Driving simulator using Deep Learning and Behavioral Cloning

The Self-Driving Car engineer program designed by Udacity is currently the only machine learning program which is focused entirely on autonomous driving. The program offers worldclass traning staff and prominent partners like Nvidia or Mercedes Benz.

Besides interesting lessons and exercises the program expects students to prove their deep learning skills in real world projects.

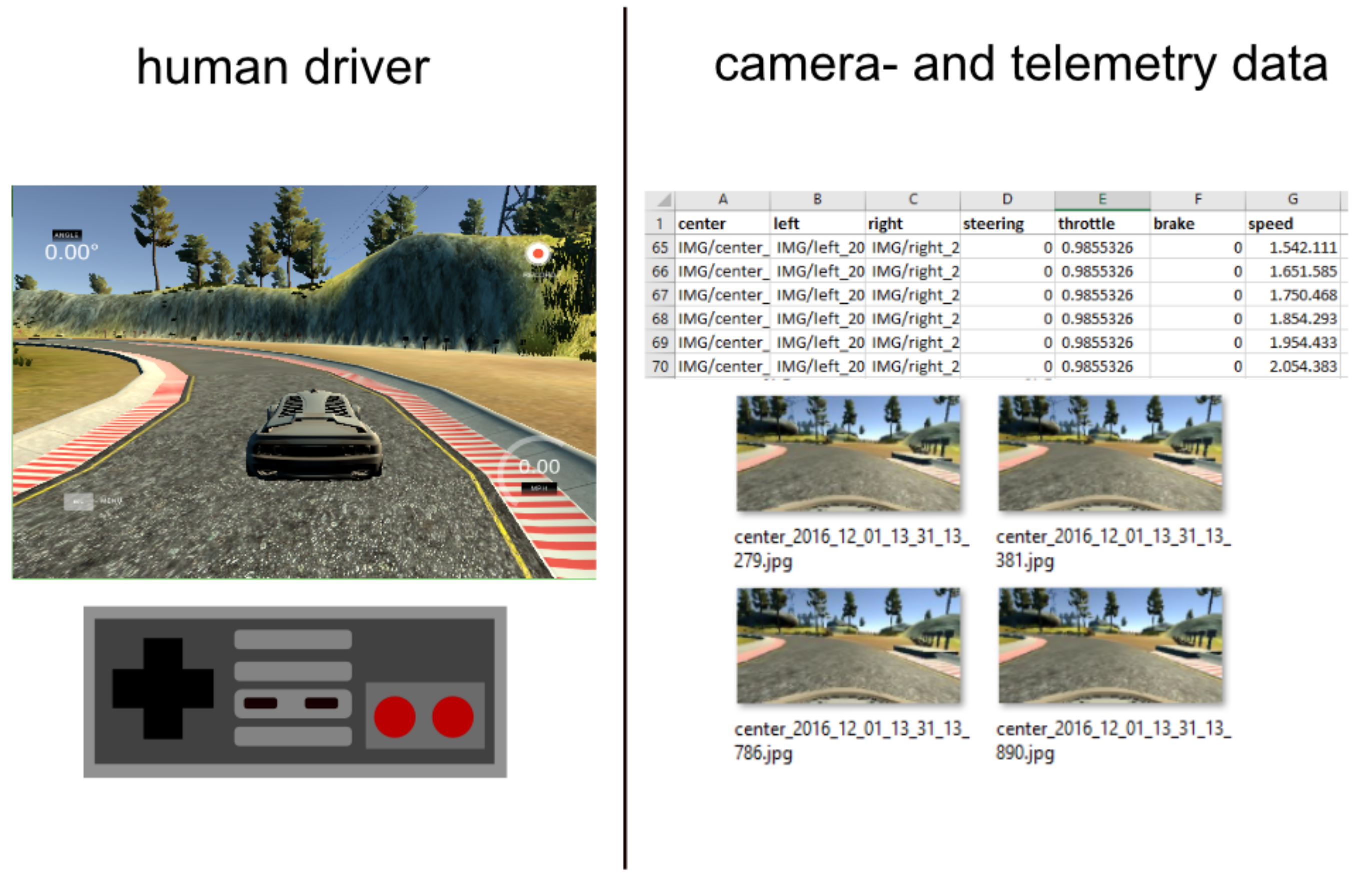
The third project of this program is called Behavioral Cloning and we will build and train a deep neural network to drive a car like us!

The goals / steps of this project

- 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

In this project one has to create a machine learning model which is able to mimic a human driver. To be more specific: the model has to be able to drive a car safely through an unseen track in a simulator. The simulator provided by Udacity has two modes:

In **manual mode** a human driver can control the car with a game controller or keyboard. The simulator will record multiple images every seconds to disk of three on-board cameras. It will also record telemetry data like steering, break, throttle and speed to a CSV file.

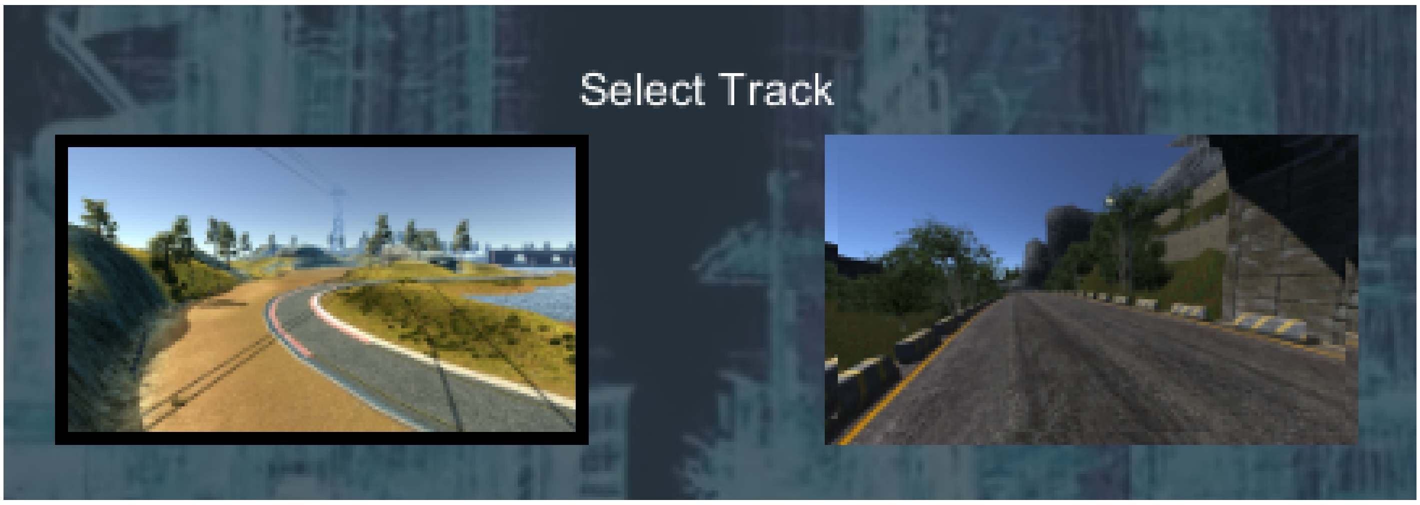

human driver
camera- and telemetry data
0.00°
0.00
center_2016_12_01_13_31_13_279.jpg
center_2016_12_01_13_31_13_381.jpg
center_2016_12_01_13_31_13_786.jpg
center_2016_12_01_13_31_13_890.jpg

