Self Driving Car, Behavioral cloning writeup

you can find the codes in the repository: https://github.com/clockworks/nd-carnd-behavioral-cloning-p3

Autonomous driving from internal camera: https://vimeo.com/261647517
Autonomous driving from outside camera: https://vimeo.com/261647234

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, model_extra4.h5 containing a trained convolution neural network
- writeup_report.md or writeup_report.pdf summarizing the results
- run1.mp4 car self driving video

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_extra4.h5
```

3. Submission code is usable and readable

The clone.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.

I also added clone.ipynb file. You can run the code as a jupyter notebook file.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

I have used a neural network architecture which is very similar to nvidia end to end autonomous driving network architecture. You can find related paper's link below.

https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf

The network consists of 13 layers, including a normalization layer, 5 convolutional layers, 5 fully connected layers and 2 dropout layers.

The first layer of the network performs image normalization. I have taken advantages of keras to apply normalization in order to utilize GPU during normalization.

The convolutional layers are designed to perform feature extraction. In the first three convolutional layers with a 2x2 stride and a 5x5 kernel size. And, in the last two convolutional layers none stride and 3x3 kernel size.

There are five fully connected layers following the convolutional layers. The fully connected layer is designed to function as a controller for steering, leading to a control output which is the inverse turning radius.

2. Attempts to reduce overfitting in the model

The model contains two dropout layers in order to reduce overfitting.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually

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. I also used inverse direction driving data. I used sample data provided by udacity but I also generated driving data myself, especially for recovering from the left and right sides of the road.

Model Architecture and Training Strategy

1. Solution Design Approach

First, I flipped the images as the data augmentation in order to increase the training dataset and decrease the overfitting.

I not only used the camera image from center but also used camera images from right and left size. They are very helpful to recover from sides.

I also cropped the images. The front side of the car and the sky cropped. I left only the road view in order to increase the accuracy of the model.

In the begining, I used a basic architecture which is faster to train.

```
model = Sequential()
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(6, 5, 5, activation='relu'))
model.add(MaxPooling2D())
model.add(Convolution2D(16, 5, 5, activation='relu'))
model.add(Flatten())
model.add(Dense(120))
model.add(Dense(84))
model.add(Dense(1))
```

2. Final Model Architecture

After collecting data and training the model with this data. I realized car changing direction too often. So, I decided to use more convolutional and connected layer. Also, I would like to reduce overfitting so I added dropout layers.
I designed an architecture which is very similar explained in the following link. I only made some slight changes after experimenting with data

https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf

```
model = Sequential()
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(12, 5, 5, subsample=(2,2),
activation='relu'))
model.add(Convolution2D(18, 5, 5, subsample=(2,2),
activation='relu'))
model.add(Convolution2D(24, 5, 5, subsample=(2,2),
activation='relu'))
model.add(Convolution2D(32, 3, 3, activation='relu'))
model.add(Convolution2D(32, 3, 3, activation='relu'))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(100))
model.add(Dense(50))
model.add(Dense(10))
```

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

First, I normized the data. I used keras lambda in order to take advantage of gpu. Then I used several convolutional layers in order to extract the features. Afterwards, I used fully connnected layers in order to extract the controlling command for steering.

model.compile(optimizer='adam', loss='mse')

The problem is a regression problem. At the end we need only one value the steering angle. So I used minimum squared error as the loss function. I also used adam optimizer.

model.fit(X_train, y_train, validation_split=0.2, shuffle=True, nb_epoch=9)

3. Creation of the Training Set & Training Process

First, I drive the car three laps around track. Then I have drive from left and right sides of the road to the center. Then I drive the car to inverse direction.

For each frame, there are three camera image from different angles, center, right and left. You can see three images from different angles for the same frame.

image 1: left camera image

image 2: center camera image

image3: right camera image

I tried to train the architecture in the local computer but it takes too much time.

In order to take advantages of the gpu, I uploaded the training data to the aws ec2 instance and run the training there. Afterwords, I downloaded the model and run the simulation in the autonomous mode. In several places, the car went out of the road. So I collected more data and upload it the new dataset to aws. I followed this patterns many times.

After a few weeks, I could not managed to finish training the model. I realized that I should be doing something wrong.

problems:

1. Data size is huge, creating data size from scratch and uploading it to the cloud is too time consuming and expensive.

2. Generating good driving data is too hard. Even simple mistakes cause waste of time. I started creating driving data from scratch many times.

3. Training the data from scratch is really expensive and time consuming. Even, if I used the cloud ec2 instance it still last considerable amount of time.

So I changed the way I worked. I realized that the trained model is works for the most part of the tracks and only at a few points car goes out of the road. So, There is no need to start training from scratch. I followed the following patterns.

1. Collect training data while driving the car in the middle for both direction. save the model

2. train the model and run the autonomous mode and observe what is going wrong

3. reload the model and train them model with with more data, such as recovering from left and right sides to the middle or turning from sharp bend.

4. train the model with new data and run the autonomous mode and observe what is going wrong

5. follow this pattern until the autonomous driving mode satisfies all expectations.