Behavioral Cloning

Said Zahrai

August 11, 2019

1 Introduction

The goal of this project is to build a model that is capable to simulate the action of a driver of a car in a simulator. The process is to drive the car and observe the environment form three cameras in front of car. For each image the actions taken by the "driver" is recorded and used for training.

As the first step, one has to drive the car in the simulator and collect data. The data will be provided as a directory with images and a comma separated file, called driving.log. The file contains the name of the three image files from center camera, left camera and right camera, together with steering, throttle, brake and speed. One set of driving data is provided initially.

The data will be used for training of the network. The problem is a regression problem, where the steering angle is to be modeled as a function of the image from the center camera. The success of the solution is measured by letting the car drive autonomously on the track. For that, a program is provided, '`drive.py`'. In this program, the speed is controlled to reach a given value, initially set to 9 (MPH). My goal was to find a solution that allows a reasonable autonomous drive at speeds 9, 20 and 30 MPH with the same model parameters.

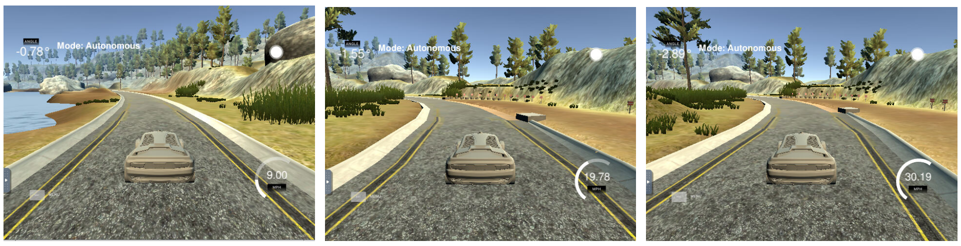

Figure 1: Simulations performed at three speeds: default in drive.py with speed set to 9 MPH (left), driving speed set to 20 MPH (middle) and speed set to 30 MPH (right).

2 Data

My input is a collection of the data provided by Udacity and test 3 and test 4 from [1].

Due to delays on the network, I had difficulties to drive the car on the cloud. Also, I did not succeed to install Unity so that I can drive the car locally on my computer. I have instead used the data that was shared on Internet.

2.1 Camera recorded images

Figure 2 below shows examples of the images recorded from the three cameras mounted in front of the car. The images are 160 pixels high and 320 pixels wide.

