trained on a small dataset to see how different approaches in combination with different number of epochs improve or degrade the performance. At the end I realised that having dropouts between the convolution layers doesn't help to fight the overfitting and eventually makes the performance poor no matter how much the model was trained. Finally I achieved the best performance by using the dropouts only between the fully connected layers.

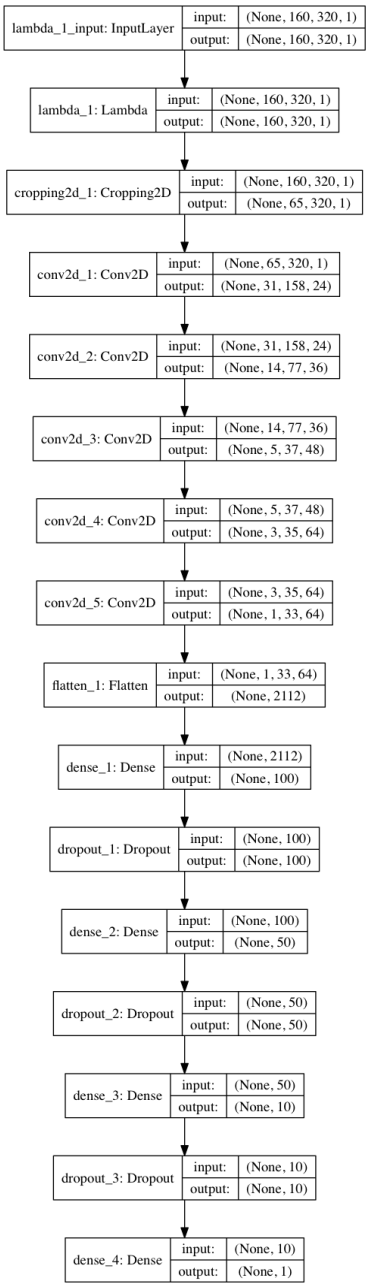
It is important to mention that without dropouts I could achieve already a very smooth behaviour by using only centre camera images and driving clockwise and counter-clockwise on the track. This model is stored in model_track1_old.h5 file with corresponding video run1_old_model.mp4. But when trying to run this model on track two it dramatically failed.

The semi-final step was to run the simulator to see how well the car trained on the final model with dropouts was driving around track two. There were a few spots where the vehicle fell off the track. To improve the driving behaviour in these cases, I added more runs including recovery on those spots.
At the end of the process, the vehicle is able to drive autonomously around the track two without leaving the road. The trained model is stored in file model_track2.h5 with the corresponding video run2.mp4.

At the end to my disappointment or amusement when trying to run the model that was trained on track one and track two datasets in the simulator back on track one, it started driving off-road on one spot where was a passage and then safely returned to the track. To fix the behaviour I had to extend the dataset with more cases from track one including that offload turn. At the end the final trained model started performing well again on track one which is supported by the corresponding video run1.mp4. But running it again on track two created cases when the car was running off the track.

2. Final Model Architecture

The final model architecture (model.py lines 55-70) consisted of a convolution neural network with the following layers and layer sizes:

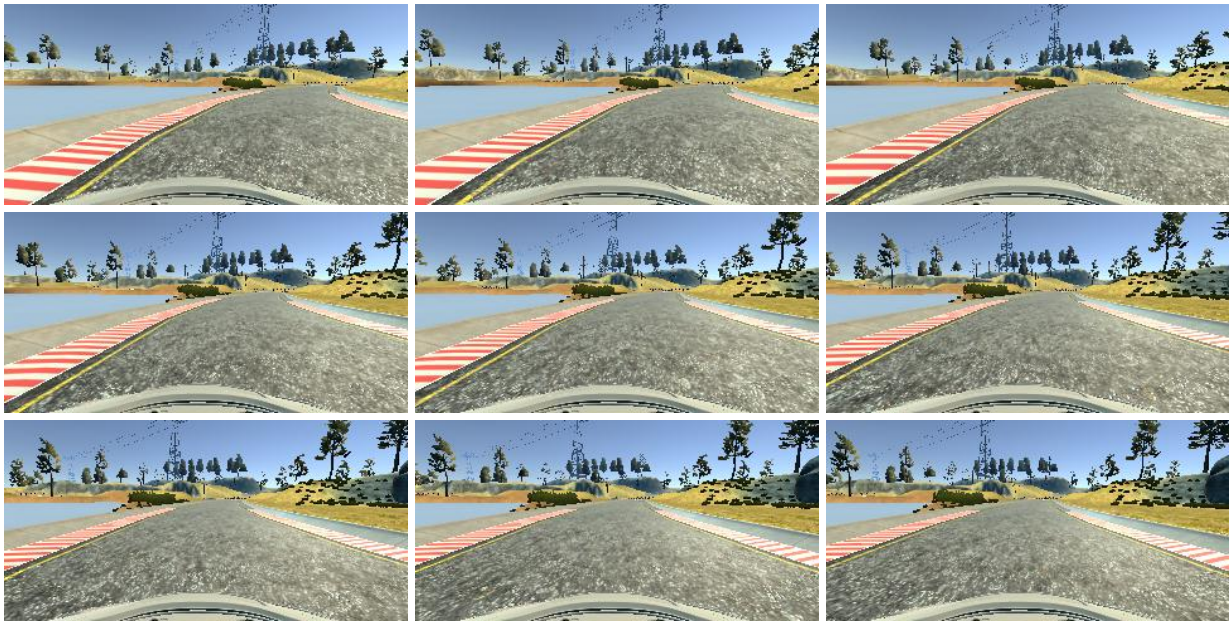


As you can see the pre-processing of the image data is built in into the model that helps us to run the simulator on the trained model without altering the fed to the model images.

3. Creation of the Training Set & Training Process

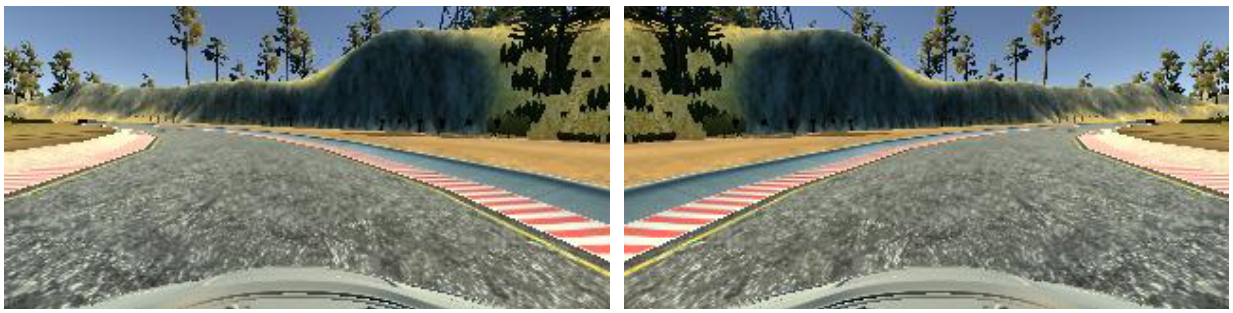
To capture good driving behaviour, I first recorded two laps on track one and later on track two using centre lane driving. Here is an example images of centre lane driving:

I then recorded the vehicle recovering from the left side and right sides of the road back to centre so that the vehicle would learn to drive back to the centre when at the side. These images show what a recovery looks like starting from the side and driving to the centre:



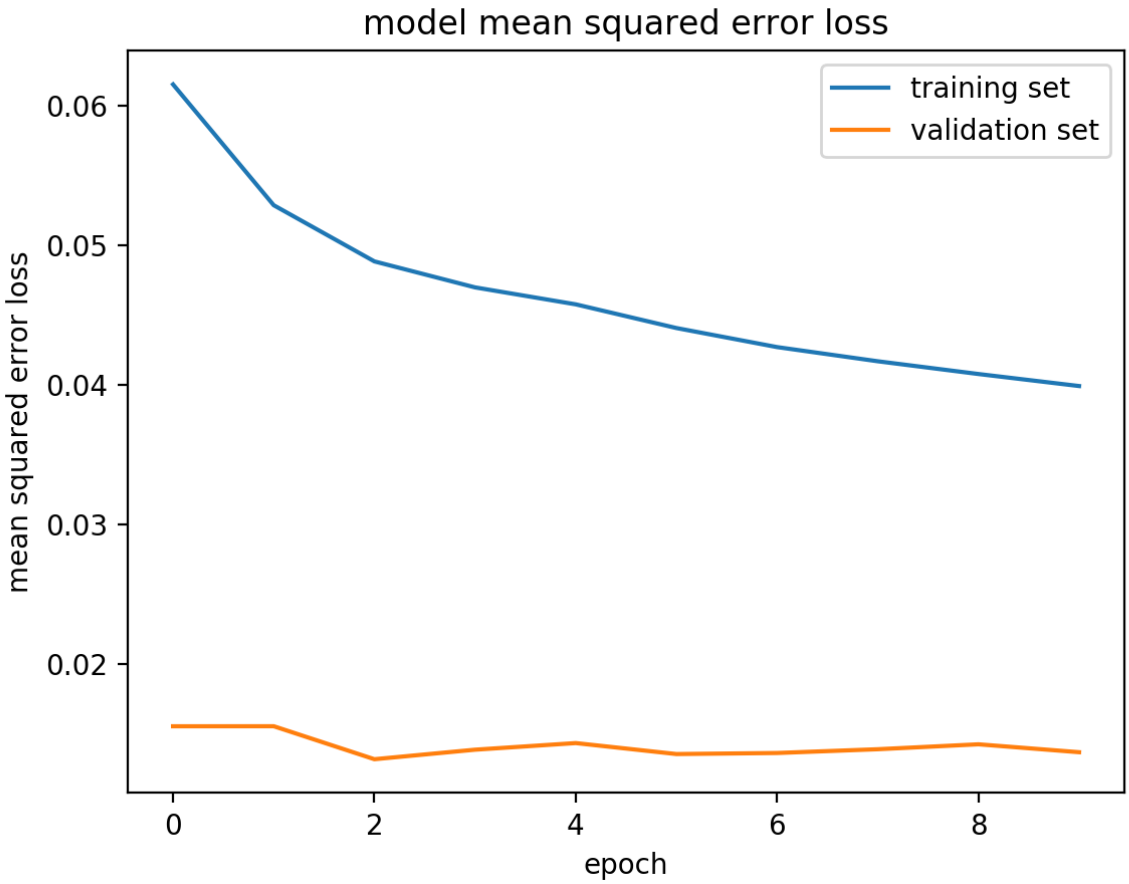
Then I repeated this process on track two in order to get more data points.