	A	B	C	D	E	F	G
1	center	left	right	steering	throttle	brake	speed
65	IMG/center_	IMG/left_20	IMG/right_2	0	0.9855326	0	1.542.111
66	IMG/center_	IMG/left_20	IMG/right_2	0	0.9855326	0	1.651.585
67	IMG/center_	IMG/left_20	IMG/right_2	0	0.9855326	0	1.750.468
68	IMG/center_	IMG/left_20	IMG/right_2	0	0.9855326	0	1.854.293
69	IMG/center_	IMG/left_20	IMG/right_2	0	0.9855326	0	1.954.433
70	IMG/center_	IMG/left_20	IMG/right_2	0	0.9855326	0	2.054.383

The driving simulator records the driving behavior of a human driver.

In **autonomous mode** the simulator streams its camera pictures to a python script using SocketIO and can receive commands for throttle and steering. Our goal is to design a neural network with the deep learning framework Keras to remote control the car safely through the track.

The simulator also offers two different tracks. To train our model we will use data of the first track. To validate it we will use the second track which is more demanding.



Loading the dataset

We are using only the samples for the Track 1 provided by Udacity. It can be downloaded from here. The Track 2 is used to test the trained model.

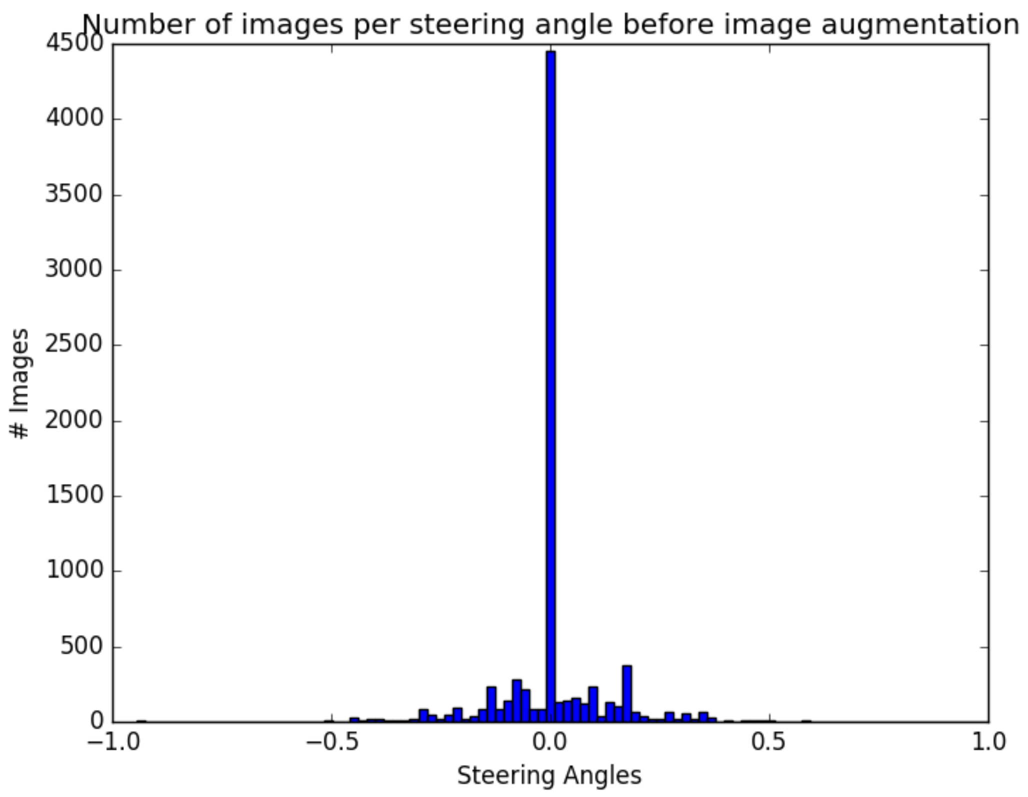
The dataset is stored in the 'driving_log.csv' file. This csv file has paths to the three left, center and right camera image files on the disk, and their corresponding steering angles.

After loading and exploring the dataset using the histogram of steering angles we can see that the dataset is very un-balanced with most of the entries for driving straight.

```
print("\nLoading the dataset from file ...")

def load_dataset(file_path):
    dataset = []
    with open(file_path) as csvfile:
        reader = csv.reader(csvfile)
        for line in reader:
            try:
                dataset.append({'center':line[0], 'left':line[1], 'right':line[2], 'steering':float(line[3]),
                            'throttle':float(line[4]), 'brake':float(line[5]), 'speed':float(line[6])})
            except:
                continue # some images throw error during loading
    return dataset

dataset = load_dataset(DATASET_PATH)
print("Loaded {} samples from file {}".format(len(dataset),DATASET_PATH))
```

```
import matplotlib.pyplot as plt

print("\nExploring the dataset ...")

# It plots the histogram of an arrray of angles: [0.0,0.1, ..., -0.1]
def plot_steering_histogram(steerings, title, num_bins=100):
    plt.hist(steerings, num_bins)
    plt.title(title)
    plt.xlabel('Steering Angles')
    plt.ylabel('# Images')
    plt.show()

# It plots the histogram of an arrray of associative arrays of angles: [{'steering':0.1}, {'steering':0.2}, ...,
{'steering':-0.1}]
def plot_dataset_histogram(dataset, title, num_bins=100):
    steerings = []
    for item in dataset:
        steerings.append( float(item['steering']) )
    plot_steering_histogram(steerings, title, num_bins)

# Plot the histogram of steering angles before the image augmentation
plot_dataset_histogram(dataset, 'Number of images per steering angle before image augmentation', num_bins=100)
print("Exploring the dataset complete.")
```

The dataset is biased towards small steering angles and certain directions as shown in the histogram below.



Partitioning the dataset

Before the dataset is partioned the samples are shuffled for a better generalization. The dataset is then split into training (80%), validation (19%) and testing (1%).

```
from random import shuffle
from sklearn.model_selection import train_test_split

print("\nPartitioning the dataset ...")

# Images of sequences must be found in all 3 datasets for a better generalization
shuffle(dataset)

X_train, X_validation = train_test_split(dataset, test_size=0.2)
X_validation, X_test = train_test_split(X_validation, test_size=0.05)

print("X_train has {} elements.".format(len(X_train)))
print("X_validation has {} elements.".format(len(X_validation)))
print("X_test has {} elements.".format(len(X_test)))
print("Partitioning the dataset complete.")
```

Dataset augmentation

Data preparation is required when working with neural network and deep learning models. We will use data augmentation to generate new images, especially for the ones with a lower value in the histogram of steering angles.

