Imitation Learning for Carla Simulator Using an End to End Driving Network

This is a documented approach for my efforts to train a vehicle to clone behaviour trained using a NVIDIA end to end deep learning model for an autonomous vehicle.

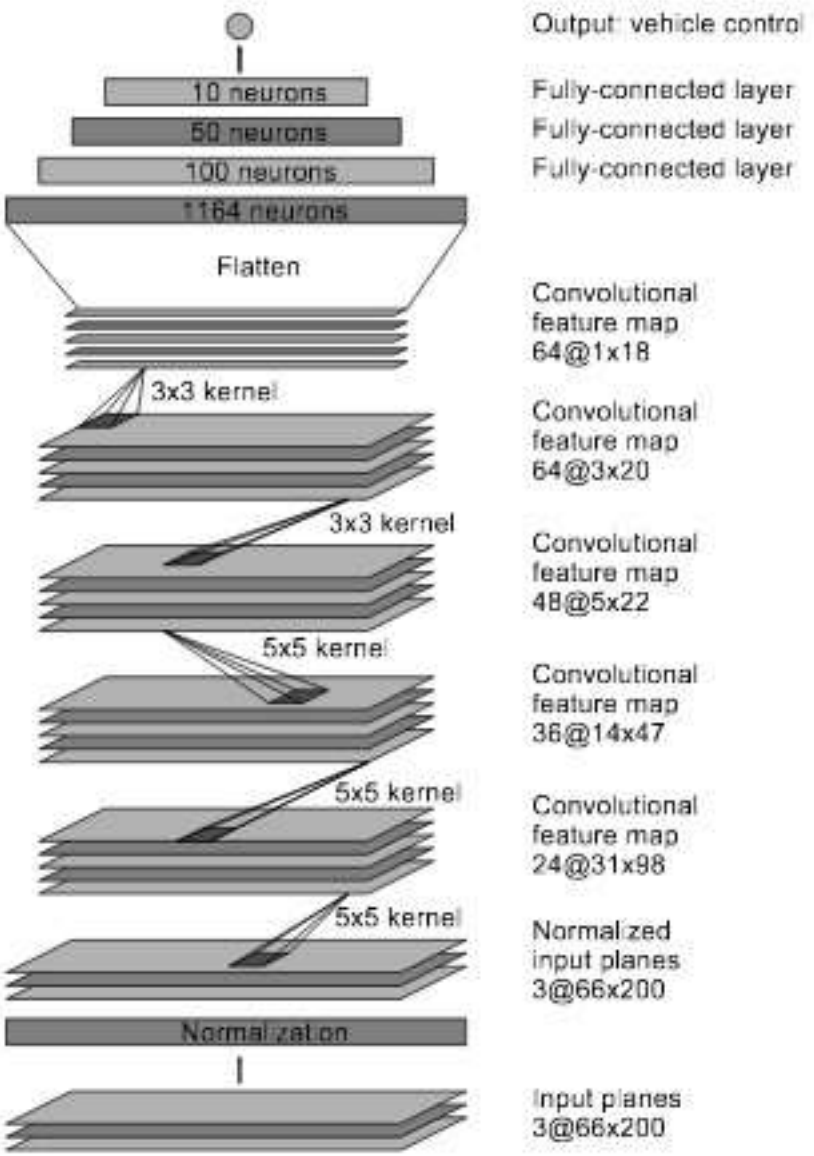
The contents of the documentation is as follows.

