Behavioral Cloning

Introduction

The aim for this project is make a Neural Network which can mimic human behavior to control car in simulated environment. For this Udacity has provided a simulator to collect data and test network on it.

The data collected is in form of images 160*320 with left, center and right sides. It also captures steering angles, speed and throttle values for each image. This is given as csv file for float values mapping to captured images.

For this I have used keras as frameworks. I have used Nvidia's model with updates and a RESNET inspired model for training. After training model on GPU, it is saved locally. To test it on simulator we need to load saved model and connect it. Run simulator in auto mode so that model can process images and predict values and send same to control car in simulator.

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

My project includes the following files:

- model.ipynb containing the script to create and train the model
- drive.py for driving the car in autonomous mode
- default_model_res.h5 containing a trained convolution neural network
- Writeup_up.pdf summarizing the results
- record_30.mp4 and record_60.mp4, for recorded video in auto mode.

2. Submission includes functional code

I have updated drive.py file to load model and start listening on port 4576 (simulator sends and receive data), it applies initial image transformation so that model can process it and predicts steering values. This is send to simulator to control car. Use below command to run drive.py file. This also captures screenshots so that it can be converted to video later .

```
py drive.py
```

Need to run simulator in auto mode and select track to check how model performs.

3. Submission code is usable and readable

The model.ipynb file contains the code for training and saving the CNN. There are 2 networks for this. First is inspired by Nvidia’s model and 2nd is inspired by RESNET which has shortcuts paths to propagate error deep in layers.

Model Architecture and Training Strategy

1. Model architecture based on Nvidia model (updated to more depth and layer filter size).

CNN Model of depth 25, with below architecture. This layer has no shortcuts to propagate error to lower layers. (Cell in 5 in model.ipynb)

1. Con with 32 filters of kernel size 7, padding as valid with 1 stride. Input shape is 80*320. Activation as ELU (Better than RELU)
2. A batch normalization layer.
3. Con with 32 filters of kernel size 7, padding as valid with 1 stride. Activation as ELU (Better than RELU)
4. A batch normalization layer.
5. A MAX pool layer with kernel size of 2.
6. Con with 64 filters of kernel size 5, padding as valid with 1 stride. Activation as ELU (Better than RELU)
7. A batch normalization layer.
8. Con with 64 filters of kernel size 5, padding as valid with 1 stride. Activation as ELU (Better than RELU)
9. A batch normalization layer.
10. A MAX pool layer with kernel size of 2.
11. Con with 128 filters of kernel size 3, padding as valid with 1 stride. Activation as ELU (Better than RELU)
12. A batch normalization layer.
13. Con with 128 filters of kernel size 3, padding as valid with 1 stride. Activation as ELU (Better than RELU)
14. A batch normalization layer.
15. A MAX pool layer with kernel size of 2.
16. Con with 128 filters of kernel size 3, padding as valid with 1 stride. Activation as ELU (Better than RELU)
17. A batch normalization layer.
18. Flatten layer to convert node to linear ANN
19. A dense layer with 200 nodes and activation as ELU
20. A dropout layer with dropout of 30
21. A dense layer with 200 nodes and activation as ELU
22. A dropout layer with dropout of 30
23. A dense layer with 100 nodes and activation as ELU
24. A dense layer with 50 nodes and activation as Linear
25. A dense layer with 1 nodes and activation as Linear for final steering values.

2. RESNET inspired model, for with shortcuts for error propagation.

RESNET has 2 types of blocks which are placed back to back in model architecture. There are shortcuts added in each block for propagating errors to lower layers so that dying gradient problem can be solved. First block is 3 layers of conv operation added to initial input while second block is 3 layers of conv and input added after and other conv operation. For this 2 helper functions are defined identity_block (cell 9) and convolutional_block (cell 10). Using these blocks a network with depth of 100 layers is created (cell 11).

