Behavioral Cloning

Model Architecture and Training Strategy

1. Solution Design Approach

The overall strategy for deriving a model architecture was to have a small model with less data, since I was training on my laptop. I couldn't run the simulator on the workspace and didn't figure out, how to upload my training data to use a gpu. Besides this limitation, the overall running time of my solution is very good and the amount of data for a good model is also not too big.

My first step was to use a convolution neural network model similar to the previous LeNet. I thought this model might be appropriate because it worked for the traffic signs classification. I never tested it, because the simulator api was blocked.. My problem was, that the python process for drive.py was still active in the background, which blocked the communication to the simulator without any error message. Until I figured out, why every model I tested wasn't working, I created more training data and tried another network architecture (NVIDIA). So I debugged a lot with the model until I figured out, that the reason was drive.py still running in the background. So I build my whole model, while I was debugging. Basically I spoiled the fun for struggling with parameter optimisation and architecture prototyping, since my first running model worked from scratch.

My first model was running, but had still a bad architecture. I used the Nvidia network architecture, but I forgot using strides for the kernels. So my model used a lot of training parameters and it needed one whole night for training. But the model was running almost smoothly on the simple track. It touched the track border only once, when the car left the bridge. As a next step, I fixed the architecture to the parameter setup and run it again. The training was done in less than 30 mins on my CPU! and the result was a very nice run on track one. As a next step, I used the same architecture for track two, where I had even much less training data (almost equal in the amount, but the second track is much larger). The result was, that the model also run very smooth. But when I recorded this ride, the car crashed always on against the rock in one of the last curves (very steep in the shadow uphill) in the circuit. Instead of creating more data, I added the two other camera perspectives to the model and run it again. The car runs also very smooth on the second track with less overall data as for model one.

At the end of the process, the vehicle is able to drive autonomously around both tracks without leaving the road. As a further step, I would train the images of both tracks together to achieve one general model, instead of two separate models.

2. Final Model Architecture

For the network architecture I followed the suggested one of Nvidia (https://arxiv.org/abs/1604.07316). This network consists of 9 layers in total, including one normalisation layer, 5 convolutional layer and 5 fully connected layers. (line 70-86 in model.py). The model architecture is the same for both models.

Before the normalisation layer is applied, the image is cropped to remove the horizon in the upper part and the car in the lower part. This also reduces the information send to the network, which speeds up the training process.

The normalisation layer: The batch normalisation sets the values of the input to a mean of 0 with a standard deviation of 1. I could also you a lambda function with a simple normalisation by dividing with 255, but it wouldn't be sure, if the image data really would be "mathematically correct" normalised, so I choose batch normalisation.

The convolutional layers: Following the NVIDIA suggestion, I was surprised in not using MaxPooling Layers. Here every convolutional layer is stacked together and the reduction of the image size is done by the stride of the applied kernel for each layer. Every layer uses a relu activation function and every filter is applied by valid padding. The first three convolutional layers are using a kernel-size of 5x5 with increasing amount of filters (24, 36, 48). The kernel-size for the 4th and 5th cone layer is reduced to 3x3, but increased to 64 filters. For a better understanding of the object sizes after each step, there is the eras visualisation below.

The fully connected layers: The amount of nodes of the first dense layer are the same as the output of the 5th convolutional network (here, an error occured; in the example it should be 1024 and not 1164 like in the paper). The other four layers reduces the amount of nodes to one for the steering decision. For each layer, there is a relu-activation function, while for the single neutron layer a linear activation is chosen. In comparison to the Nvidia architecture, I added dense_layer 1 as an additional layer, which increases the training time.

Hyperparameters: I didn't change much on the hyperparameters. I used the default values. Only the batch_size was increased to 128 images. The number of epochs was set to 20 to enable enough iterations. I mainly rely on the EarlyStopping procedure, which stops the training automatic, when the validation loss won't improve anymore. Therefor I waited for 2 iterations. With introducing a checkpoint, the best model on the validation loss was stored.

batch_size = 128
epochs =20
learning_rate = 0.001
camera_correction = 0.2

Optimizer: I used the AdamOptimizer without any tuning. My approach worked from the beginning, so I didn't saw the need for further improvement here.

Overfitting: Since this model approach doesn't have any dropout layer and rely on the validation set, I used early stopping and checkpoint saving to be not affected by overfitting. So here, I can't prevent overfitting, but I make sure, that the model stops training, when overfitting starts. For testing the model with the simulator I used the model with the smaller validation loss.