Most of the methods are wrappers of implementations in Keras. The data augmentation methods used in this project include:

- zoom
- rotation
- height and width shift
- horizontal flip
- channel shift
- crop and resize

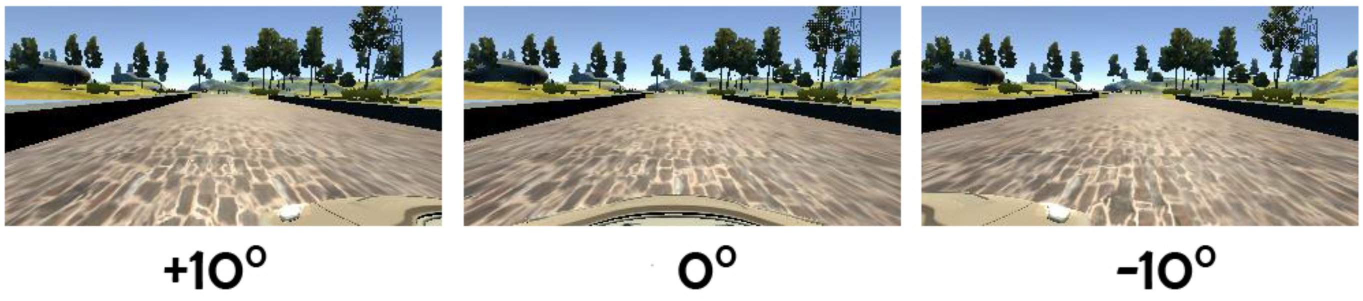
- brightness shift

All these transformations are randomly applied in memory on the original image using a generator, a Keras technique to avoid storing all these new images on the disk.

```
import cv2
import numpy as np
from keras.preprocessing.image import *

# Flip image horizontally, flipping the angle positive/negative
def horizontal_flip(image, steering_angle):
    flipped_image = cv2.flip(image,1)
    steering_angle = -steering_angle
    return flipped_image, steering_angle

# Shift width/height of the image by a small fraction of the total value, introducing an small angle change
def height_width_shift(image, steering_angle, width_shift_range=50.0, height_shift_range=5.0):
    # translation
    tx = width_shift_range * np.random.uniform() - width_shift_range / 2
    ty = height_shift_range * np.random.uniform() - height_shift_range / 2

    # new steering angle
    steering_angle += tx / width_shift_range * 2 * 0.2

    transform_matrix = np.float32([[1, 0, tx], [0, 1, ty]])
    rows, cols, channels = image.shape

    translated_image = cv2.warpAffine(image, transform_matrix, (cols, rows))
    return translated_image, steering_angle

# Increase the brightness by a certain value or randomly
def brightness_shift(image, bright_increase=None):
    image_hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)

    if bright_increase:
        image_hsv[:,:,2] += bright_increase
    else:
        bright_increase = int(30 * np.random.uniform(-0.3,1))
        image_hsv[:,:,2] = image[:,:,2] + bright_increase

    image = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB)
    return image

# Shift range for each channels
def channel_shift(image, intensity=30, channel_axis=2):
    image = random_channel_shift(image, intensity, channel_axis)
    return image

# Rotate the image randomly up to a range_degrees
def rotation(image, range_degrees=5.0):
    #image = random_rotation(image, range_degrees)
    degrees = np.random.uniform(-range_degrees, range_degrees)
    rows,cols = image.shape[:2]
    matrix = cv2.getRotationMatrix2D((cols/2,rows/2),degrees,1.0)
    image = cv2.warpAffine(image, matrix, (cols,rows), borderMode=cv2.BORDER_REPLICATE)
    return image

# Zoom the image randomly up to zoom_range, where 1.0 means no zoom and 1.2 a 20% zoom
def zoom(image, zoom_range=(1.0,1.2)):
    #image = random_zoom(image, zoom_range)
    # resize
    factor = np.random.uniform(zoom_range[0], zoom_range[1])
    height, width = image.shape[:2]
    new_height, new_width = int(height*factor), int(width*factor)
    image = cv2.resize(image, (new_width, new_height), interpolation=cv2.INTER_LINEAR)

    # crop margins to match the initial size
    start_row = int((new_height-height)/2)
    start_col = int((new_width-width)/2)
    image = image[start_row:start_row + height, start_col:start_col + width]

    return image
```

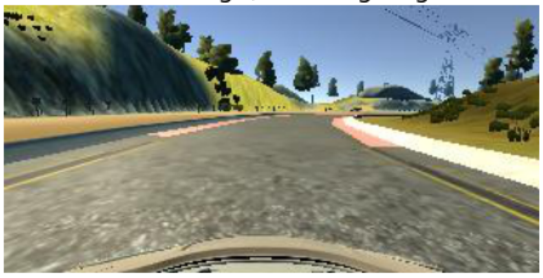
```
# Crop and resize the image
def crop_resize_image(image, cols=INPUT_IMAGE_COLS, rows=INPUT_IMAGE_ROWS, top_crop_perc=0.1,
bottom_crop_perc=0.2):
    height, width = image.shape[:2]

    # crop top and bottom
    top_rows = int(height*top_crop_perc)
    bottom_rows = int(height*bottom_crop_perc)
    image = image[top_rows:height-bottom_rows, 0:width]

    # resize to the final sizes even the aspect ratio is destroyed
    image = cv2.resize(image, (cols, rows), interpolation=cv2.INTER_LINEAR)
    return image

# Apply a sequence of random tranformations for a bettwe generalization and to prevent overfitting
def random_transform(image, steering_angle):

    # all further transformations are done on the smaller image to reduce the processing time
    image = crop_resize_image(image)

    # every second image is flipped horizontally
    if np.random.random() < 0.5:
        image, steering_angle = horizontal_flip(image, steering_angle)

    image, steering_angle = height_width_shift(image, steering_angle)
    image = zoom(image)
    image = rotation(image)
    image = brightness_shift(image)
    image = channel_shift(image)

    return img_to_array(image), steering_angle
```

A very important way to introduce images with new steering angles is to use the side cameras to simulate recovery paths. If you only use the center camera pictures your car will soon leave the track and crash. This is due to the fact that when recording only the ideal driving path, it will not know what to do when being slightly off. One way to solve this would be to record new data while driving from the side of the street back to the middle.

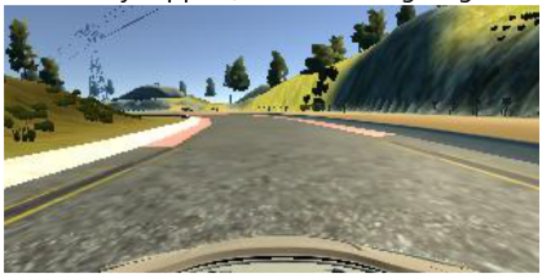


Bellow we can see the original image in the top-left and the result of each method independently.

