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

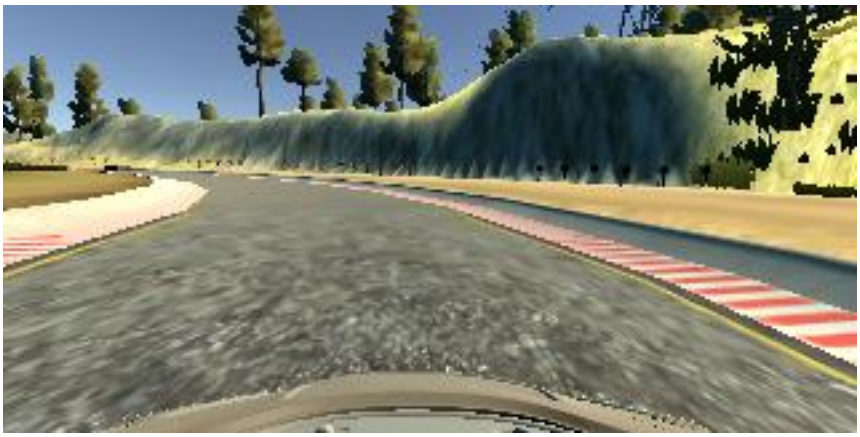


Figure 1: A sample input image

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.md or writeup.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

I based my model on the NVIDIA pipeline.

At first, the input images needed to be converted to RGB, since the openCV function reads in the images with BGR and the simulation returns the images in RGB.

Second the input data with the shape (160,320,3) is normalized with zero mean and lies between +- 0.5.

As a third step I cropped the images to get rid of the unimportant parts of the the image, like the front hood of the car that the camera shows or the environment. These parts of the pictures are confusing for the training of the model, since there is no correlation or a correlation that I do not want to represent in my model.

My model consists of 5 convolutional layers with 5x5 and 3x3 filters and depths between 3 (RGB input image) and 64 (model.py lines 90-98). After the first three convolutions there is a 2x2 pooling layer. After every convolution there are non-linear relu activation functions.

A flatten layer follows in the sequential model. Since this layer is quite large (64 * 3 * 3 = 567) and I need a measure against overfitting I applied a dropout with 25% dropped units as the next layer.

The next layers consist of three Dense layers with 100, 50, 10 and finally 1 layer height.

The one neuron in the last layer corresponds to the steering angle that the input images will be trained on.

Since the steering angle is a continuous value I did not use any activation function like sigmoid or softmax but just used the output of the neuron as such. This is a regression problem rather than a classification problem.

2. Attempts to reduce overfitting in the model

The model contains one dropout layer after the flatten layer, which is the biggest Dense layer in the network. This way I can reduce overfitting (model.py lines 100).

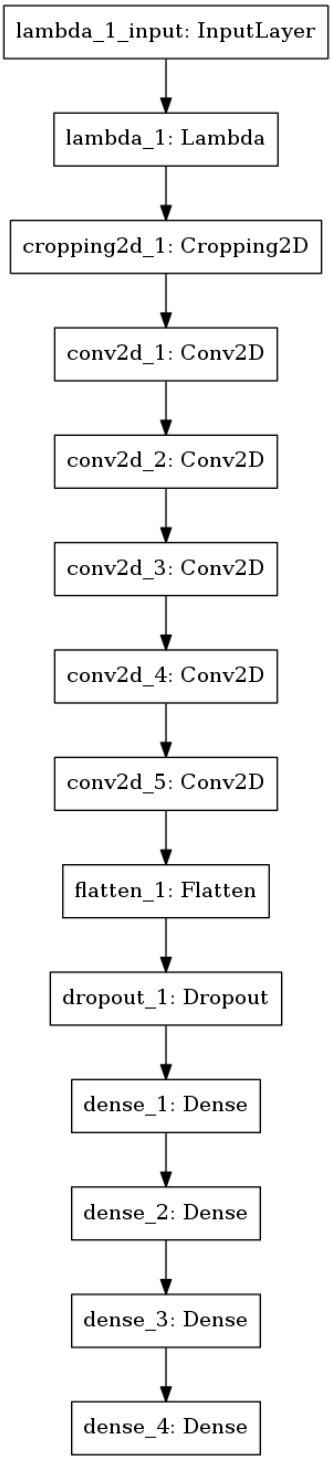


Figure 2: Model Visualization

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 35 and 113-117). 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 109).

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.

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 a simple non-convolutional network with just one input layer and one output neuron for the steering angle. This way I could quickly see if my whole pipeline including the testing with the simulation was working and if the autonomous car was reacting at all. From this start model I was adding two convolutional layers. This seems reasonable to me since convolutional layers are very powerful image perception layers, extracting important image information of the lanes, the road and everything besides the road. Additionally convolutional layers are space invariant, meaning it does not matter where in the image the lanes are for example.

