Udacity Self-Driving Car Nanodegree

Project 3- Behavioral Clong

Author: Johannes Betz

Date: 07/21/2017

Table of Contents

1. Introduction

The whole code described on the next pages can be found in the python file:

projectNN.py

With the described packages and python 3.5.2 the code can be run. My Github Repository contains the following files

- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- writeup_Betz_Johannes – My PDF Writeup

The goal of this project was to implement a convolutional neural network (CNN), train this with different datasets produced by a simulator and then run an autonomous vehicle in the simulator. This template describes the different steps to develop a CNN that can be run with the simulator.

2. Collecting Training Data

The first step in this project was the collection of training data. A simulator, were a vehicle can be steered is used to collect images and steering angles. The images are displaying the feature data, the steering angles the label data for the convolutional network.

Different test drives with the vehicles are made, so that different images and situations are collected. In the final database the following data was collected:

- 3 rounds around the track counter clockwise
- 1 round around the track clockwise
- A recovery lap, where the car is driven offside from the track and then back again
- 2 rounds around Track 2

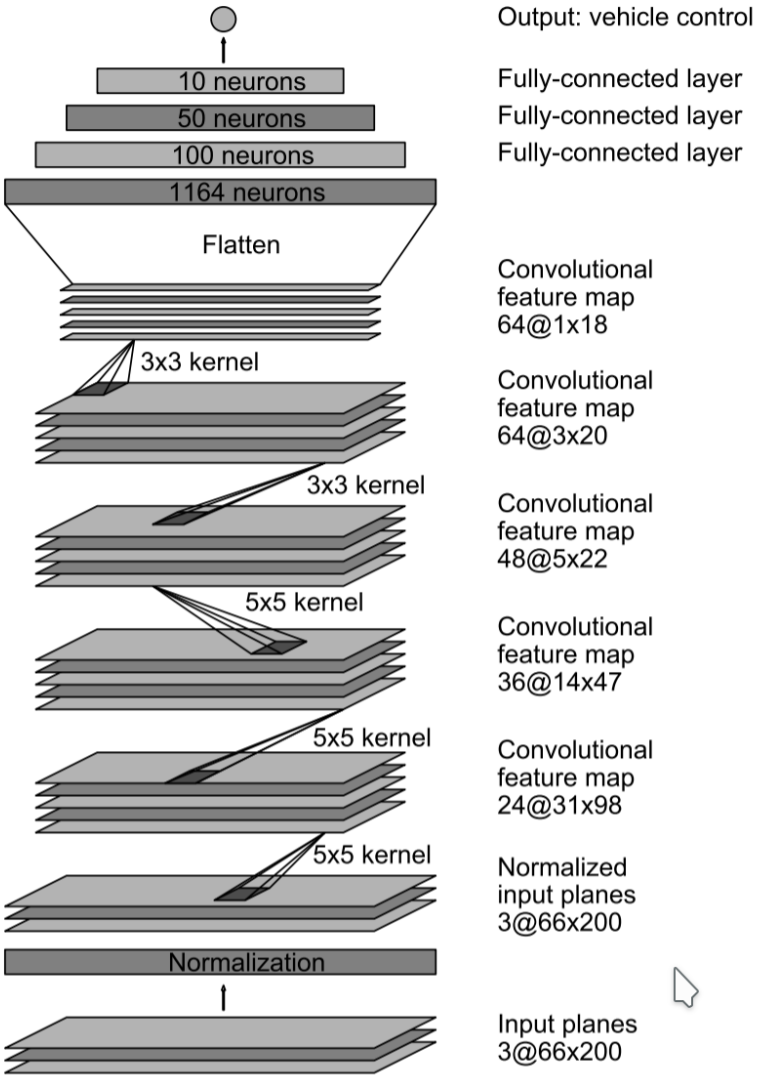
The images are divided in left, center and right camera pictures and are all used for training the network, the following pictured displaying used pictures for training the network.

To improve the data, the following techniques are used:

- Randomly choose right, left or center images.
- For left image, steering angle is adjusted by +0.2
- For right image, steering angle is adjusted by -0.2
- Randomly flip image left/right

3. Model Architecture

My Model is based on the CNN displayed in the NVidia paper. This model consists of 5 convolution Layers with additional 4 Fully connected layers.



The following code displays the CNN Models:

```
channels, height, width = 3, 160, 320  # image format
model = Sequential()
model.add(Lambda(lambda x: x / 255. - 0.5,
                 input_shape=(height, width, channels),
                 output_shape=(height, width, channels)))

model.add(Cropping2D(cropping=((70, 25), (0, 0))))
model.add(Conv2D(24, (5, 5), activation="relu", strides=(2, 2)))
model.add(Conv2D(36, (5, 5), activation="relu", strides=(2, 2)))
model.add(Conv2D(48, (5, 5), activation="relu", strides=(2, 2)))
model.add(Conv2D(64, (3, 3), activation="relu"))
model.add(Conv2D(64, (3, 3), activation="relu"))

model.add(Flatten())
model.add(Dense(100))
```

```
model.add(Dense(50))
model.add(Dense(10))
model.add(Dropout(0.5))
model.add(Dense(1))
```

After starting with a normalization in the lambda layer, the model is using 5 Conv2D layers. All Layers are connected with RELU layers to introduce the nonlinearity. After flatten the output of the Conv2D Layers, the the fully connected layers with dropout are used. The model is using an ADAM optimizer at the end for tuning and optimizing the model. The model used an adam optimizer, so the learning rate was not tuned manually

```
model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])
```

After the collection process, the following data points where used:

Total Samples : 69624

Train Sample Size : 62661

Validation Sample Size : 6963

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 Z as evidenced by ... I used an adam optimizer so that manually training the learning rate wasn't necessary.

, to reduce the overfitting of the model, the loss and accuracy where display and watched.

```
model.fit(X_train, y_train, validation_data = (X_val, y_val), epochs=2,
shuffle=True) #, shuffle=True, validation_split=0.1
model.save('model.h5')
```

Epoch 1/10

62661/62661 [==============================] - 715s - loss: 0.0395 - acc: 0.2008 - val_loss: 0.0303 - val_acc: 0.1986

Epoch 2/10

62661/62661 [==============================] - 630s - loss: 0.0325 - acc: 0.2013 - val_loss: 0.0268 - val_acc: 0.1991

Epoch 3/10

62661/62661 [==============================] - 634s - loss: 0.0281 - acc: 0.2020 - val_loss: 0.0245 - val_acc: 0.1991

Epoch 5/10

62661/62661 [==============================] - 627s - loss: 0.0258 - acc: 0.2021 - val_loss: 0.0248 - val_acc: 0.1991

Epoch 6/10

62661/62661 [==============================] - 628s - loss: 0.0240 - acc: 0.2022 - val_loss: 0.0215 - val_acc: 0.1993

Epoch 7/10

62661/62661 [==============================] - 629s - loss: 0.0228 - acc: 0.2024 - val_loss: 0.0206 - val_acc: 0.1995

Epoch 8/10

62661/62661 [==============================] - 629s - loss: 0.0214 - acc: 0.2024 - val_loss: 0.0191 - val_acc: 0.1996

Epoch 9/10

62661/62661 [==============================] - 633s - loss: 0.0204 - acc: 0.2024 - val_loss: 0.0188 - val_acc: 0.1996

4. Test and validation

To test the produced CNN, the model was save das an .h5 file and is then loaded into the tool again. This time, the car-tool is switched to the autonomous mode and is steered by the neural network. The solution for the the own produced .h5 file can be seen in the following videos:

Track 1: https://www.youtube.com/watch?v=vo9pM-g_4NA