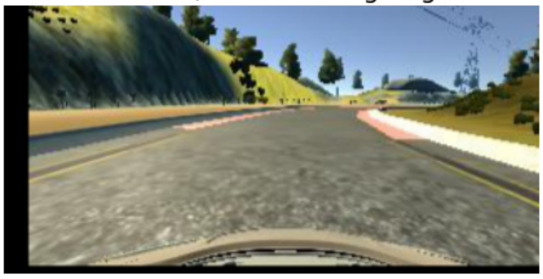
Initial test image, steering angle: 0.0



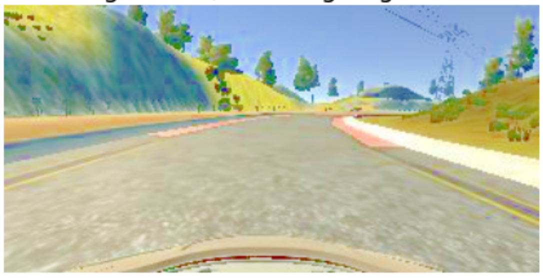
Horizontally flipped, new steering angle: -0.0



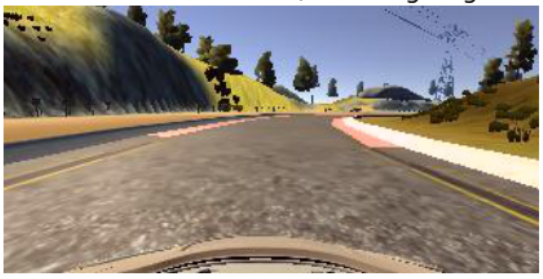
Random width shifted, new steering angle: 0.11858296



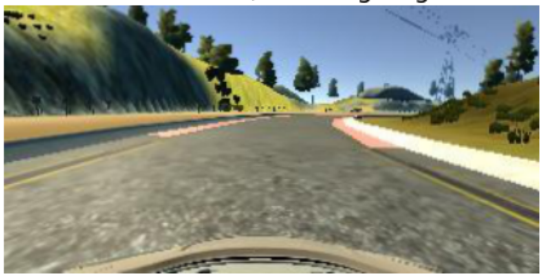
Brightened, Steering angle: 0.0



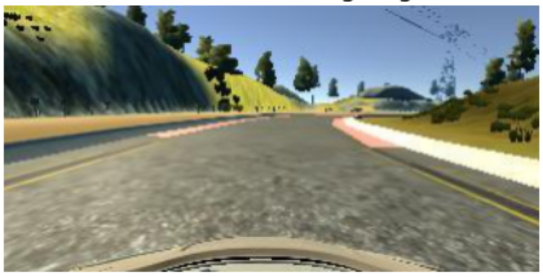
Random channel shifted, steering angle: 0.0



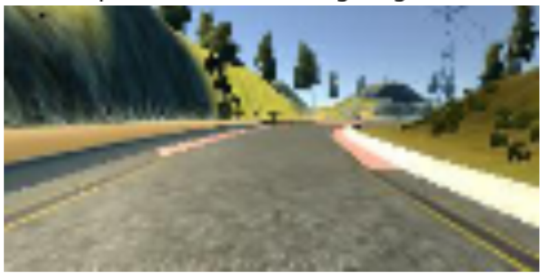
Random rotation, steering angle: 0.0



Random zoom, steering angle: 0.0



Crop & resize, steering angle: 0.0



And here we can see several images after randomly combining all these methods. This transformed images are sent as input to the neural network, and never the original images.

```
# Apply a sequence of random tranformations for a better generalization and to prevent overfitting
def random_transform(image, steering_angle):

    # all further transformations are done on the smaller image to reduce the processing time
    image = crop_resize_image(image)

    # every second image is flipped horizontally
    if np.random.random() < 0.5:
        image, steering_angle = horizontal_flip(image, steering_angle)

    image, steering_angle = height_width_shift(image, steering_angle)
    image = zoom(image)
    image = rotation(image)
    image = brightness_shift(image)
    image = channel_shift(image)
```

Using these data augmentation methods an infinite number of new images can be generated. In this project images with steering angles near 0.0 (driving straight) are less augmented than the ones at the margins. This histogram equalization is needed to give the model the chance to learn how to recover in the cases when the car is near the margins.

Keras generator for data augmentation

The memory is a limited resource and is not possible and efficient to load all images at once. Therefore we will utilize Keras' generator to sample images such that all angles have the similar probability no matter how they are represented in the dataset. This alleviates any problem we may ecounter due to model having a bias towards driving straight. To implement this, a hit dictionary for each angle range is maintained, and each hit bin is allowed a maximum percentage of the the total number of already augmented images.

Given a row from the .csv file, this method randomly reads the center, left or right image. It applies all image augmentation methods to it and returns the augmented image and transformed steering angle. This method is used in the Keras generator as shown below.

```
# It loads one of the 3 images (center, left or right) and applies augumentation on it
# This method is called by the Keras generator
def load_and_augment_image(image_data, side_camera_offset=0.2):

    # select a value between 0 and 2 to swith between center, left and right image
    index = np.random.randint(3)

    if (index==0):
        image_file = image_data['left'].strip()
        angle_offset = side_camera_offset
    elif (index==1):
        image_file = image_data['center'].strip()
        angle_offset = 0.
    elif (index==2):
        image_file = image_data['right'].strip()
        angle_offset = - side_camera_offset

    steering_angle = image_data['steering'] + angle_offset

    image = cv2.imread(image_file)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # apply a misture of several augumentation methods
    image, steering_angle = random_transform(image, steering_angle)

    return image, steering_angle

# Define some global variables used for augumenting the images
# A list of generated angles in all epochs alltogether for showing the displaying the final histogram
augmented_steering_angles = []
# The number of generated angels in the current epoch, reseted at the end of each epoch
epoch_steering_count = 0
# A dictionary with the hits for each angle, reseted at the end of each epoch
# for the angles is multiplied by 100 and converted to integer, with the range aproximately (-150, +150)
# for the angle -0.142323 the index will be -14
bin_range = int(AUGMENTATION_NUM_BINS / 4 * 3) # for AUGMENTATION_NUM_BINS = 200 the range is (-150, 150)
epoch_bin_hits = {k:0 for k in range(-bin_range, bin_range)}

@threadsafe_generator
```

```
def generate_batch_data(dataset, batch_size = 32):
    global augmented_steering_angles
    global epoch_steering_count
    global epoch_bin_hits
    batch_images = np.zeros((batch_size, INPUT_IMAGE_ROWS, INPUT_IMAGE_COLS, INPUT_IMAGE_CHANNELS))
    batch_steering_angles = np.zeros(batch_size)

    while 1:
        for batch_index in range(batch_size):

            # select a random image from the dataset
            image_index = np.random.randint(len(dataset))
            image_data = dataset[image_index]

            while 1:
                try:
                    image, steering_angle = load_and_augment_image(image_data)
                except:
                    continue # some images throw error during loading/augmentation

                # images with smaller angles have a lower probability of getting represented in the dataset,
                # and the model tends to have a bias towards driving straight
                # for equalizing the histogram a dictionary of hits for each bin is used: epoch_bin_hits

                # get current bin index, for AUGMENTATION_NUM_BINS = 200 the angle is multiplied by 100
                bin_idx = int (steering_angle * AUGMENTATION_NUM_BINS / 2)

                # don't allow one bin to have more than AUGMENTATION_BIN_MAX_PERC percent of the total augmented
angles in the current epoch,
                # except for the case when not enough (less than 500) angles are already augmented
                if( epoch_bin_hits[bin_idx] < epoch_steering_count/AUGMENTATION_NUM_BINS*AUGMENTATION_BIN_MAX_PERC
                    or epoch_steering_count<500 ):

                    batch_images[batch_index] = image
                    batch_steering_angles[batch_index] = steering_angle
                    augmented_steering_angles.append(steering_angle)

                    epoch_bin_hits[bin_idx] = epoch_bin_hits[bin_idx] + 1
                    epoch_steering_count = epoch_steering_count + 1
                    break

        yield batch_images, batch_steering_angles
```

