Behavioral Cloning

Project Writeup Report by Krishna Swaroop

Behavrioal Cloning Project

The goals / steps of this project are the following:

- 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

Rubric Points

Here I will consider the rubric points (https://review.udacity.com/#!/rubrics/432/view) individually and describe how I addressed each point in my implementation.

Required Files

1. Submission includes all required files and can be used to run the simulator in autonomous mode

My submission includes the following files:

File Name	Description
model.py	containing the script to create and train the model
drive.py	for driving the car in autonomous mode
model.h5	containing a trained convolution neural network
writeup_report.ipynb & writeup_report.pdf	summarizing the results

Quality of Code

1. Submssion 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.h5
```

The model provided can be used to successfully operate the simulation.

2. Submssion code is usable and readable

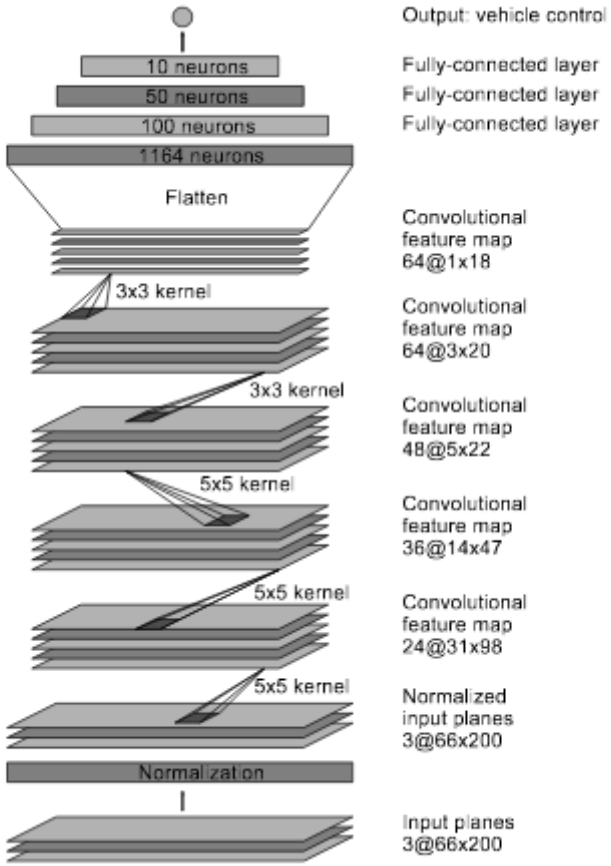
The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works.

The code in `model.py` uses a Python generator, to generate data for training rather than storing the training data in memory.

Model Architecture and Training Strategy

1. An appropriate model architecture been employed for the task

After some experimentation, I settled on NVIDIA model architecture, described in End to end learning for self-driving cars (https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf).



I used a variation of the architecture. The original NVIDIA has 5 convolution neural network with filter sizes 5x5 and 3x3, and varying depths (model.py lines 20-41)

The model includes ELU layers to introduce nonlinearity (`activation='elu'`), and the data is normalized in the model using a Keras lambda layer (code line 22).

I added a couple of convolution layers for recognising extra features and lane line colour independence (code lines 31-33).

2. Attempts to reduce overfitting of the model

The original model did not use any dropout layers. I added one dropout layer after my addtional convolution layers in order to reduce overfitting (model.py lines 33).

My initial attempts used a training and validation split. I eventually went for image augementation and since validation set was not adding any extra benefit (the objective was for the model to keep the car on track), I ended up not using the validation set.

The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Model parameters been tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 42).

4. Appropriate training data chosen

After several miserable attempts at trying to run the car in training mode to collect training data, I realised my extremely limited game-playing skills, and gave up. There were several reports in the Udacity forums of success using Udacity provided training data.

I used the Udacity training data along with image augmentation techniques. The left and right camara images, along with augmented images that shifted the image off-center were used to for training recovery from the road.

When the model is trained to reduce the mean squared error (MSE) loss to below 0.03, it is sufficient to induce the desired behavior in the simulation (i.e. keeping the car on the track).

For details about how I created the training data, see the next section.

Architecture and Training Documentation

1. Solution Design Approach

My first approach was to modify the simple Lenet architecture from Proejct 2 (traffic sign classifier). The early difficulty faced was to create a regression model rather than classification. This was solved by using linear activation in the final fully connected layer (model.py code line 40). I used Mean Square Error (MSE) as the loss function to minimise (code line 42). I tried several architectures with varying depths of convolution, pooling and dropouts. This experimentation took large part of two weeks.

I quickly realised that validation set accuracy doesn't count for much, whether the car remains on track was the larger objective. However, I did notice a correlation between low MSE and performance in autonomous mode in simulator!

I eventually settled on NVIDIA's architecture. It has 5 convoution layers with sub-sampling (hence no need to add any pooling layer). I stuck to the original NVIDIA architecture with did not have any drop out layers. I was able to generate sufficient training data for it to pass the first track.

The second track had gradients and initial constant throttle of 0.2 was not sufficient. I modified the driver code to adjust for current speed. I added options for minimum and maximum target speed and also addtional parameter to boost or damp throttle if the speed went out of range (drive.py code lines 35-38, function `adjust_throttle_for_speed`).

When the MSE was reduced to below 0.025, I noticed that the car did well on the second track too if the speed was limited to 20-30 range. I also noticed that the batch size used for training somehow had an impacted on trained model. Batch size of 256 gave the best results.

All was good until Udacity added a new track! It has white lane lines. The model was not able to recognise those and performed really poorly. To overcome this, I added two additonal convolution layers (one of 1x1) which will help recognise additional features and color independence on white lane lines.

* *The README thoroughly discusses the approach taken for deriving and designing a model architecture fit for solving the given problem.*

2. Final Model Architecture