- Motivation and Brief Introduction.
- Behavioural Cloning on the Udacity Simulator.
- Introduction to the Carla Simulator and the code architecture.
- Added Scripts to extract information from the Carla Client.
- Comparison of performance between RGB, Depth and Semantic Segmented Images.
- Current Scenario and future prospects.
- Brief ending with the final Python Client Structure

1 Introduction

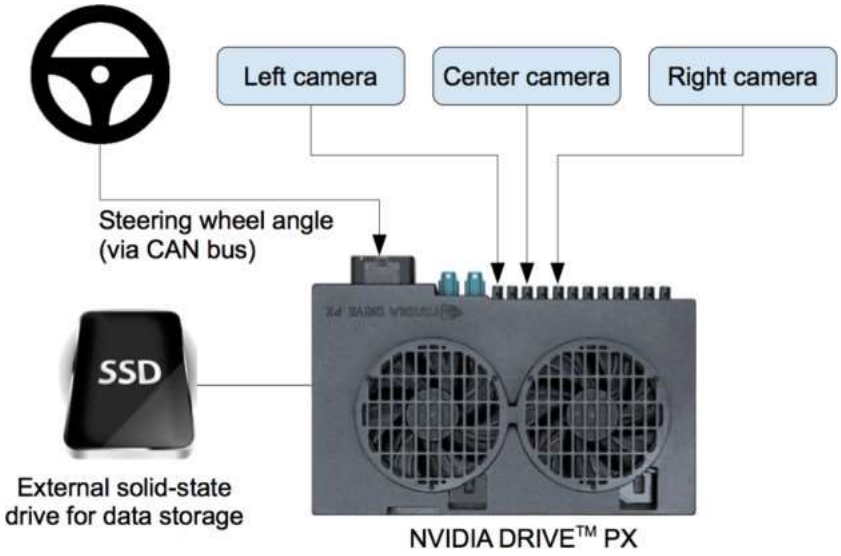
The Neural Network used to train our autonomous vehicle is a NVIDIA CNN model with a little tuning. It starts with a normalisation layer, goes ahead with few convolultion layers and ends with Dense layers to give an output.

The model i have used involves Batch Normalisation and Relu activation.



The solution to higher level regression problems can be found in classification problems and this network is a clear example of this.

The input to this model is the image as seen by the camera and the label is the steer value of the car in degrees.



2 Behavioural Cloning on the Udacity Simulator

The first step to this Project is collecting data by driving on the car on the Simulator in training mode.

There are various tactics to be kept in mind while collecting data as follows

- the car should stay in the center of the road as much as possible
- if the car veers off to the side, it should recover back to center
- driving counter-clockwise can help the model generalize
- flipping the images is a quick way to augment the data
- collecting data from the second track can also help generalize the model
- we want to avoid overfitting or underfitting when training the model

The network used is the end to end learning CNN as mentioned above.

Each image is associated with two other images with 2 other images, left and right. A driving log is provided which gives values of steer,throttle, break and speed. Each image is 160 pixels and 320 pixels wide.

The image is augmented and cropped to mask out only the part of the road. The image is flipped and resized.

A toplevel class is maintained which takes care of every aspect

```
class TopLevel:
    def __init__(self, model, base_path='', epoch_count=4):
        self.data = [] self.model = model self.epochs = epoch_count #tune this
        self.base_path = base_path self.correction_factor = 0.2 self.image_path =
        self.base_path + '/IMG/' self.driving_log_path = self.base_path +
        '/driving_log.csv' self.training_samples = [] self.validation_samples = []
        self.batch_size = 128 #tune this
```

The model is compiled with **ADAM** optimizer and the loss function is **MSE**.

The training set is split into train set and validation set with a 8:2 ratio. Tuning is done to take care that the validation loss doesn't exceed at max 2 time the train loss. Generators are used to fit the data(model fit generator used instead of model fit) because yielding with such large datasets is better than returning. In total 4 epochs are trained with each having approx 5000 images

The server sends real time images to the client which sends the steer controls to the car. The throttle/speed value is kept constant.

The car completes one full lap but crashes on the bridge, it might be because the texture of the road is different than the usual road. Creating extra data for the bridge

and converting the BGR images sent by the simulator and converting them to RGB and training solves the issue of the bridge.

This model steers the car perfectly through out the whole lap.The maximum steering angle is around 6 degrees so it was relatively an easier simulator.