Model architecture

We have to design a neural network which takes camera pictures as inputs and outputs steering angles. You can come up with your own neural network architecture. If you want to save time you can use the following two architectures as a starting point:

- comma.ai - Learning a Driving Simulator
- End to End Learning for Self-Driving Cars by nvidia

I used the comma.ai network as reference which uses a simple yet elegant architecture to predict steering angles. Comma.ai's approach to Artificial Intelligence for self-driving cars is based on an agent that learns to clone driver behaviors and plans maneuvers by simulating future events in the road.

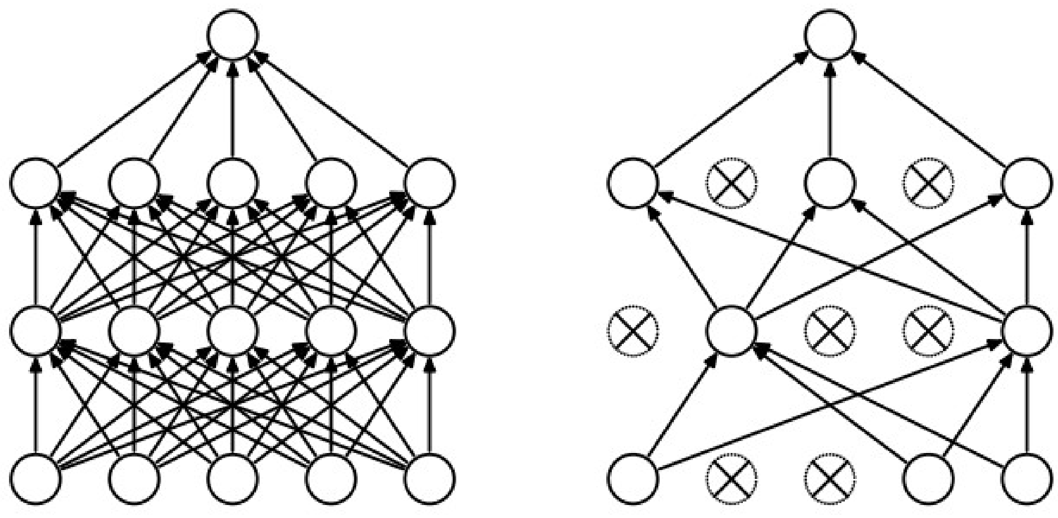
```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to                     
====================================================================================================
lambda_1 (Lambda)                (None, 60, 120, 3)    0           lambda_input_1[0][0]             
____________________________________________________________________________________________________
Conv1 (Convolution2D)            (None, 15, 30, 16)    3088        lambda_1[0][0]                   
____________________________________________________________________________________________________
Conv2 (Convolution2D)            (None, 8, 15, 36)     14436       Conv1[0][0]                      
____________________________________________________________________________________________________
Conv3 (Convolution2D)            (None, 4, 8, 64)      57664       Conv2[0][0]                      
____________________________________________________________________________________________________
flatten_1 (Flatten)              (None, 2048)          0           Conv3[0][0]                      
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 2048)          0           flatten_1[0][0]                  
____________________________________________________________________________________________________
elu_1 (ELU)                      (None, 2048)          0           dropout_1[0][0]                  
____________________________________________________________________________________________________
FC1 (Dense)                      (None, 512)           1049088     elu_1[0][0]                      
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 512)           0           FC1[0][0]                        
____________________________________________________________________________________________________
elu_2 (ELU)                      (None, 512)           0           dropout_2[0][0]                  
____________________________________________________________________________________________________
output (Dense)                   (None, 1)             513         elu_2[0][0]                      
====================================================================================================
Total params: 1124789
____________________________________________________________________________________________________
```

The network has a normalization layer followed by 3 convolutional layers with 8x8 and 5x5 kernel size and strides of [2,2]. After the convolutional layers the output gets flatten and then processed by one fully connected layer. The whole network will have roughly 1 milion parameters and will offer great training performance on modest hardware.

The whole network can be implemented in keras with just a few lines of code:

```
from keras.models import Sequential, Model
from keras.layers.core import Lambda, Dense, Activation, Flatten, Dropout
from keras.layers.convolutional import Cropping2D, Convolution2D
from keras.layers.advanced_activations import ELU
from keras.layers.noise import GaussianNoise
from keras.optimizers import Adam

print("\nBuilding and compiling the model ...")

model = Sequential()
# Preprocess incoming data, centered around zero with small standard deviation
model.add(Lambda(lambda x: (x / 127.5) - 1.0, input_shape=(INPUT_IMAGE_ROWS, INPUT_IMAGE_COLS,
INPUT_IMAGE_CHANNELS)))
# Block - conv
model.add(Convolution2D(16, 8, 8, border_mode='same', subsample=[4,4], activation='elu', name='Conv1'))
# Block - conv
model.add(Convolution2D(36, 5, 5, border_mode='same', subsample=[2,2], activation='elu', name='Conv2'))
# Block - conv
model.add(Convolution2D(64, 5, 5, border_mode='same', subsample=[2,2], activation='elu', name='Conv3'))
# Block - flatten
model.add(Flatten())
model.add(Dropout(0.2))
model.add(ELU())
# Block - fully connected
model.add(Dense(512, activation='elu', name='FC1'))
model.add(Dropout(0.5))
model.add(ELU())
# Block - output
model.add(Dense(1, name='output'))
model.summary()
```

To make the architecture more robust and to prevent overfitting dropout layers are added to the network. Dropout disables neurons in your network by a given probability and prevents co-adaption of features.