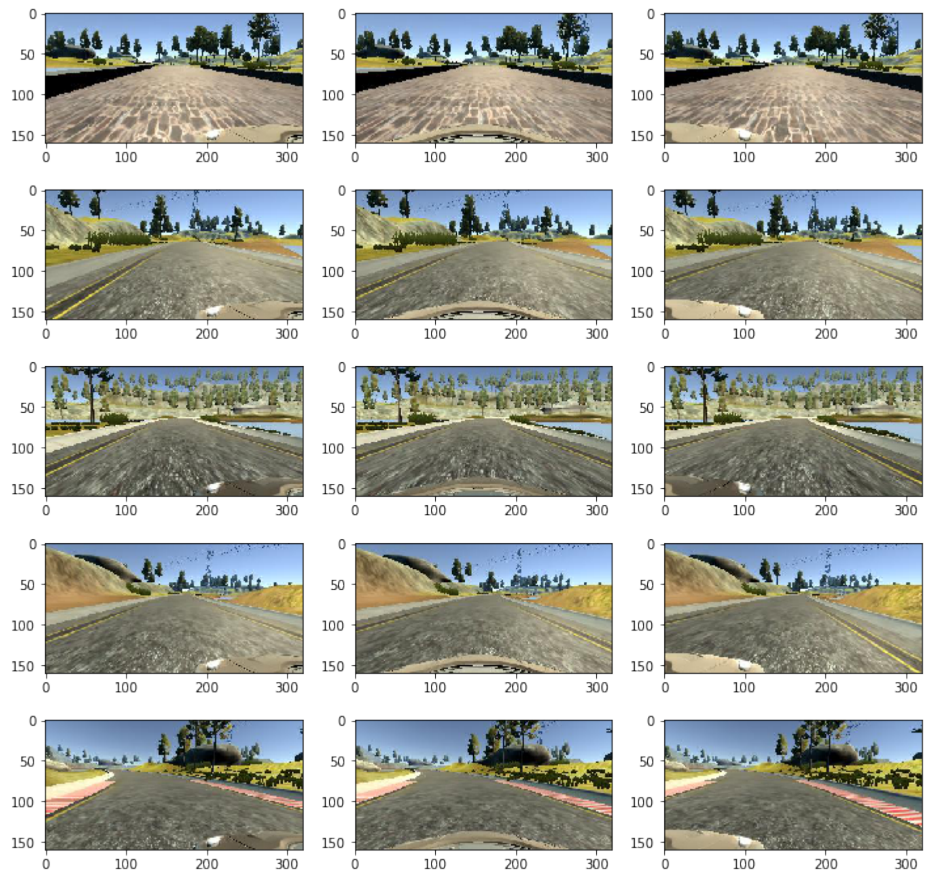

Figure 2: Camera recorded images. Left, seen from left camera, center from center camera and right from right camera.

As noted in the description of the project, there are large regions in the images that do not contain any relevant information for the purpose of the network, i.e. driving the car. Therefore, I have cut out the 60 pixels from the top and 50 pixels from the buttom. This makes the region of interest only 50 pixels high, which is less than one third of the original size.

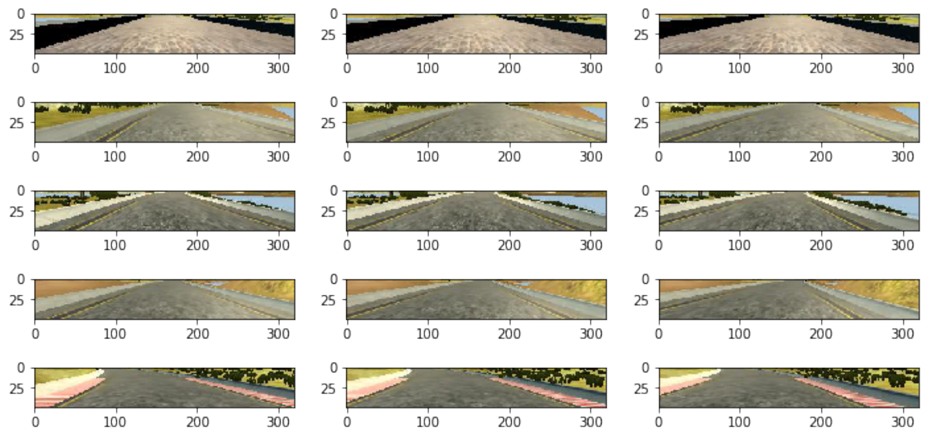

Figure 3: Camera recorded images reduced in the vertical direction. Left, seen from left camera, center from center camera and right from right camera.

2.2 Data augmentation

Data augmentation was done by three actions. First, following the description of the project, left images and right images were also used as input with a corrected steering value. The correction value was set to 0.25 after some experimentations.

```
x.append(left_image_address)
y.append(steering_value+0.25)
x.append(left_image_address)
y.append(steering_value-0.25)
```

The second suggested action for augmentation of data was to flip the images horizontally. This has been done but combined with more manipulations as described below. Before going through the procedure, there is a point worth some attention. Figure 4 shows the distribution of the data over the given steering command. As viewed here, the number of input data for zero angle is substantially higher than other values. This is a fact that will have some impact on the training procedure.

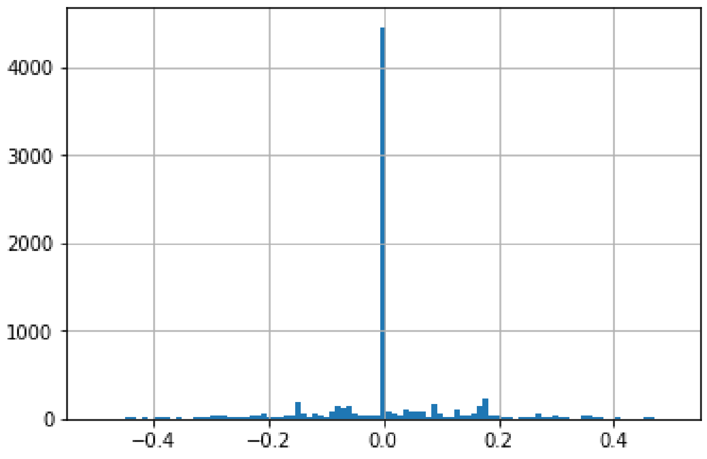

Figure 4: Distribution of steering values.