1. It takes input with shape of 80*320.
2. Conv layer 64 filters of size 7 with strides of 2.
3. Batch normalization
4. Activation layer RELU.
5. Max pooling with kernel size as 2 and strides as 2.
6. Then a convolutional_block
7. 2 layers of identity_block
8. Then a convolutional_block
9. 3 layers of identity_block
10. A max pooling layer.
11. A average pooling layer.
12. 3 layers of FC network with 128 nodes.
13. FC layer with 64 nodes.
14. Final output layer.

2. Attempts to reduce overfitting in the model

Model 1 has 2 dropout layers in line 54 and 56 of cell 5.

Model 2 does not have any dropouts as RESNET does not have have it. Shortcuts in error back propagation handles overfitting.

3. Model parameter tuning

Both models use ADAM as optimizer for training so learning rate is adjusted automatically.

4. Appropriate training data

For training data, I have used default data from Udacity, and collected data locally with 6 laps 3 in forward direction and 3 in backward direction to avoid any bias of left/right turns on track.

Model Architecture and Training Strategy

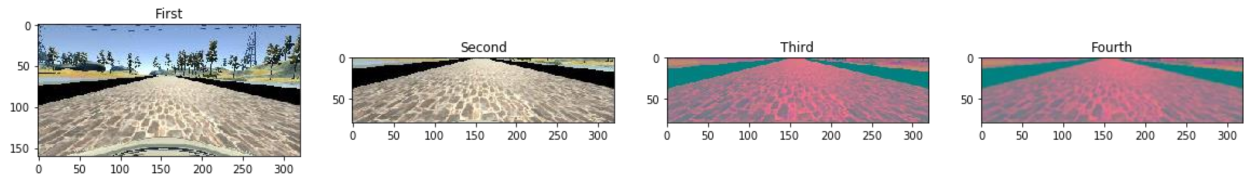
1. Solution Design Approach

For image processing I have created 2 pipelines. Following are steps of pipeline

1st Pipeline

1. Crop image to required size of 80*320
2. Change color space from RGB to YUV
3. Add gaussian blur to image to reduce high frequency of colors.

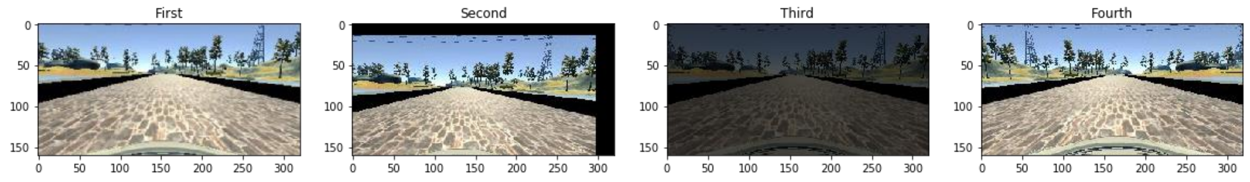
Original imager, followed by cropped image, YUV transformation and final with gaussian blur.



2nd Pipeline, apply random transformation to images.

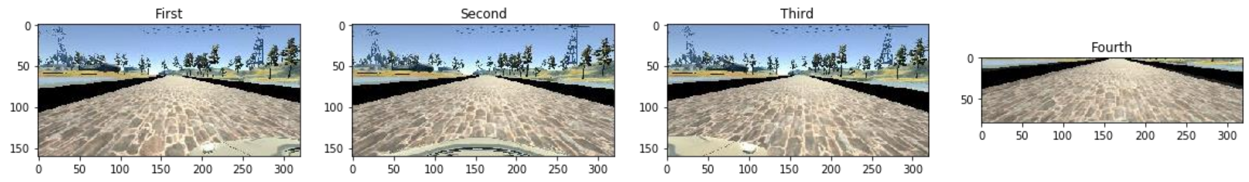
1. Zoom from 0.9 to 1.1
2. Randomly squeeze images from 90 to 110 in both X and Y
3. Randomly increase brightness of images.
4. To generate more data, we can flip images and add negative steering to our list.

Zoomed image, randomly squeezed image, random brightness changed and flipped image.