(a) Standard Neural Net (b) After applying dropout.

A 'mean_squared_error' loss function is used. As for the optimization algorithm Adam is used and configured with a learning rate of 0.0001. This learning rate was selected after many trial and error. With larger learning rates (bigger or equal to 0.001) the car was making abrupt steering angle changes.

Training strategy

The dataset provided by the project contains a csv file of metadatas and a list of images. The csv file was first parsed and image paths and steering angles were extracted, shuffled and then splitted into training, validation and test data.

An image generator that takes the image metadata as input and generates batches of training data was implemented. By using the right and left camera images with a 0.20 degrees shift the dataset size became 3 times larger. The input images were augmented using different methods such as zoom, scale, brightness shift, horizontal flip, height and width shift and color channel shift. Images were as well cropped out to keep only the region of interest under the horizon and above the car board. The images were then resized to 120 × 60 pixels before being passed into the model.

The ADAM optimizer with a learning rate of 0.0001 was used. The data is trained over 10 epochs of 24000 samples each.

Model fitting

One of the key strategies of this solution is to group the steering angles into a fixed number of hit bins and to augument only a certain number of images of each bin. This way each angle range will be equaly learned by the model. This will prevent the model of having a bias towards driving straight. For better results the bin hits are reinitialized after each epoch. To implement this a LifecycleCallback used as shown bellow.

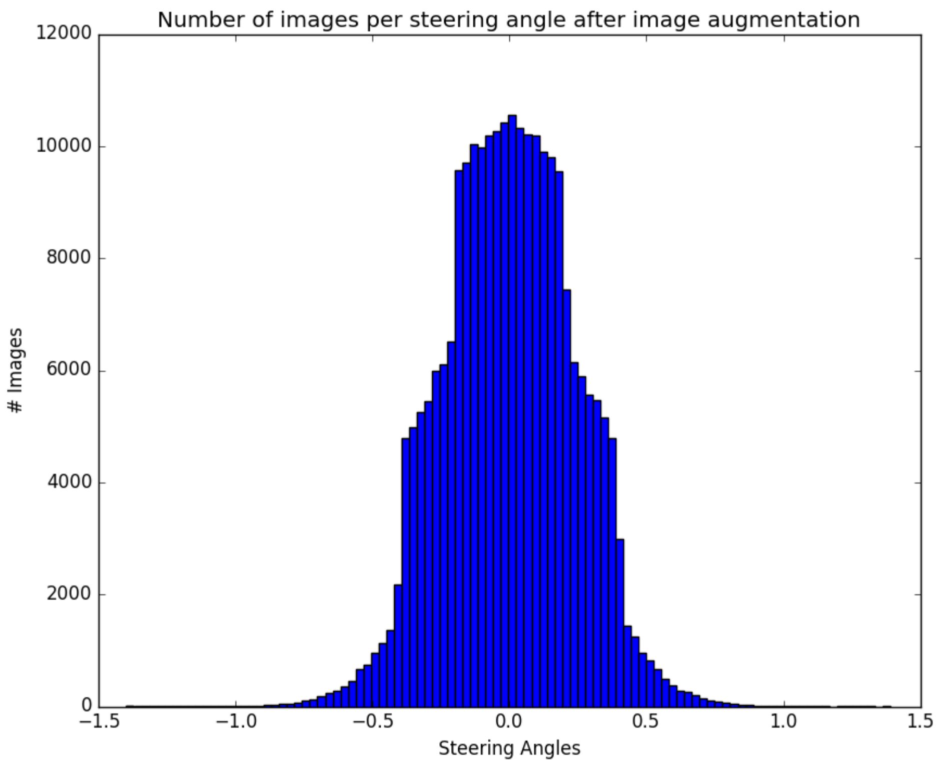
```
import keras
from keras.callbacks import Callback
import math

print("\nTraining the model ...")

class LifecycleCallback(keras.callbacks.Callback):

    def on_epoch_begin(self, epoch, logs={}):
        pass

    def on_epoch_end(self, epoch, logs={}):
        global epoch_steering_count
        global epoch_bin_hits
        global bin_range
        epoch_steering_count = 0
        epoch_bin_hits = {k:0 for k in range(-bin_range, bin_range)}

    def on_batch_begin(self, batch, logs={}):
        pass

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

    def on_train_begin(self, logs={}):
        print('Beginning training')
        self.losses = []

    def on_train_end(self, logs={}):
        print('Ending training')

# Compute the correct number of samples per epoch based on batch size
def compute_samples_per_epoch(array_size, batch_size):
    num_batches = array_size / batch_size
    samples_per_epoch = math.ceil(num_batches)
    samples_per_epoch = samples_per_epoch * batch_size
    return samples_per_epoch

lifecycle_callback = LifecycleCallback()

train_generator = generate_batch_data(X_train, BATCH_SIZE)
validation_generator = generate_batch_data(X_validation, BATCH_SIZE)

samples_per_epoch = compute_samples_per_epoch((len(X_train)*AUGMENTATION_FACTOR), BATCH_SIZE)
nb_val_samples = compute_samples_per_epoch((len(X_validation)*AUGMENTATION_FACTOR), BATCH_SIZE)

history = model.fit_generator(train_generator,
                              validation_data = validation_generator,
                              samples_per_epoch = samples_per_epoch,
                              nb_val_samples = nb_val_samples,
                              nb_epoch = NUM_EPOCHS, verbose=1,
                              callbacks=[lifecycle_callback])

print("\nTraining the model ended.")
```

The network was trained for only 10 epochs using 25,000 augmented example per epoch. Training takes roughly 10 minutes on a GTX970 GPU.