Here is a visualization of the architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
cropping2d_1 (Cropping2D)    (None, 80, 320, 3)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 80, 320, 3)        12
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 38, 158, 24)       1824
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 17, 77, 36)        21636
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 37, 48)         43248
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 18, 64)         27712
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 16, 64)         36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1164)              1193100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               116500
_________________________________________________________________
dense_3 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_4 (Dense)              (None, 10)                510
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 11
=================================================================
Total params: 1,446,531
Trainable params: 1,446,525
Non-trainable params: 6
_________________________________________________________________
```

3. Creation of the Training Set & Training Process

For the first model I used more data than for the second model. For both I used two laps, where the car drove in the mid of the lane. Afterwards, I took another two laps in the opposite direction of the lap, also keeping the car in the mid of the lane.

As described earlier, I had problems with running the simulator. Until I figured out this problem, I added more training data for the first model by driving 2 laps in curly lines. With driving those curly lines, I made sure, that I never touch the curbs or limitation of the track.

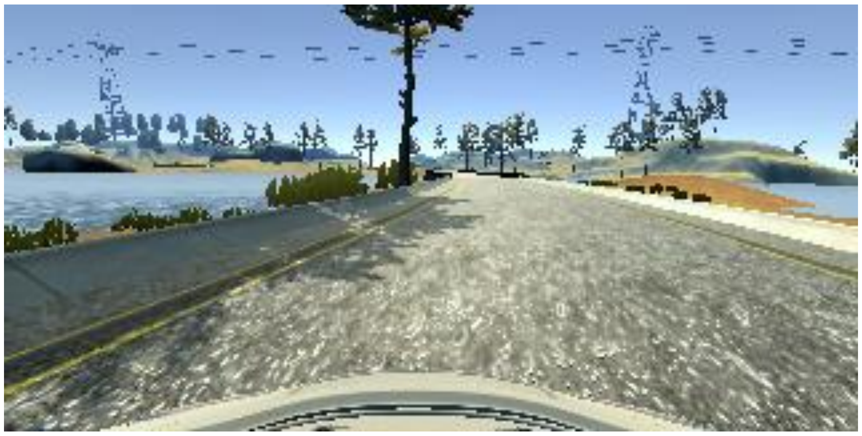
For the first model, I used only the centre image of the camera, while for the second model I used all three cameras. The number of images is:

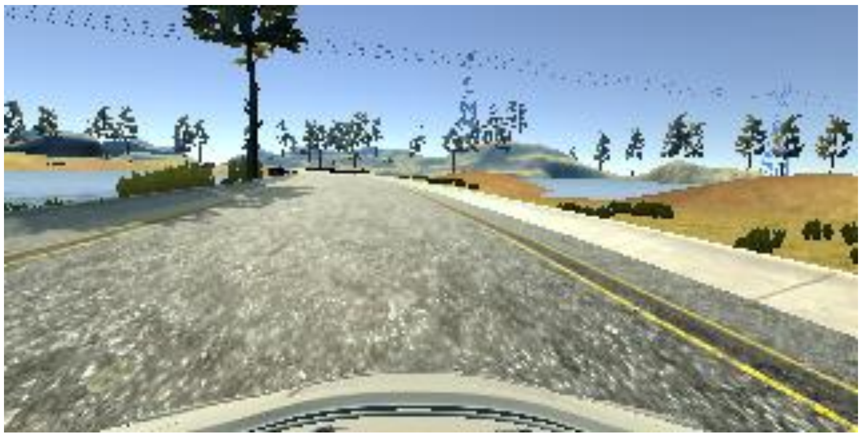
First model: 2189 images for centre view

Second model: 8167 images for all three camera perspectives or 2722 for each camera.

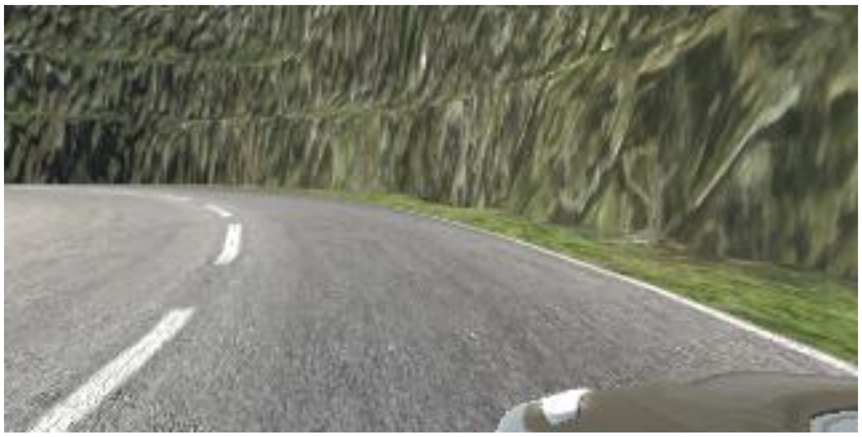
For augmenting the data set, I flipped vertically randomly 30 % of the images for each epoch. Also I inverted the steering angle for those images. The split between train and validation set is 80 % to 20 %.

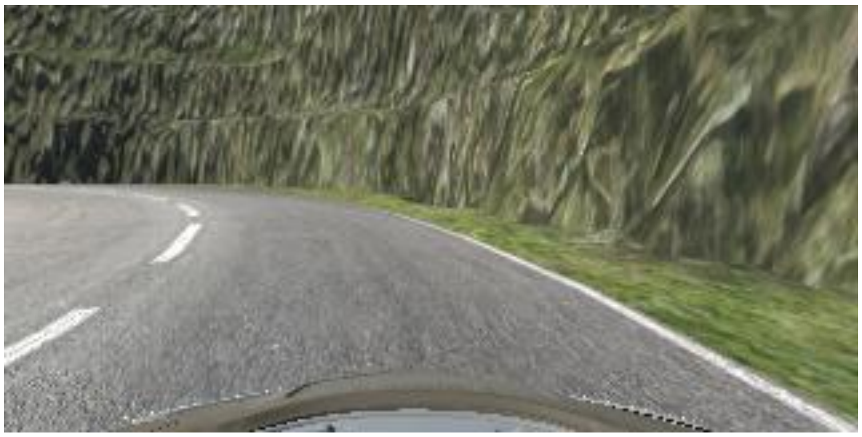
Here are some example image for the simple and the hard model. For the simple model, I show two images from the central camera, where I drove a curly line.

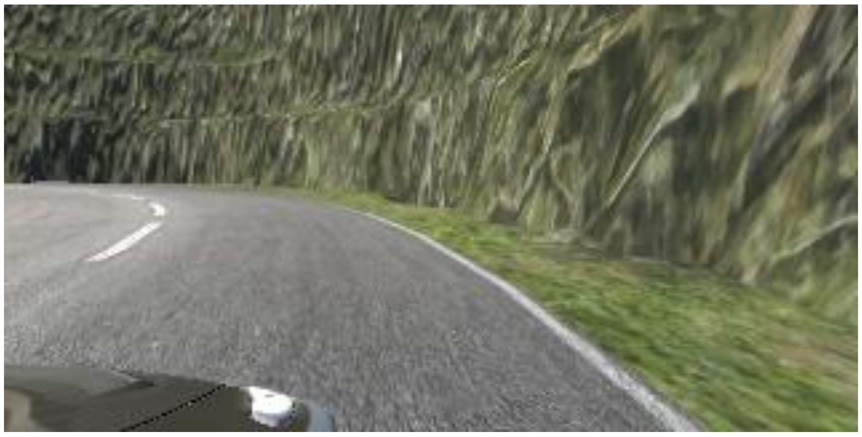




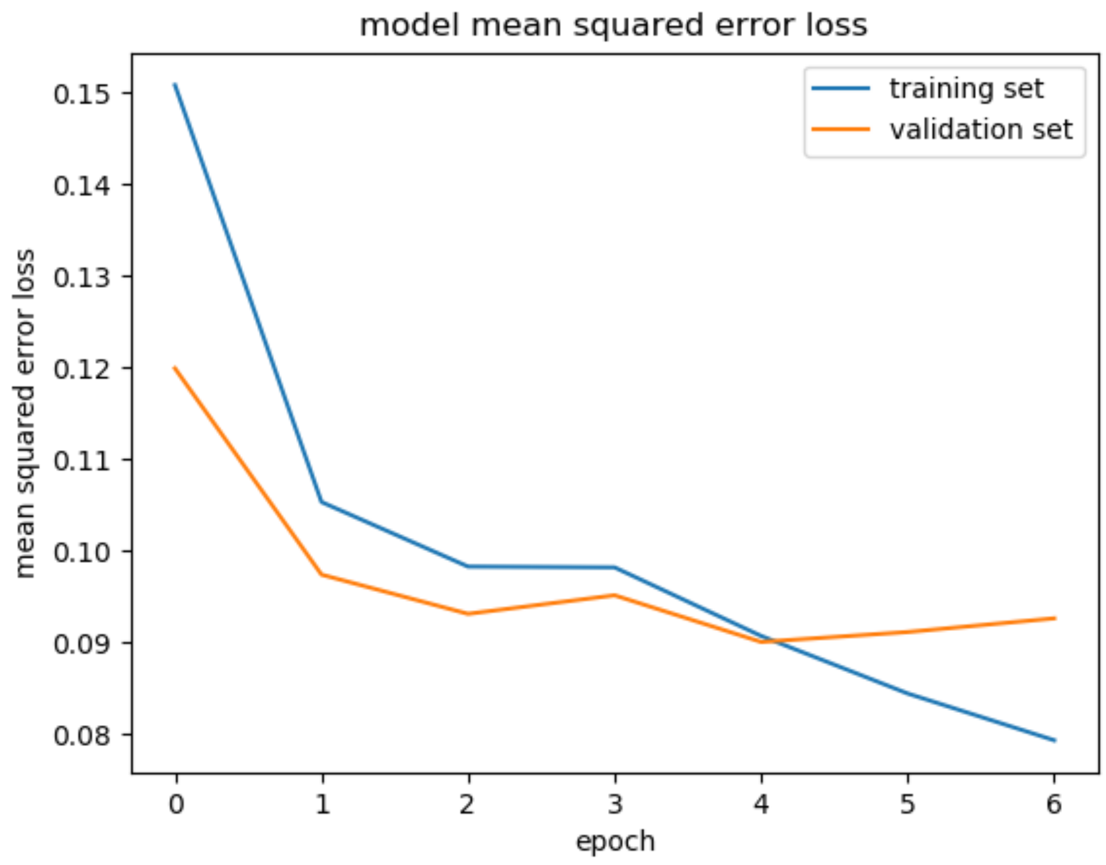
For the hard model, I show the different perspective of one image by front, left and right camera. For the model I kept the whole time on the right side of the road:







The validation set helped determine if the model was over- or underfitting. To prevent overfitting I used early stopping with saving the mode with the lowest validation lossl. The number of trained epochs for the first model was 2 and for the second model was 5. Since it doesn't make sense to plot the training output for the first model, I show here the visualisation for the second model. Also the influence of the early stopping with checkpoint saving is shown. The training loss keeps decreasing, but the validation set start increasing at epoch 5 (4 in the graph). So for epoch 6 and 7 there is already a slight overfitting. Here is the output of the model and the visualisation of the loss:



Second model training output

