Behavioral Cloning

Behavioral 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

Introduction

In this project the objective was to autonomously drive a car in a simulated environment using deep learning principles. The simulator provided by Udacity provides two tracks and two running modes. One mode is designed for training and the second mode is for driving the car autonomously in one of the provided tracks.

In the training mode we can generate driving data, like images from the car dashboard and control data(steering angle, throttle, brake, seed), that can be used for training a CNN model with the help of Keras (a deep learning framework). The obtained model can be saved either as model.h5 (model weights) file or as json file(just model architecture).

After saving the model, we can use the drive.py file (given by Udacity and slightly modified to correspond to the requirements of my model) to start a local server to control the vehicle in the simulator when put in autonomous mode. The command I used to start the server is: python drive.py model.h5.

Files Submitted & Code Quality

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 containing a trained convolution neural network
- writeup_report.md or writeup_report.pdf summarizing the results
- video showing the autonomous driving of the car

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.h5
```

3. Submission 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.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

After the images were transformed to another colour space (YUV), cropped(such that only the bottom road region remained) and blurred to remove artefacts and resized I performed a normalization of the data at the start of my neural network using a batch normalization layer. From here I deepened my network to include more layers. I have used 5 convolutional layers, then a flattening layer followed by 3 fully connected layers. The last layer consists only of one neuron since the result should be only the steering angle output.

The convolutional layers have filters totalling 16,24, 36, 48 and 48 in this order. The kernel sizes are 3,3 and 2,2. The fully connected layers have sizes of 512, 10 and 1. I have trained my model for 10 epochs and used a batch size of 128. The model was compiled with an adam optimizer and a MSE error function.

I have placed a dropout after the first fully connected layer. The dropout was 0.5 for this layer which I realized allowed for a better convergence of the model while preventing overfitting. The activation used was relu since this activation is useful for finding non-linear relationships between layers.

```
model.add(BatchNormalization(axis=1, input_shape=(height,width,channels)))
model.add(Convolution2D(16, 3, 3, border_mode='valid', subsample=(2,2),
activation='relu'))
model.add(Convolution2D(24, 3, 3, border_mode='valid', subsample=(1,2),
activation='relu'))
model.add(Convolution2D(36, 3, 3, border_mode='valid', activation='relu'))
model.add(Convolution2D(48, 2, 2, border_mode='valid', activation='relu'))
model.add(Convolution2D(48, 2, 2, border_mode='valid', activation='relu'))
model.add(Flatten())
model.add(Dense(512))
model.add(Dropout(.5))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('relu'))
model.add(Dense(1))
```

2. Attempts to reduce overfitting in the model

To reduce overfitting I have added a dropout layer with a dropout of 0.5 after the fully connected layer with 512 neurons.

I have split my initial dataset into training, test and validation sets. I have also tested that my car runs well on the simulator.

3. Model parameter tuning

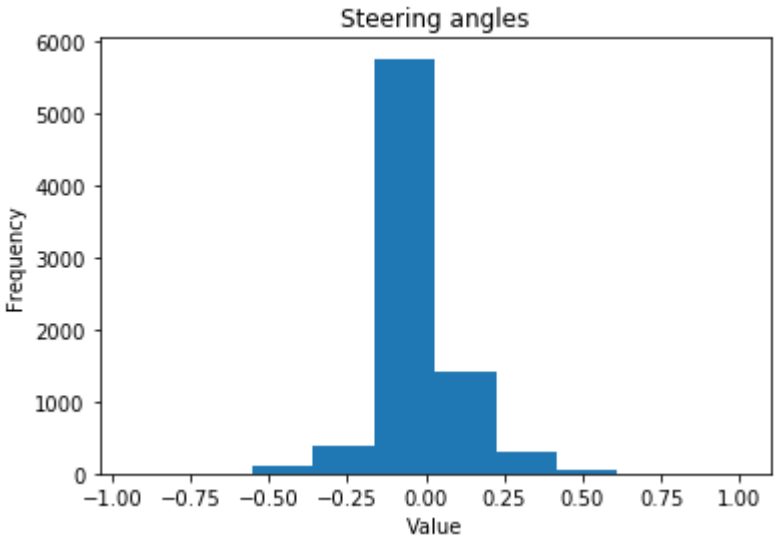
Several parameters were tuned within the project. The number of neurons in the fully connected layers was chosen based on the nVidia and coma.ai models. I have also adjusted the neurons manually so that I would obtain a lower error.

I have used an adam optimizer with a learning rate of 0.0001. The batch size selected is 128 and the number of epochs is 10. These parameters also were chosen such that I obtain the lowest possible error on the test and validation sets.

4. Appropriate training data

For training I have driven the model around the first track several times. I have also driven the car backwards.

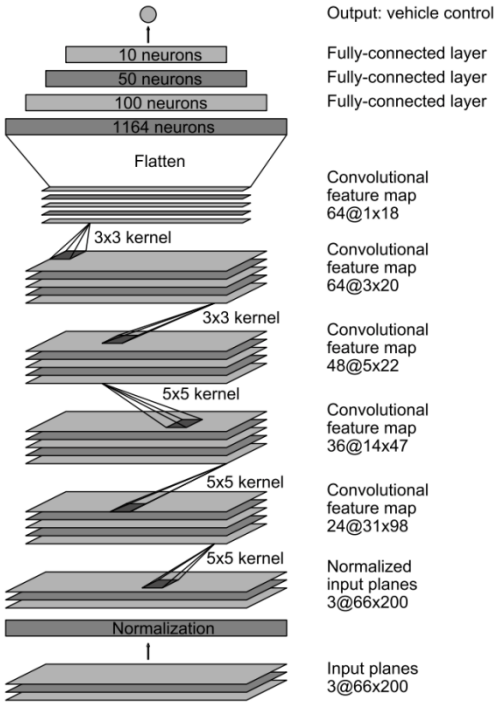
I have also explored the driving data given by Udacity. To my surprise the data provided by was sufficient to successfully drive the vehicle using my model around the track autonomously. Below I have shown an exploratory view of steering angle distribution of the data provided by Udacity.



Model Architecture and Training Strategy

1. Solution Design Approach

In the lecture notes from Udacity we were recommended to use a well-known self-driving car model implemented by nVidia (the paper can be found at this link (https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf).



The diagram of the model is shown above. So at the very beginning I started experimenting with this model in Keras. I have used this model as described in the paper, and I have also converted the original RGB images to YUV color space but the results were not very satisfying. I have split my original data set into 3 sets i.e. training, testing and validation sets. Due to the fact that I received a small error on the training set and a large error on the validation set, I have deduced that my model was overfitting. To combat this I wanted to use dropout in an efficient manner. After I have researched a bit I have found another model that can be used for training a steering model (https://github.com/commaai/research/blob/master/train_steering_model.py). My final solution used strategies from both models.

I have used adam optimizer, a learning rate 0.0001, and the chosen loss function was mean square error.

2. Final Model Architecture

The final model architecture can be seen in the table below. This model summary was generated using keras function model.summary().