The histogram after data augmentation

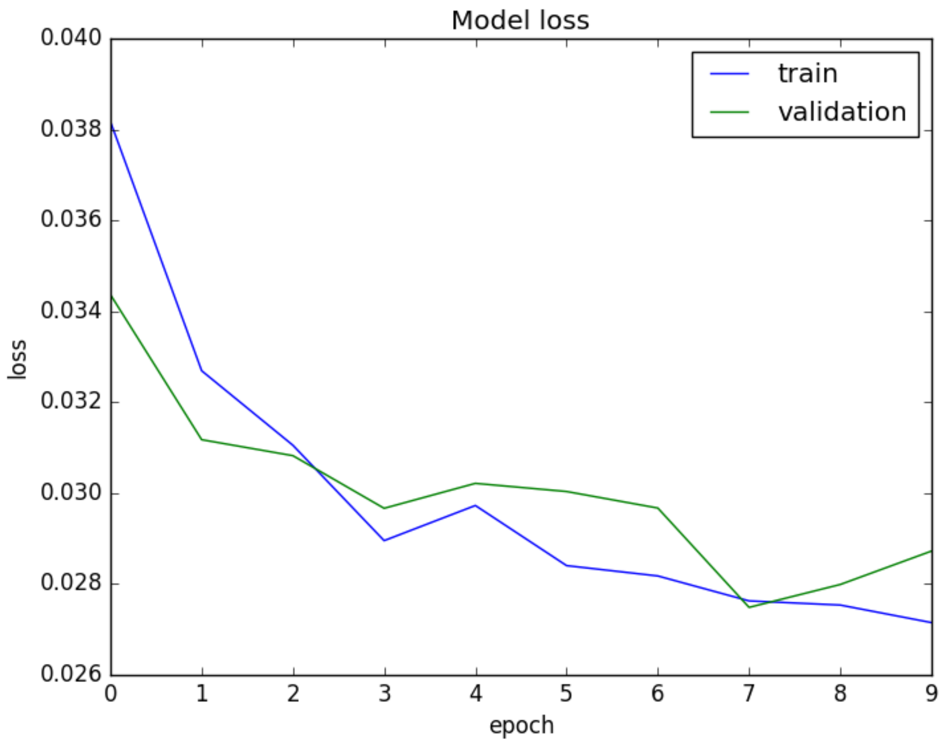
In the initial histogram we could see that the steering angles were mostly near the 0.0 angle. We noted then that, in order to teach our car to take turns, we have to generate many of new images with non-zero angles by augmentation.

The new histogram after augmentation is much better balanced.



The model loss

As shown in the plot bellow, both training and validation losses systematically dropped during the 10 training epochs. This is a clear sign that the model did not overfit the data and generalized well.

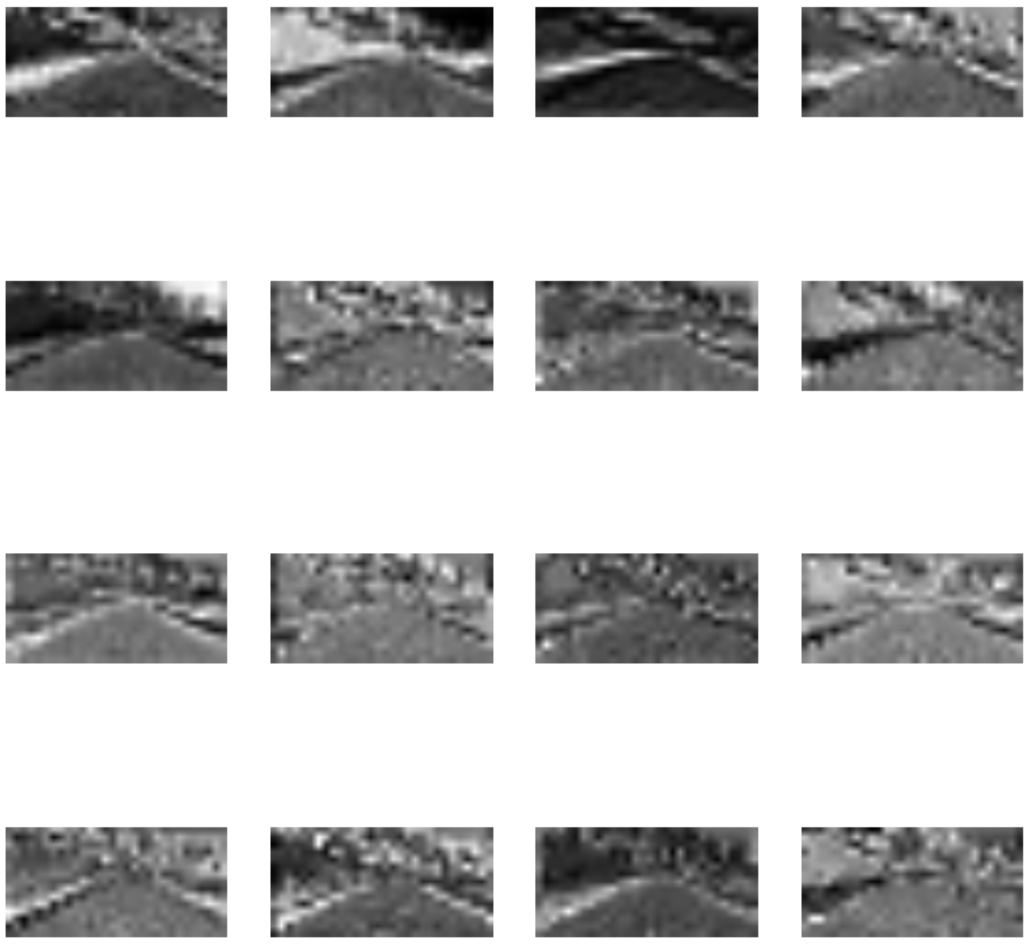


Visualization of convolutional layers

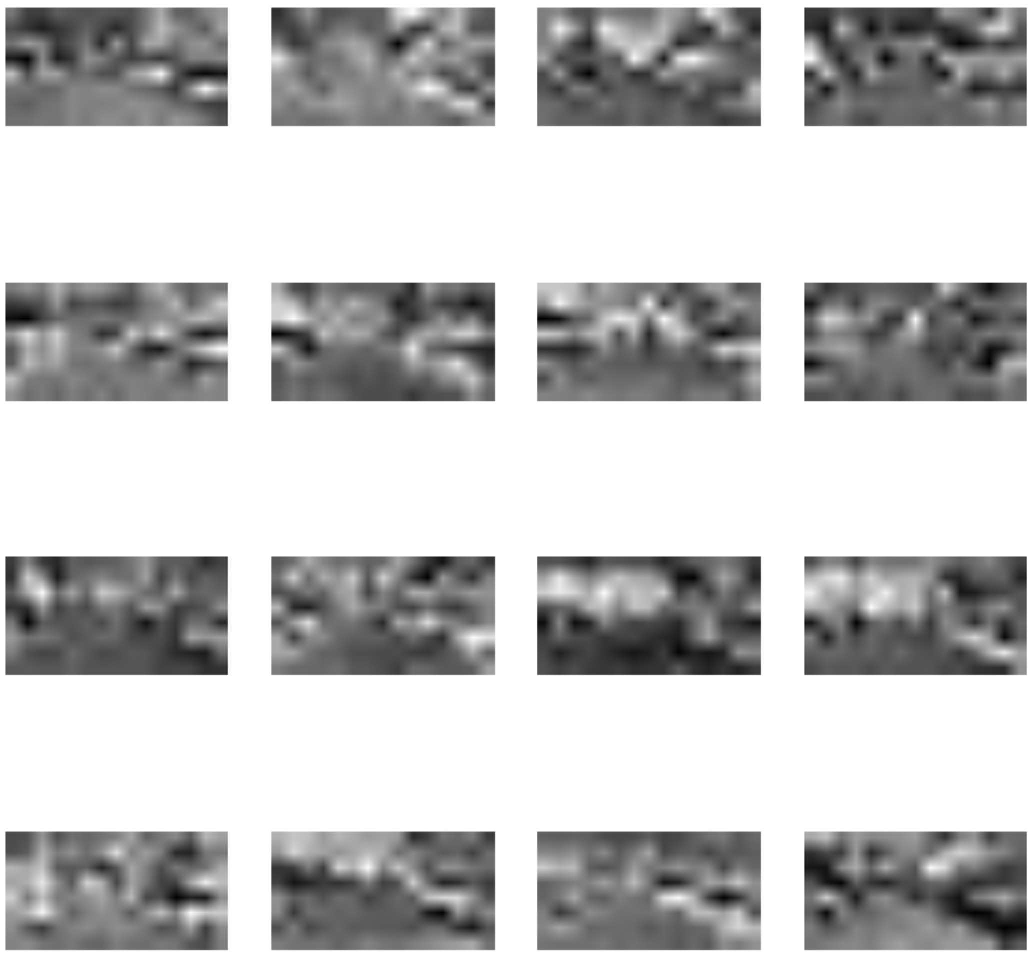
Bellow we can see what the first 3 convolutional layers learned.

```
##Visualization of the convolutional layers
test_file = X_test[0]['center']

def visualize_model_layer_output(model, layer_name, image_file):
    test_model = Model(input=model.input, output=model.get_layer(layer_name).output)

    image = load_img(image_file)
    image = crop_resize_image(img_to_array(image))
    image = np.expand_dims(image, axis=0)

    conv_features = test_model.predict(image)
    print("Convolutional features shape: ", conv_features.shape)

    # plot features
    plt.subplots(figsize=(10, 10))
    for i in range(16):
        plt.subplot(4, 4, i+1)
        plt.axis('off')
        plt.imshow(conv_features[0,:,:,i], cmap='gray')
    plt.show()

visualize_model_layer_output(model, 'Conv1', test_file)
visualize_model_layer_output(model, 'Conv2', test_file)
visualize_model_layer_output(model, 'Conv3', test_file)
```