To reduce the overweight at zero, I have applied the following procedure to those input images where the absolute value of steering command has been less than 0.01. For 5% of the data, the image itself and its horizontally flipped image were added to the input list. For the remaining 95% the image was shifted to left and/or right and the steering command was altered by 0.02 per pixel. To make the effect more visible, for each image, two random numbers between 10 and 20 or between -20 and -10 is used to make the shift and the steering command was adjusted with 0.02 per pixel. This is implemented in Python as

```
img = plt.imread(image_file_address)
if (abs(batch_sample[1])<0.01):
    r = random.randint(0,400)
    if (r<20):
        images.append(img)
        angles.append(steering_value)
        images.append(np.fliplr(img))
        angles.append(-steering_value)
    elif (data_augmentation):
        r =random.randint(10,20)
        M = np.float32([[1,0, r],[0,1,0]])
        images.append(cv2.warpAffine(img,M,(cols,rows)))
        angles.append(batch_sample[1]+0.02*r)
        r =random.randint(-20,-10)
        M = np.float32([[1,0, r],[0,1,0]])
        images.append(cv2.warpAffine(img,M,(cols,rows)))
```

```
angles.append(steering_value+0.02*r)
```

Typical results with 20 pixels shifted to left or to right are shown in figure 5. To avoid dark edges in the figure after the shift, all images are corped such that 20 pixels from left and right are eliminated. Typical resulting images are shown in figure 6.

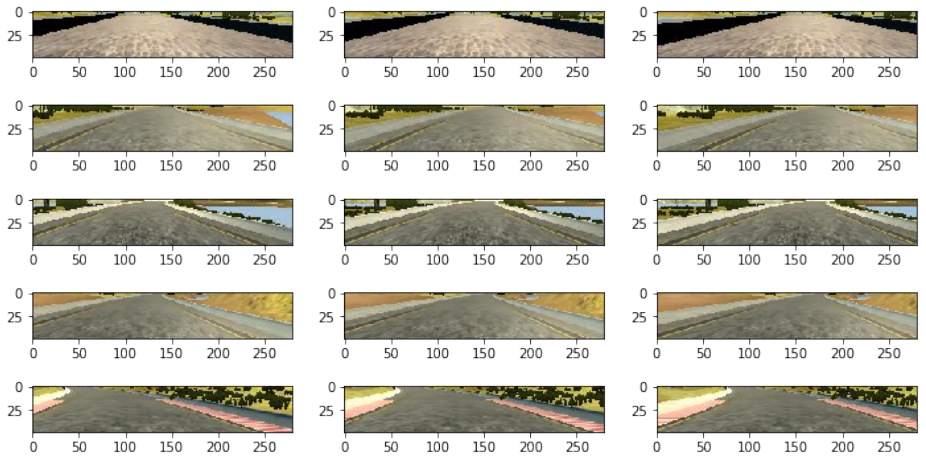

Figure 5: Shifted images: Left, shifted 20 pixels to left, center, original image and right, shifted 20 pixels to right.