```
Layer (type)                 Output Shape
=================================================
batch_normalization_1 (Batch (None, 33, 100, 3)
_________________________________________________
conv2d_1 (Conv2D)            (None, 16, 49, 16)
_________________________________________________
conv2d_2 (Conv2D)            (None, 14, 24, 24)
_________________________________________________
conv2d_3 (Conv2D)            (None, 12, 22, 36)
_________________________________________________
conv2d_4 (Conv2D)            (None, 11, 21, 48)
_________________________________________________
conv2d_5 (Conv2D)            (None, 10, 20, 48)
_________________________________________________
flatten_1 (Flatten)          (None, 9600)
_________________________________________________
dense_1 (Dense)              (None, 512)
_________________________________________________
dropout_1 (Dropout)          (None, 512)
_________________________________________________
activation_1 (Activation)    (None, 512)
_________________________________________________
dense_2 (Dense)              (None, 10)
_________________________________________________
activation_2 (Activation)    (None, 10)
_________________________________________________
dense_3 (Dense)              (None, 1)
=================================================
Total params: 4,948,949
Trainable params: 4,948,883
Non-trainable params: 66
```

3. Creation of the Training Set and Training Process

For creating the training set I have initially recorded 3 laps: the first lap using the was using the center of the lane(an ideal scenario), another lap was created with scenarios of recovering from left and right sides of the road and the last scenario was driving backwards.

Additionally I have also mirrored the images to increase the data set. The code for performing the mirroring and appending the mirrored data to the available dataset is given below.

```
mirror = np.ndarray(shape=(X_train.shape))
  count = 0
  for i in range(len(X_train)):
     mirror[count] = np.fliplr(X_train[i])
     count += 1
  mirror.shape

  # Create mirror image labels
  mirror_angles = y_train * -1
 # Combine regular features/labels with mirror features/labels
  X_train = np.concatenate((X_train, mirror), axis=0)
  y_train = np.concatenate((y_train, mirror_angles),axis=0)
```

All the images in the dataset have been scaled, cropped(the upper part – which contained the sky, and a bit of the lower part), transformed to YUV colour space and a 3x3 blur filter has been applied on them.

I had first split my data set into a 90% training and 10% test set. The training set was further split into a 90% training and 10% validation set. I used this training data for training the model. The validation set helped determine if the model was over or under fitting. I have trained my model for 10 epochs.

Conclusion and Discussion

In this project we had to solve a regression problem, of finding the correct steering angle, in the context of self-driving cars. The project had numerous phases, from data gathering, to model creation and evaluation and model refinement in case of under or over fitting. For creating the model two main references were used, one provided by the course instructors and one found in the discussion forums. The final result was a combination of the two approaches. I had to also modify the drive.py file so that the input to my model would be correct. I have used MSE as error function and it worked grate. I have recorded a video as explained in the course :

```
python drive.py model.h5 run1
python video.py run1 --fps 48
```

I have also created a new video with screen capture, since the first video had a bad resolution. The second video was created using the model with only the data set provided by Udacity. I submitted both videos.

I would have liked to see a similar approach in the context of moving obstacles. As future work I will try to modify part of the simulator (since the code is open source) to include some

moving obstacles (that would symbolize pedestrians). Due to the fact that the video sizes are too large to submit on the Udacity platform, I have uploaded them on youtube the links are:

1. https://youtu.be/Hy4xyzU0VmQ - For the video generated using the course commands

2. https://youtu.be/jQMLGrJcToA - This is the screen recording

I have also added 2 compressed videos, but if you find the quality too lacking please check the youtube versions.