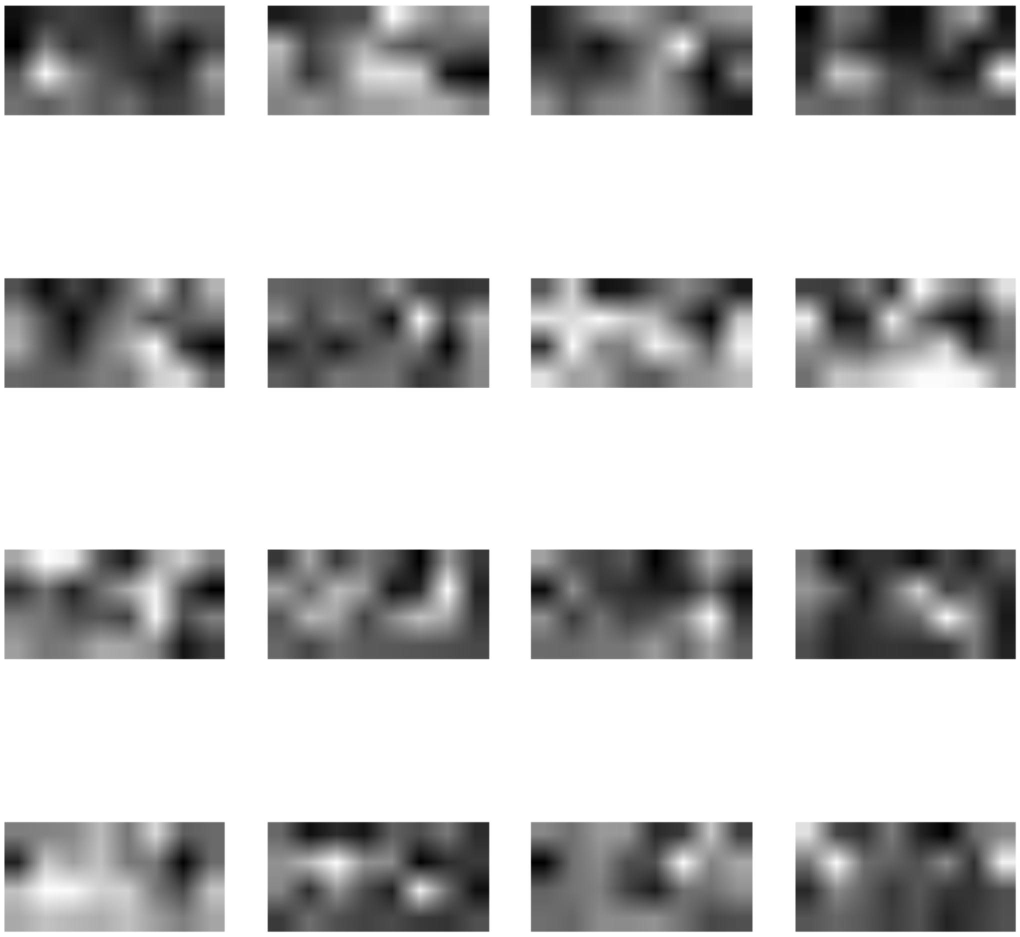
Conv Layer 1



Conv Layer 2



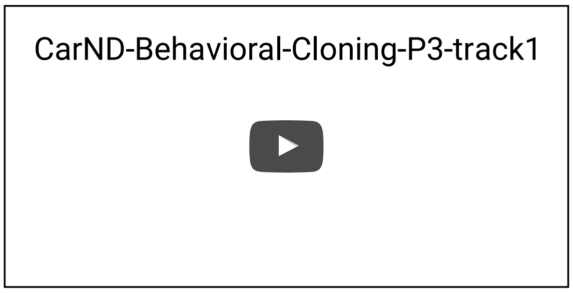
Conv Layer 3



Testing in simulation

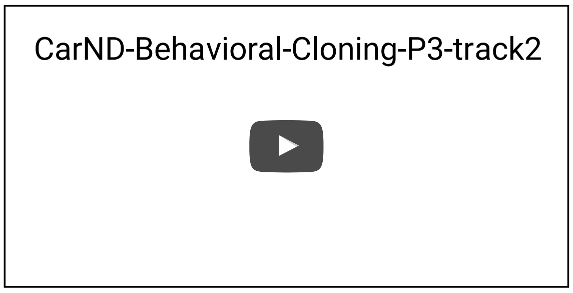
The best way to test the model is to let the car drive autonomously on both training Track 1 and test unseen Track 2. During the Track 1 the car drives very smooth and does not leave the road. In the case of unseen Track 2 it stays on the road most of the time, but it also hits the margins twice. The model was configured and trained good, but it still needs improvements to generalize better in the cases of unseen roads.

Track 1



CarND-Behavioral-Cloning-P3-track1

Track 2



CarND-Behavioral-Cloning-P3-track2