3 Introduction to Carla

The basic idea is that the CARLA simulator itself acts as a server and waits for a client to connect. A Python process connects to it as a client. The client sends commands to the server to control both the car and other parameters like weather, starting new episodes, etc. The server (i.e., the simulator) sends measurements and images back to the Python process. The Python client process can then print the received data, process it, write it to disk, etc. By default all the communication between the client and the server happen on TCP ports 2000, 2001 and 2002. The messages sent and received on these ports is explained here, but it is not very important to understand everything over there, as most of the client-server communication is abstracted by the carla module in the PythonClient directory.

The two scripts of importance is client example.py and manual control.py. Both of these sends controls to the server via TCP connection. Manual control provides a pygame window with RGB and Semantic Segmented images of the camera view.

Inside the Python Client, in the directory carla/ There are various classes such as sensor class which describes the sensor or the different types of cameras along with an image converter.py which converts various different types of images into viewable form such as converting the Semantic Segment labels into an image in the R channel.Other images such as the depth images give different weightages to different channels of RGB.

I have made various new scripts:

- client example2.py - it sends the controls to the simulator after the model predicts the steer value.

- manual control rgb semseg.py - it extracts the images generated by the semantic segmented post processing stored into a .npy file.

- buffered saver.py - it is a buffer class imported in the manual control, this buffer stores images to the ram rather than the disk.

- augmentation.py - this script augments and pre processes the data befoere it is trained by the neural network.

- sensor.py - this script has been edited with various helper functions added to the sensor data class to extract real time values returned by the server.

4 Comparison of Performance

The carla camera returns three types of images- RGB, Depth and Semantic Segmented Images.

the sensors return a carla.Image object.

The depth image codifies the depth in 3 channels of the RGB color space, from less to more significant bytes: R -> G -> B. The actual distance in meters can be decoded with

normalized = (R + G * 256 + B * 256 * 256) / (256 * 256 * 256 - 1) in_meters = 1000 * normalized

The semantic segment image is an image from the camera classified into 10 different types of objects, it felt really ideal to train the neural net on because it identified different segments of an image before hand like the lane and the pavement so it seems convincing to let the car stay in the lane when trained on this.

When training on RGB various set of augmentations had to be done before the network could be trained which will be listed below.

Figure 1: Depth Image

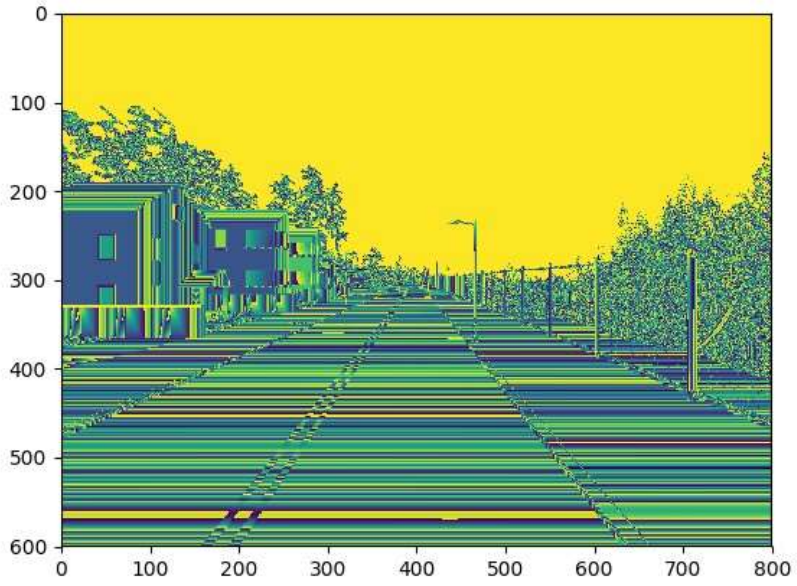
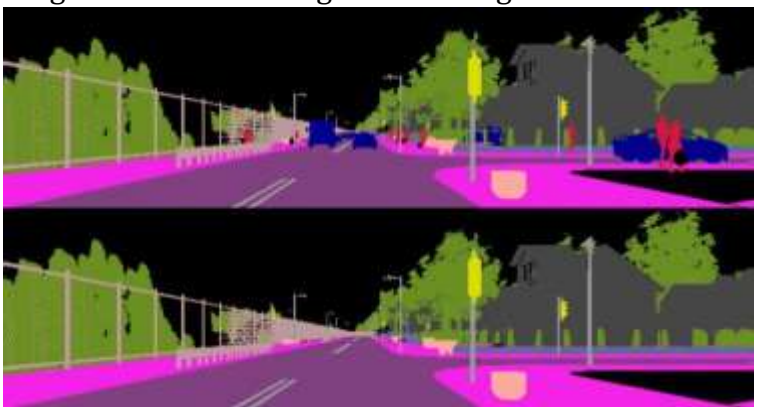