Training this model with the training data while validating it with the validation set (20% of all images) showed that the training and the validation error could be reduced over some few epochs. Since I already implemented the Dropout layer I did overfit the network (low training error but high validation error). Testing it with the simulation gave me already good results. The car stayed on the road for quite a while. Nevertheless it left the road after a while.

One further approach was to improve and enlarge the training set by augmenting images and using the left and right car cameras. I will discuss this in detail in a next section. Another improvement was to crop the images from the car camera as mentioned above.

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

The final model architecture (model.py lines 90 - 104) consisted of a convolution neural network with the following layers and layer sizes:

```
# Keras model
model = Sequential()
# Preprocess incoming data, centered around zero with small standard deviation
model.add(Lambda(lambda x: x/255.0 - 0.5, input_shape=(160,320,3)))
model.add(Cropping2D(cropping=((70,25),(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())
model.add(Dropout(0.25))
model.add(Dense(100))
model.add(Dense(50))
model.add(Dense(10))
model.add(Dense(1))
```

3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps in one direction and two laps in the opposite direction, both with center lane driving. Here is an example image.

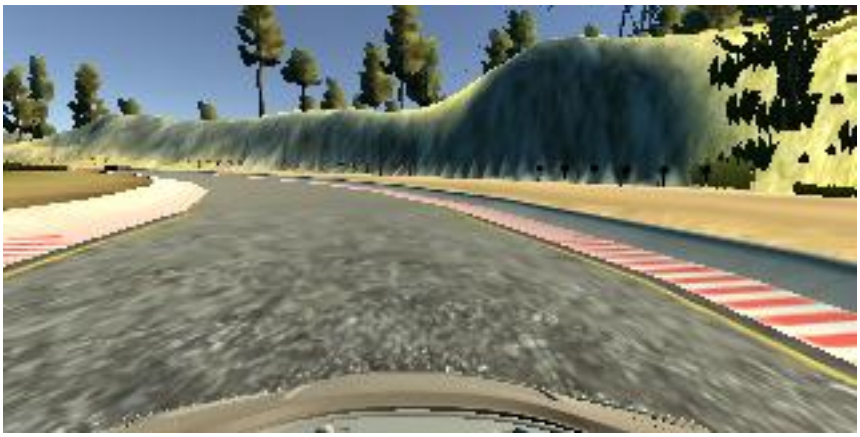
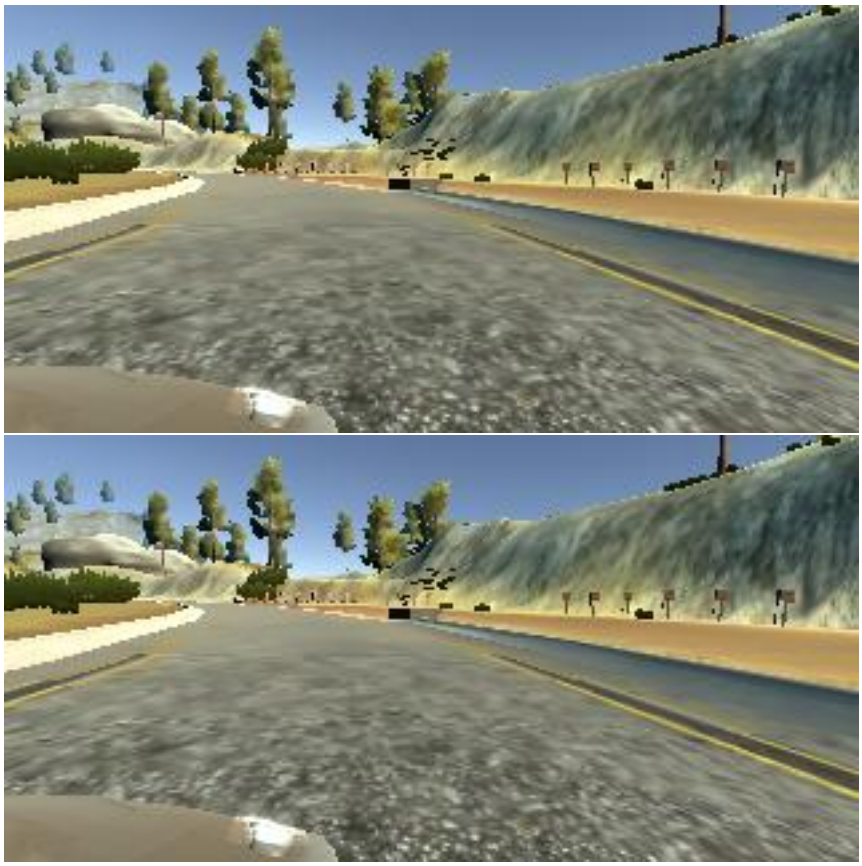
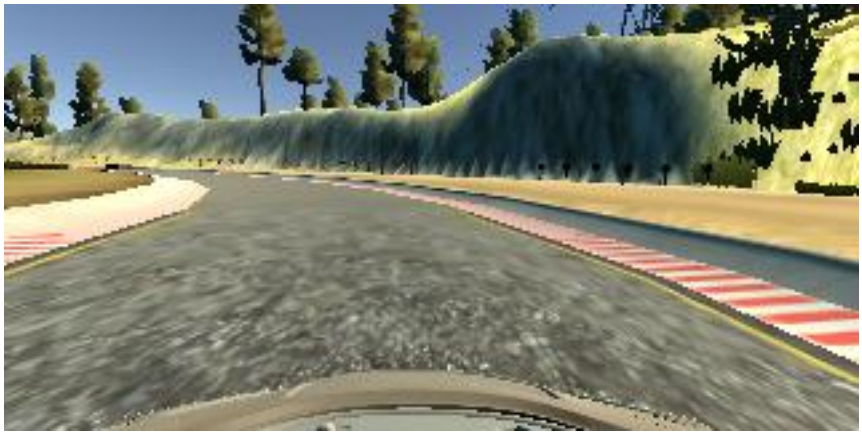