```
    ##########################################################
    # The model is defined below.  It is a variation of NVIDIA architecture with a couple of
    # additional layers for recognising extra features and lane line colour independence.
    ##########################################################
    def nvidia():
        model= Sequential([
             Lambda(lambda x: x/127.5 - 1., input_shape=(66,200,3)),  # Normalisation
             Convolution2D(24, 5, 5, init='he_normal', subsample=(2,2), activation='elu'),
             Convolution2D(36, 5, 5, init='he_normal', subsample=(2,2), activation='elu'),
             Convolution2D(48, 5, 5, init='he_normal', subsample=(2,2), activation='elu'),
             Convolution2D(64, 3, 3, init='he_normal', subsample=(1,1), activation='elu'),
             Convolution2D(64, 3, 3, init='he_normal', subsample=(1,1), activation='elu'),

             # These are two additional convolution layers not in the NVIDIA architecture
             #    vvvvvvvvvvvvvvvvvvvvvvvvvvv
             Convolution2D(128, 3, 3, init='he_normal', subsample=(1,1), activation='elu', border_mode='same'),
             Convolution2D(256, 1, 1, init='he_normal', subsample=(1,1), activation='elu'),
             Dropout(0.25),
             #   ^^^^^^^^^^^^^^^^^^^

             Flatten(),
             Dense(1164, activation='elu', init='he_normal'),
             Dense(100, activation='elu', init='he_normal'),
             Dense(50, activation='elu', init='he_normal'),
             Dense(10, activation='elu', init='he_normal'),
             Dense(1, init='he_normal') # Linear Regression
             ])
        model.compile(optimizer='adam', loss='mean_squared_error') # ADAM optimiser, MSE loss
        return model
```

The model has following sequence of layers, *Orig* denotes layers from the original NVIDIA architecture:

1. Lambda layer for image normiazation
2. *Orig*: Convolution layer 5x5, with 2x2 sub sampling
3. *Orig*: Convolution layer 5x5, with 2x2 sub sampling
4. *Orig*: Convolution layer 5x5, with 2x2 sub sampling

5. *Orig*: Convolution layer 3x3, with 1x1 sub sampling
6. *Orig*: Convolution layer 3x3, with 1x1 sub sampling
7. *NEW*: Convolution layer 3x3, with 1x1 sub sampling
8. *NEW*: Convolution layer 1x1, with 1x1 sub sampling
9. *Orig*: Flatten
10. *Orig*: Fully connected layer dense to 1164
11. *Orig*: Fully connected layer dense to 100
12. *Orig*: Fully connected layer dense to 50
13. *Orig*: Fully connected layer dense to 10
14. *Orig*: Final layer dense to 1

3. Creation of the Training Set & Training Process

My mentor pointed me to the excellent article (https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9#.nrjpeprps) by Vivek Yadav. I will refer the reader to the article itself where the techniques are described, I simply acknowldge their use.

The following image augmentation techniques were used:

- Brightness augmentation
- Horizontal and vertical shifts
- Left and right camera images with steering angle adjusted by a constant in opposite direction
- Flipping images

I used a training image generator function that uses these techniques. It differs a bit from that in Vivek's article. I did not use shadow augmentation as described in the article referenced above. I flipped all images to overcome left turning bias in the first track, this differs frm Vivek's usage of flipping 50% of the images. I offset the steering angle by 0.25 when left or right camera image was used.

I did not use any data other than Udacity provided data. The model was able to drive on track 2 if speed is limited. In fact it drives a bit on the new third track too when speed is limited to be very slow, though third track has much sharper trurning angles, hence additional data will definitely help.

I am limited to the biases in the training set and too low MSE results in data model also mimicking the recovery runs int he training set which makes actual runs unstable. Since image augmentation applies random values for the operations, the training is not repeatable, i.e. different runs of training can result in different paths being taken. I didn't find very low MSE (~0.02) to be much better than around 0.024, hence I have used the model from an itermediate epoch from one of the training runs.

Simulation

1. Navigation on 1st track

Tire did not leave the drivable portion of the track surface. The car did not pop up onto ledges or roll over any surfaces that would otherwise be considered unsafe (if humans were in the vehicle). It drove several laps and was going strong when I had enough. I have been able to use many image sizes and several resolutions. An mp4 video is included in submission at ./examples/track1.mp4

2. Navigation on 2nd track (hilly track with shadows; as per original simulator)

When speed is limited to range, it was able to complete the run, video is included in submission at ./examples/track2.mp4.

When the quality is turned above Simple, the image resolution results in strong shadows that distract the model.

3. Navigation on 3rd track (new track recentely added to simulator, white lane lines)

I set the drive to very slow speed, 5-10 speed range. It drove the car to first 180 degree turn, without any additional training. Video is included in submission at ./examples/track3.mp4

```
python drive.py model.h5 --throttle .5 --min 5 --max 10
```