To augment the data set, I also flipped images and angles thinking that this would help to prevent from leaning to one side of the road and also to better generalise. For example, here is an image that has then been flipped:

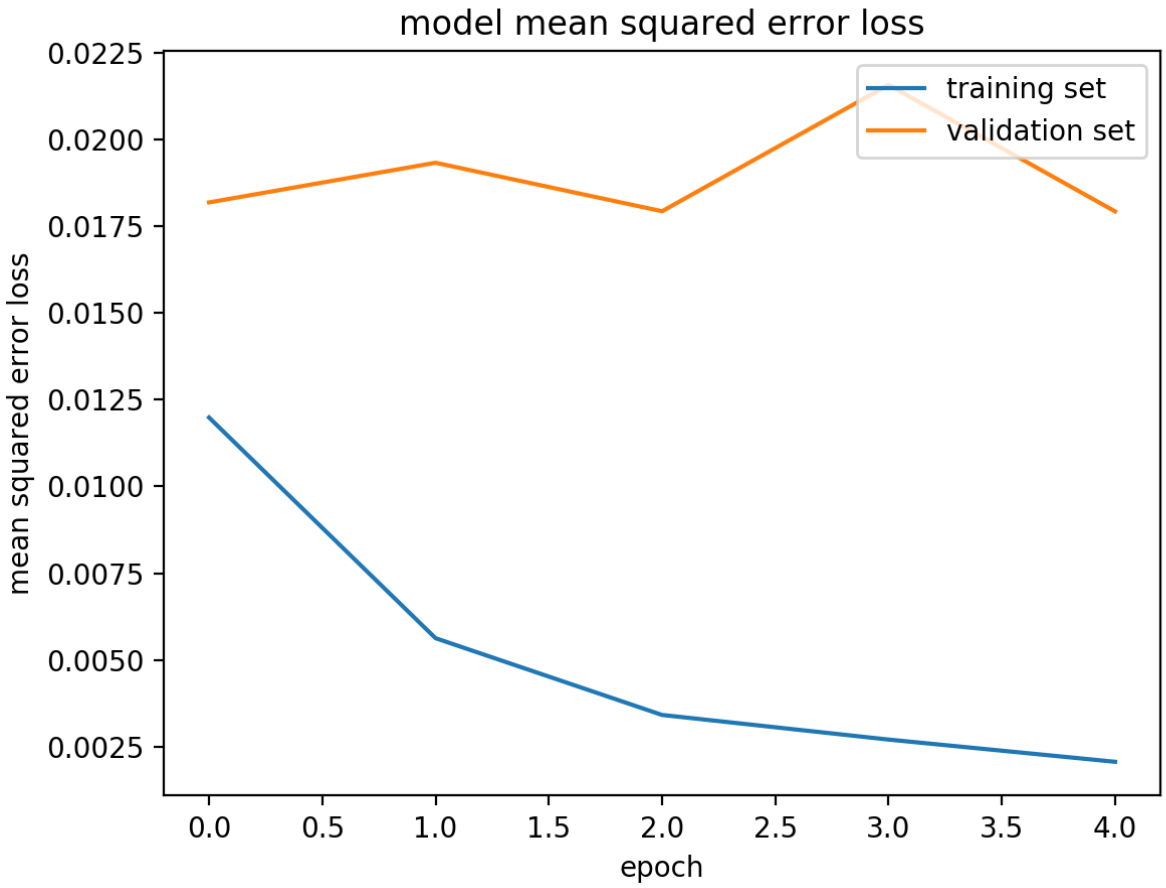


After the collection process, I had 1765 x 3 = 52953 of unique data points and 52953 x 2 = 105906 images after flipping augmentation. I then preprocessed this data by normalising the images in the Lambda layer of the model. I finally randomly shuffled the data set and put 20% of the data into a validation 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 10 as evidenced by the image bellow. I used an Adam optimiser so that manually training the learning rate wasn't necessary.



An example of a poor performance with overfitting can be seen in this image bellow when mean squared error is bouncing and eventually growing on the validation set although is going down on the training set.



In this case one would select 3 epochs as optimum (see image bellow) but unfortunately the test performance in the simulator was still not satisfactory.