Figure 2: Semantic Segmented Image



4.1 Training on RGB images

RGB data is generated by saving images by running CARLA in server mode and running python3.5 client_example.py --autopilot --images-to-disk

A directory out is created with various episodes of the simulation with RGB and Depth Images. The simulation runs at 3 fps because saving the images to the disk is an expensive process.

Preprocessing is done with a script augmentation.py which does cropping of the image, resizing it, creating flips of images and the most important converting BGR to RGB images. A new directory is created by the name DATA/.

The input now is the images stored in the DATA/ and the training labels is stored in a text file steer.txt which has driving log values for the sim. Flipped images are labelled with -1*(steer value of straight image)

I trained the model for 10 epochs after which the training loss remained almost same.

To drive the vehicle on the model prediction, we use the keras.models.load_model()

function. It is adviced to remove the normalisation layer out of the layers in the keras model because otherwise the model doesn't load and gives a segmentation fault. It is adviced to preprocess the image with normalisation using cv2.normalize() and then feed it into the Neural Network. It is a high computation task so normalizing a bunch of 5000 images takes roughly 2 hours. But it was very useful because sending un-normalised images vs normalising changed the game.

An argument –imitation-run was added which allows the car to steer on what is predicted by the model. In the carla/sensor.py , in the class SensorData i have added a function return image() which returns the image from the camera at every frame.

The sensor data and the measurements in the client example2.py include all the values of the sensor cameras along with the values associated with the car. The single image is made into a numpy array and resized into

[1]dimensions. Then the image is normalized using cv2.normalize() and the image is now input into the model.predict() function of keras.

The car provided by CARLA 0.8.4 has a maximum steer of 70 degrees on one wheel. Whenever the model predicts a steer value out of the bounds of the car, the steer is set to a fixed value ie. 70 degrees and an ”Out of bounds” alert is printed. The model trained by me has this limitation where the car is not able to make sharp turns because the steer value predicted is really high which is not practically possible.

Any low value of steer of the order of -0.001 to 0.001 is set to 0 degrees to allow the car to stay static at a red light or when it has stopped.

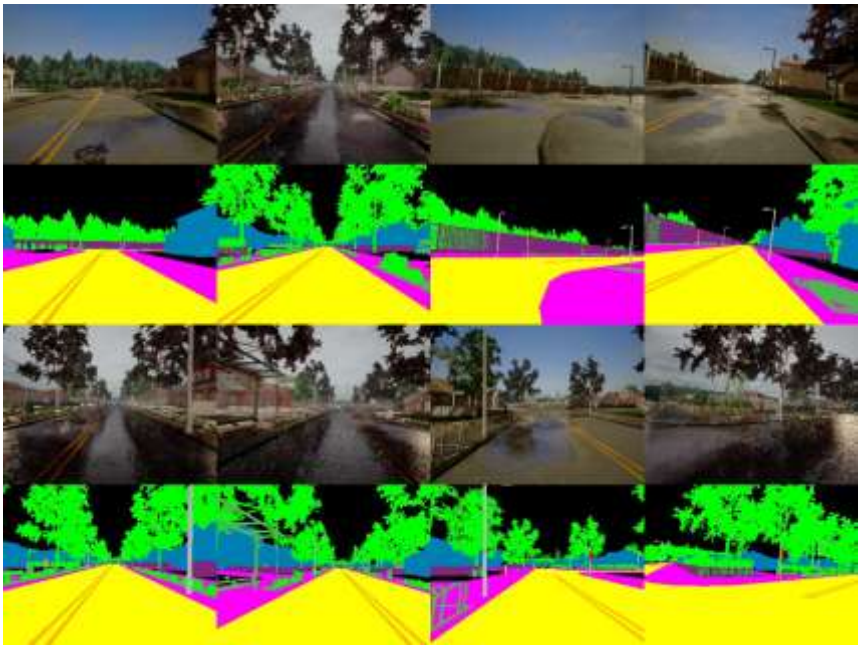
The car when runs perfectly on a straight road in most weather conditions with RGB excluding weather with shadows that change the behaviour of the car and make it perform a turn. The turnings is a bit of a problem where really sharp turns on the corner of the outer road is not able to be executed perfectly and steer goes out of bounds. Training on depth images has been pretty much the same. The results with depth images have been worse than RGB hence discarded.