```
Epoch 1/20
152/153 [============================>.] - ETA: 7s - loss: 0.1511 -
mean_squared_error: 0.1511 Epoch 00001: val_loss improved from inf to 0.11988, saving
model to hard_test.h5
153/153 [==============================] - 1339s 9s/step - loss: 0.1508 -
mean_squared_error: 0.1508 - val_loss: 0.1199 - val_mean_squared_error: 0.1199
Epoch 2/20
152/153 [============================>.] - ETA: 7s - loss: 0.1046 -
mean_squared_error: 0.1046 Epoch 00002: val_loss improved from 0.11988 to 0.09739, saving
model to hard_test.h5
153/153 [==============================] - 1258s 8s/step - loss: 0.1044 -
mean_squared_error: 0.1044 - val_loss: 0.0974 - val_mean_squared_error: 0.0974
Epoch 3/20
152/153 [============================>.] - ETA: 7s - loss: 0.0973 -
mean_squared_error: 0.0973 Epoch 00003: val_loss improved from 0.09739 to 0.09311, saving
model to hard_test.h5
153/153 [==============================] - 1260s 8s/step - loss: 0.0975 -
mean_squared_error: 0.0975 - val_loss: 0.0931 - val_mean_squared_error: 0.0931
Epoch 4/20
152/153 [============================>.] - ETA: 7s - loss: 0.0996 -
mean_squared_error: 0.0996 Epoch 00004: val_loss did not improve
153/153 [==============================] - 1260s 8s/step - loss: 0.0997 -
mean_squared_error: 0.0997 - val_loss: 0.0951 - val_mean_squared_error: 0.0951
Epoch 5/20
152/153 [============================>.] - ETA: 7s - loss: 0.0910 -
mean_squared_error: 0.0910 Epoch 00005: val_loss improved from 0.09311 to 0.09005, saving
model to hard_test.h5
153/153 [==============================] - 1257s 8s/step - loss: 0.0910 -
mean_squared_error: 0.0910 - val_loss: 0.0900 - val_mean_squared_error: 0.0900
Epoch 6/20
152/153 [============================>.] - ETA: 7s - loss: 0.0844 -
mean_squared_error: 0.0844 Epoch 00006: val_loss did not improve
153/153 [==============================] - 1256s 8s/step - loss: 0.0844 -
mean_squared_error: 0.0844 - val_loss: 0.0911 - val_mean_squared_error: 0.0911
Epoch 7/20
152/153 [============================>.] - ETA: 7s - loss: 0.0796 -
mean_squared_error: 0.0796 Epoch 00007: val_loss did not improve
153/153 [==============================] - 1262s 8s/step - loss: 0.0795 -
mean_squared_error: 0.0795 - val_loss: 0.0926 - val_mean_squared_error: 0.0926
```