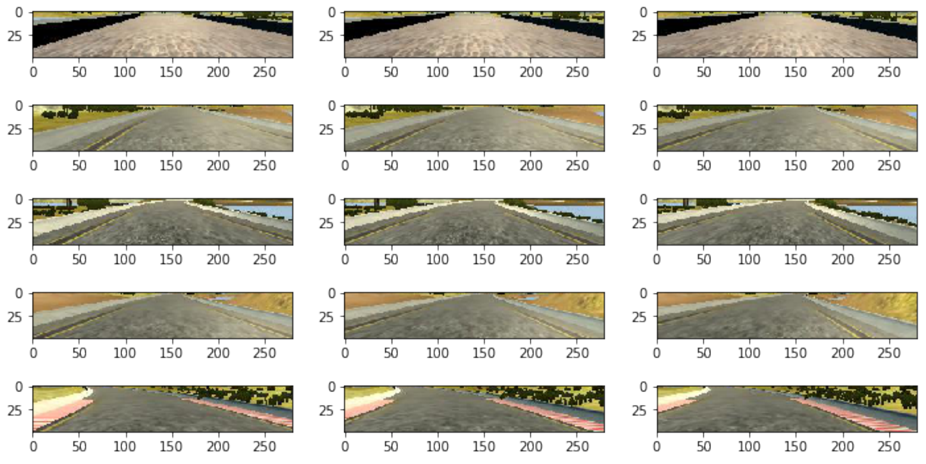

Figure 6: Corped images with focus on interesting region. Left, seen from left camera, center from center camera and right from right camera.

One can find quite some ideas for augmenting the input data by preprocessing the images. A good collection can be found in [?]. Although these methods most probably will have a positive impact on the results in a real-world situation, I believe the impact on the specific problem here will not be of interest. The network learns to handle situations that will not occur. The proposed method here, will be applicable by introducing a tendency to stay in the middle of the track.

3 The model

3.1 The network

Two different models, Lenet's network as presented in the course, and Nvidia's suggestion as in [3], were considered. Nvidia's network has 5 convolution layers with (24, 36, 48, 64, 64) kernels for feature extraction and 4 dense layers with (100, 50, 10, 1) neurons for regression, while Lenet has two convolution layers with (12, 32) kernels and 3 dense layers with (120, 84) neurons for regression. Consequently, Nvidia network is deeper and has more features.

To have a fair comparison, two reduced versions of Nvidia were also considered, where the number of features is reduced to 32 to be comparable to Lenet network. The reduced network has 5 convolution layers with (12, 18, 24, 32, 32) kernels for feature extraction and 4 dense layers with (50, 25, 10, 1) neurons for regression. This model is referred to as "half-Nvidia".

3.2 Activation

For Lenet, I used RELU to be consistent with previous projects. For Nvidia network, I tried both RELU and ELU. I could not see any major differences except that ELU seemed to be faster. Therefore, I decided to choose ELU.