Figure 3: RGB Image vs Semantic Segment Image

[1] .2 Training on Semantic Segmented Images

Training the network on Semantically Segmented Images seems obvious because of the clear division of different objects in an image which allows the network feature maps to make understand the components of the image much better. My effort in training with segmented images has not been up to the mark and the reason is the bad results obtained during the initial testing.

The first script written to extract segmented images is the manual control rgb semseg.py. It is the modification of the manual control.py. On running this script, a blank pygame window will open and the images of the run will be saved in the episodes depending on the location you want or the None/ directory as a default.



Finally, since I eventually want to train a neural network with the collected data, it would be really convenient if all my collected data were stored in numpy arrays. Then I would not have to open thousands of .png files and read them into memory. Storing and retrieving the data in bulk would also be very easy because there would be no need to encode/decode from the PNG format, and besides, both opencv and matplotlib work with numpy arrays under the hood, so it does not make visualization any harder. Use Jupyter Notebook instead to view the image using plt.imshow(display_array[:, :, 0], cmap='tab20', aspect='auto')

Steps to run manual control along with the carla server running are as follows

python3.5 manual_control_rgb_semseg.py --images-to-disk

--location=<save_location>

Detecting Lanes, Road Edges and other vehicles is an easy task with these images, a combination of these used in the neural network instead of only the images can do wonders in this job.

To improve the process of training and running the car fully autonomously, another input should be ideally given which describes the current state and orientation of the car and not only the images of the car. This might be an extra feature left to implement and thoughts should go as to how to integrate the state with the images.

5 Current Scenario and future prospects

The highest autonomousity achieved is in the case of RGB images as compared to other types of images. The car makes slight and not so sharp turns but fails to complete a sharp turn and moves successfully on straight roads in almost all weathers except for those with shadow.

It is at a good enough stage for the model to be used as a benchmark and the weights to be quantized using various quantization techniques and to be integrated back with the model to check the difference in performance with and without quantization. The next step would be to import alphaRT in the client example2.py and to create a session environment in alphaRT and let it do the predictions instead of the model.predict of Keras.

The Carla client has a benchmarks script which creates various sessions of the simulation and returns various benchmark results. Corl2017 paper has said that there are around 24 different benchmarks out of which my simulation completes close to 9/10 episodes. There is major scope of improvement in this.

Computer Vision techniques can be integrated with the Semantic Segmented Images to train it better. To improve the process of training and running the car fully autonomously, another input should be ideally given which describes the current state and orientation of the car and not only the images of the car. This might be an extra feature left to implement and thoughts should go as to how to integrate the state with the images.

Figure 4: Turning on Udacity Simulator

Figure 5: Turning on Carla Simulator

Finally a snapshot of the car running and taking a (somewhat unsuccesful) turn on a sunny day environment. This was one of the tougher turns in /Game/Map/TownMap02 and the car went out of bounds at a stage but still completed the turn succesfully at the end. Note that the control of the speed/throttle and it's interaction with different cars and people ahead of itself has not been trained and the controls sent to the client is carla's autopilot itself and has not been tweaked with.

————————————————–THE END————————————————-