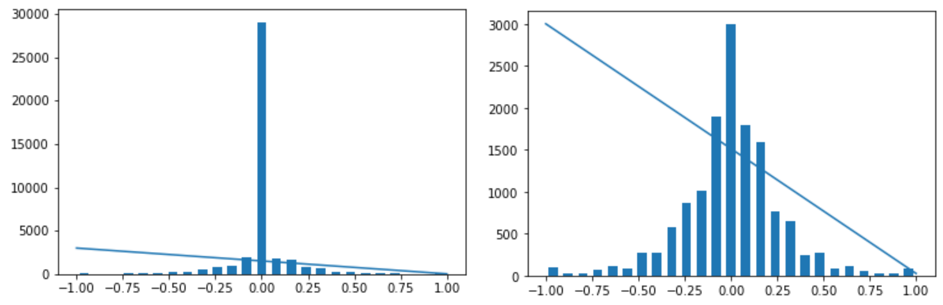


Apart from these 2 pipelines, I have also used one more image transformation in which I have taken center and left/right images in 80:20 ratio and then added both left_center and right_center with 50:50 ration to create a new image.

Left image, followed by center image and right image. Final image is last in row.



To check for distribution of steering angles I have plotted histogram. On left we can see that nearly ~30K images have 0 as steering angle while for others that count much less than 1000. To overcome this bias, I have limited 3000 as cap for images in each bin. From selected data histogram on right we can see it is now more towards normal distribution.



To generate more data from collected data I have applied below steps

1. Combine left/center/right images into single image.
2. Apply random transformation (Zoom/Squeeze/Brightness) on all images.(pipeline 2)
3. Apply crop/color space conversion and gaussian blur on original image.
4. Combine all above 3 into one set.

This leads to 3 times more data than our collected data.

2. Final Model Architecture

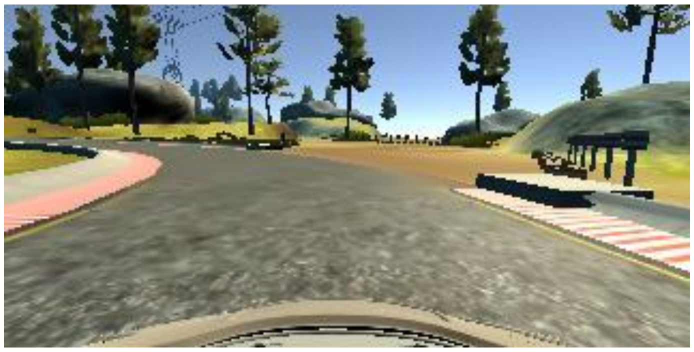
The final model which I have used to capture video recording is RESNET inspired model. It has identity block in cell 9 and conv block in cell 10 of model.ipynb. Model using these to functions is created in cell 11 of same file. This model is about 100 layers deep and while training I was able to get MSE error to as low as 0.024. This model has more than 2M training parameter.

Please check cell 11 of model.ipynb, as it has complete architecture of chose model.

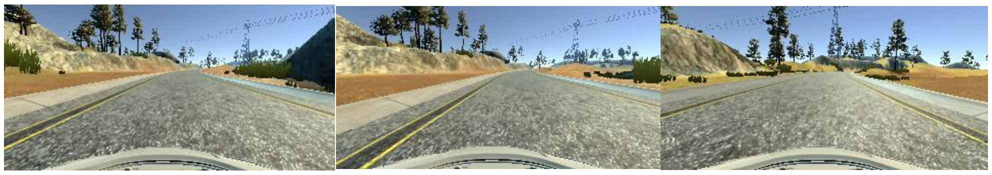
3. Creation of the Training Set & Training Process

To capture data for training, I have recorded 6 laps in track 1. 3 in forward direction and 3 in backward direction to avoid any bias for left/right turns.

From normal lap:



From reverse lap to avoid bias for left/right turns



After collecting data from local run, I added data shared by Udacity. Then I applied pipeline 2 (random transformation) and image addition to increase my data set size. This increased my data set size 3 folds. Then before feeding this to model I applied pipeline 1 to all collected images and converted it to format which can be used by my model. Total there were 42072 data points and divided into 35761 and 6311 parts as train and test data.