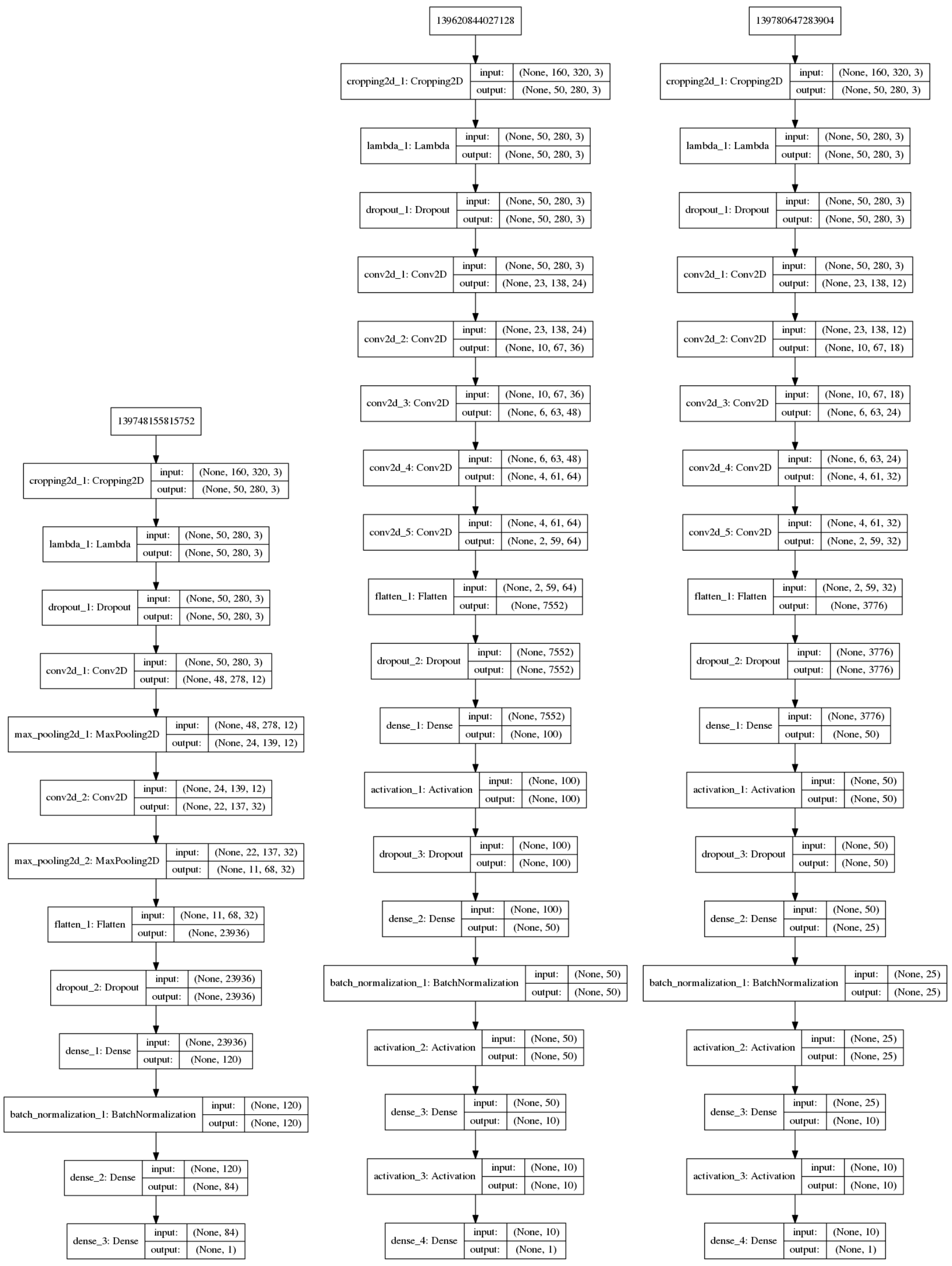
139748155815752
cropping2d_1: Cropping2D input: (None, 160, 320, 3) output: (None, 50, 280, 3)
lambda_1: Lambda input: (None, 50, 280, 3) output: (None, 50, 280, 3)
dropout_1: Dropout input: (None, 50, 280, 3) output: (None, 50, 280, 3)
conv2d_1: Conv2D input: (None, 50, 280, 3) output: (None, 48, 278, 12)
max_pooling2d_1: MaxPooling2D input: (None, 48, 278, 12) output: (None, 24, 139, 12)
conv2d_2: Conv2D input: (None, 24, 139, 12) output: (None, 22, 137, 32)
max_pooling2d_2: MaxPooling2D input: (None, 22, 137, 32) output: (None, 11, 68, 32)
flatten_1: Flatten input: (None, 11, 68, 32) output: (None, 23936)
dropout_2: Dropout input: (None, 23936) output: (None, 23936)
dense_1: Dense input: (None, 23936) output: (None, 120)
batch_normalization_1: BatchNormalization input: (None, 120) output: (None, 120)
dense_2: Dense input: (None, 120) output: (None, 84)
dense_3: Dense input: (None, 84) output: (None, 1)
139620844027128
cropping2d_1: Cropping2D input: (None, 160, 320, 3) output: (None, 50, 280, 3)
lambda_1: Lambda input: (None, 50, 280, 3) output: (None, 50, 280, 3)
dropout_1: Dropout input: (None, 50, 280, 3) output: (None, 50, 280, 3)
conv2d_1: Conv2D input: (None, 50, 280, 3) output: (None, 23, 138, 24)
conv2d_2: Conv2D input: (None, 23, 138, 24) output: (None, 10, 67, 36)
conv2d_3: Conv2D input: (None, 10, 67, 36) output: (None, 6, 63, 48)
conv2d_4: Conv2D input: (None, 6, 63, 48) output: (None, 4, 61, 64)
conv2d_5: Conv2D input: (None, 4, 61, 64) output: (None, 2, 59, 64)
flatten_1: Flatten input: (None, 2, 59, 64) output: (None, 7552)
dropout_2: Dropout input: (None, 7552) output: (None, 7552)
dense_1: Dense input: (None, 7552) output: (None, 100)
activation_1: Activation input: (None, 100) output: (None, 100)
dropout_3: Dropout input: (None, 100) output: (None, 100)
dense_2: Dense input: (None, 100) output: (None, 50)
batch_normalization_1: BatchNormalization input: (None, 50) output: (None, 50)
activation_2: Activation input: (None, 50) output: (None, 50)
dense_3: Dense input: (None, 50) output: (None, 10)
activation_3: Activation input: (None, 10) output: (None, 10)
dense_4: Dense input: (None, 10) output: (None, 1)
139780647283904
cropping2d_1: Cropping2D input: (None, 160, 320, 3) output: (None, 50, 280, 3)
lambda_1: Lambda input: (None, 50, 280, 3) output: (None, 50, 280, 3)
dropout_1: Dropout input: (None, 50, 280, 3) output: (None, 50, 280, 3)
conv2d_1: Conv2D input: (None, 50, 280, 3) output: (None, 23, 138, 12)
conv2d_2: Conv2D input: (None, 23, 138, 12) output: (None, 10, 67, 18)
conv2d_3: Conv2D input: (None, 10, 67, 18) output: (None, 6, 63, 24)
conv2d_4: Conv2D input: (None, 6, 63, 24) output: (None, 4, 61, 32)
conv2d_5: Conv2D input: (None, 4, 61, 32) output: (None, 2, 59, 32)
flatten_1: Flatten input: (None, 2, 59, 32) output: (None, 3776)
dropout_2: Dropout input: (None, 3776) output: (None, 3776)
dense_1: Dense input: (None, 3776) output: (None, 50)
activation_1: Activation input: (None, 50) output: (None, 50)
dropout_3: Dropout input: (None, 50) output: (None, 50)
dense_2: Dense input: (None, 50) output: (None, 25)
batch_normalization_1: BatchNormalization input: (None, 25) output: (None, 25)
activation_2: Activation input: (None, 25) output: (None, 25)
dense_3: Dense input: (None, 25) output: (None, 10)
activation_3: Activation input: (None, 10) output: (None, 10)
dense_4: Dense input: (None, 10) output: (None, 1)