Figure 3: Training image

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 get back to the middle of the road in case it drifts to the side. Here are two examples:



To augment the data set, I also flipped images and angles thinking that this would enrich the training set. For example, here is an image that has then been flipped:



After the collection process, I had 10294 number of data points. I then preprocessed this data by the steps mentioned above (BGR to RGB, normalization, cropping)

To reduce the number of images that need to be loaded into GPU memory at one time I used a generator (=input iterator) that only loads/yields the amount of batch_size images into ram for training. To find a good batch_size it is recommended to take multiple of 2 since GPU ram is also almost always a multiple of 2. The concrete number I eventually needed to search by trial and error. 32 turned out to be a bit too high. Batch sizes of 8 or 16 seemed to not overload the Udacity GPU on which I was training.

Here is a shortend part of the implemented generator:

```
# Define the generator function
def generator(samples, batch_size=32, training=True):
    num_samples = len(samples)
    while 1:  # Loop forever so the generator never terminates
        shuffle(samples)
        for offset in range(0, num_samples, batch_size):
            batch_samples = samples[offset:offset+batch_size]

            # 1) Read in images in given batch sizes
            # ...
            # 2) Append images and measurements to lists
            # ...
            # 3) Augment images after all
            # ...
            # 4) Turn into numpy arrays
            # ...
            # 5) Yield shuffled data
            yield shuffle(X_data, y_data)
```

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. A good number of epochs was up to 4 as evidenced by the following picutre. I used an adam optimizer so that manually training the learning rate wasn't necessary.

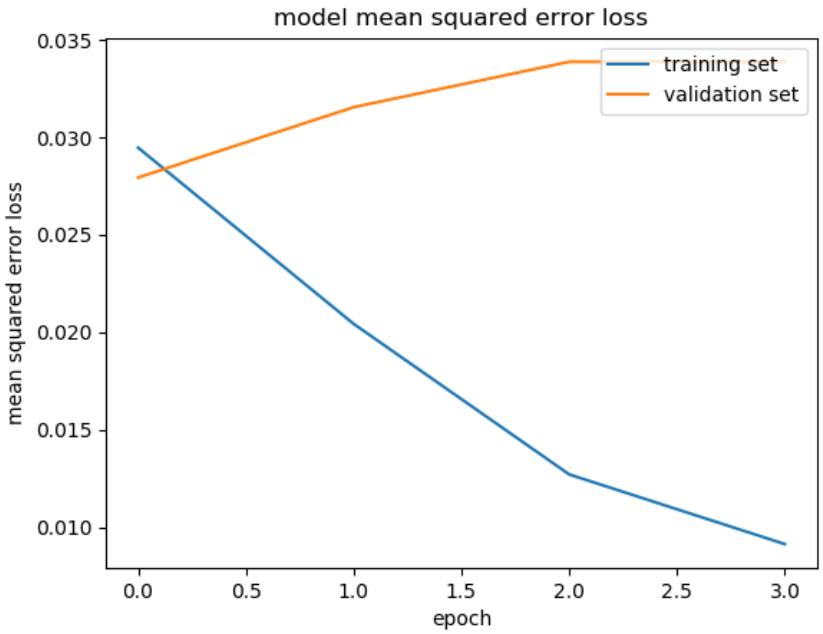


Figure 4: Training and validation history

As can be seen the training error has a big gradient in the beginning which decreases slightly over the first epochs. the validation error goes up very slightly, which is an indication that training should be stopped. Since the validation error only increases very little I used four epochs for the final model.

For further improving the algorithm, more difficult input data like the second track should be used. Also transfer learning would be interesting to try for the tracks and see how well the car does on a an already build net, where only the last classification/regression part is retrained for these images while the first convolutional layers (trained on image net or similar) are freezed.