Figure 7: Model structure: On left is Lenet's network, in the middle, Nvidia's network and on right, half-Nvidia.

3.3 Regularization

Earlier experiments showed that dropout was an efficient method for regularization in classification problems. According to the information that I could find, dropout could be applied to all layers, with recommended probability of 20% at input and 50% in hidden layers. In the regression problems, it appeared to me not suitable to have dropout at or near output layers. Equally, I preferred not to have dropout between the convolution layers as these layers are to extract the features.

To have a smooth output, I decided to use Batch regularization for the next last layer of the dense regression part and no regularization for the absolute last layer. I did try to use batch regularization throughout the network, but I found dropout as more efficient. Batch regularization implementation is using internal variables that are counted as parameters, but these are function of other parameters and therefore they are not trainable.

Input to convolution	Dropout 20%
Hidden convolution layers	None
After flattening	Dropout, 20% for Nvidia and 50% for Lenet
Hidden dense layers	Dropout, 50% (note: Lenet does not have any)
Next last hidden dense	Batch normalization
Before output	None

Table 1: Regularization strategy for regression problem.

3.4 Keras generator

Following the recommendation, a generator was used to avoid high consumption of memory. In this generator, for each batch of 32 the original images from three cameras were loaded and processed for augmentation of data according to the above mentioned procedure.

3.5 Data normalization and cropping

The input image was first cropped by excluding 60 pixels from top, 50 from bottom and 20 pixels from each side. Thereafter a lambda layers was used to normalize the input data to range of [-1,1].

3.6 Optimization

Adam optimizer was used with mean square error as loss function with early stopping when changes int he validation loss was less than 0.0001 and patience of two to avoid stopping because of accidental jumps. The mode was set to 'min' so that the output will be the best solution.

3.7 Training and validation

The list of input data was first processed so that each data point presented the file name of one image and one expected output steering command. At this stage, already the left and right images are used with an estimated steering command corrected with -0.25 for right camera image and +0.25 for left camera image. The list is worked on in batches of 32 posts and the information in each data post is used in the generator function, where the images are read, further augmentation is made and the data prepared for optimization.

For all cases, the 20% of the data was used for validation. The images from left and right cameras were only used for training and not for the validation step. Likewise, the generator did not perform any data augmentation for validation data. Consequently, the validation data will be pure collected data.

The above steps lead to an optimization problem with the size as presented in 2.

	Lenet	Nvidia	half-Nvidia
Total params	2,886,993	892,419	223,842
Trainable params	2,886,753	892,319	223,792
Non-trainable params	240	100	50

Table 2: Model parameters.

4 Results

To see the impact of the data size the networks were first trained with the data provided by Udacity and thereafter with additional two data sets from [1]. Figures 8 and 9 show the convergence of the optimization process and tables 3 and 4 present the summary of results. As can be found in these tables, use of more data has definitely a positive impact on the results. Further, the Nvidia network does a better job than Lenet, even when the width of the network is reduced to half.

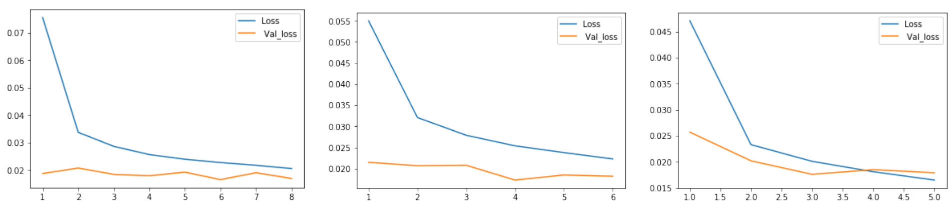

Figure 8: Convergence history for group the networked trained with the data set provided by Udacity. Left, Nvidia's network, center, half-Nvidia network and right Lenet's network.

Speed	Nvidia	half-Nvidia	Lenet
9 MPH	OK	OK	Crashed
20 MPH	wavy ride	wavy ride	Crashed
30 MPH	Crashed	Crashed	Crashed

Table 3: Summary of results trained with one data set.

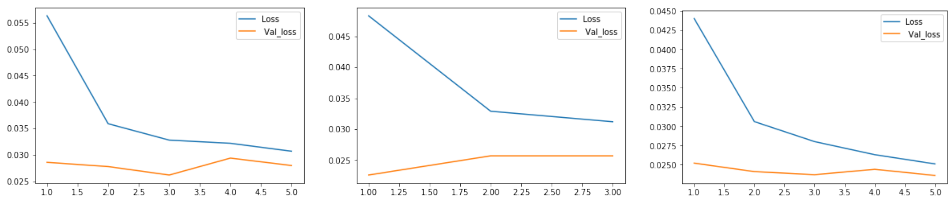

Figure 9: Convergence history for group the networked trained with 3 data sets, one from Udacity and two from [?]. Left, Nvidia's network, center, half-Nvidia network and right Lenet's network.

Speed	Nvidia	Lenet	Reduced Nvidia
9 MPH	OK	OK	OK
20 MPH	OK	OK	Crashed
30 MPH	wavy ride	wavy ride	Crashed

Table 4: Summary of results trained with one data set.

5 Discusison and conclusion

Due to the difficulties I had for generating data, I focused on data from the first track. This is certainly not acceptable for a real-world problem and data from alternative tracks need to be collected and used. However, I think for a given target, one should be able train the network for that specific problem with combined proper data augmentation and dimensioning of the network.

My observation from the cases presented here together with other tests I made indicate that I would be able to obtain better result by tuning the parameter I used for data augmentation. I did not find it particularly useful as it will be an artificial improvement.

The method only clones the action of the drivers. A collection of data from better drivers would certainly give a better result. Also, in a real problem, one could edit the data and eliminate sections with bad driving skills. Obviously, adding some rules, such as trying to follow lane lines, will have a strong positive impact.

The results of this project indicate that a deep network can become more competent. This is an important observation as a deeper network appears be more efficient to train than a wide one.

6 Acknowledgment

I would like to thank Mr. Asshish Vivekanand for kindly sharing his data.

References

[1] Vivekanand, Asshish, Behavioral-Cloning-P3

[2] Yadav, Vivek, An augmentation based deep neural network approach to learn human driving behavior

[3] Nvidia team, End to End Learning for Self-Driving Cars