

Behavioral Cloning

Self Driving Car Udacity NanoDegree. Project 3

Pablo Bermell-Garcia, May 2017

pbermell@gmail.com

0. Introduction

This document reports on the work carried out to implement a behavioral cloning system from the learning gained in the Udacity's Self Driving Car Nanodegree course. The project focuses on the implementation of deep learning and computer vision techniques enabling the autonomous drive of a car in a simulated environment. The key learning outcome of this project is the ability to exploit a trained convolutional neural network into a real time computer vision application to drive the car using the resulting model. The other main take from this project is the importance of the data used to train a model. Finally, the project is a great way of consolidating many of the concepts we have studied so far in the course. This report captures key insights gained on the project. Execution required extensive inputs from the Udacity community since it had some uncertainties. Remarkably these are: the fact that the loss does not correlate very well with the performance of the driving; and the difficulty to capture driving data without an analog joystick as pointed out in the forums. As in my previous project, I used Jeremy Shannon's implementation as a starting point for my own, [1]. Jeremy's implementation includes various OpenCV transformations and jitterings to the input images. However, it recognises the importance of the data to produce satisfactory results. Based on this, the working hypothesis is that I could train the model with the minimum amount of data and image transformations. This hypothesis is successfully validated in the project which uses only the dataset provided by Udacity and no jittering.

0.1 Implementation platform

The project was implemented on an Acer Aspire v15 Nitro – Black Edition Laptop, (Intel Core i7-4720HQ processor, 16Gb RAM), equipped with an NVIDIA GEFORCE GTX 960m GPU. The software implementation for learning uses Docker version 17.03.1-ce hosted on Ubuntu 16.04 LTS. On top of this, the solution uses the NVIDIA-DOCKER software solution enabling the containers to access the GPU of the host computer. The container used for training in the project is a modified version of the official Python 3 and GPU enabled Tensorflow Docker image ("tensorflow/tensorflow:latest-gpu-py3"), available at the Docker Hub. While the default base image supports GPU and python 3, additional layers were built into the container in order to implement the OpenCV 3 dependency for the project. For driving, the project uses the official

Udacity Docker image with the exception of an update of Keras to version 2.0.3 which is the same used by the training container.

1. Required files and quality of code

All required files are uploaded to the Udacity project repository for review and evaluation. The code is annotated to help on its interpretation. Most of the engineering process of writing and testing the `model.py` code has been carried out in a Jupyter notebook decomposed in cells structured according to the main functionalities of the program as described below.

1.1 `model.py`

As expected, the code uses a generator to prevent excessive usage of computer memory. Testing without the generator rendered unusable the development laptop using in the project. Key parts of the code are:

Initialisation: imports the libraries and components necessary to perform the training.

Program settings: collects the variables and settings related to the file management (images and csv files), and the key settings of the neural network. As it is setup, the `model.py` file has to be in a directory containing another directory called “data”. The latter contains the csv file provided by Udacity and the IMG directory containing all the images of the dataset.

Supporting functions: the `preprocess_img` function is used to transform any single images coming from the simulator, (shape 160x320x3 RGB color schema), so they can be utilised as input to the neural network, (shape 66x200x3 YUV color schema). The function `prepare_training_data` is the generator used in the project to “yield” images to the neural network in batches defined in the parameter `batch_size` in order to manage efficiently the memory used by the system.

Main program. This part of the code executes the pipeline transforming the images and labels in the computer file system into a preprocessed dataset ready to be fed into the neural network. First, the program opens the csv file and loads the dataframe into a list, (`driving_data`). An iteration through this list removes rows of the dataframe in which the captured speed is near zero. As recommended by [1] and others on the Udacity forum, this iteration includes adjustments on the angle reading of left and right hand images by +0.25 and -0.25. The result is the two arrays with the same amount of elements: `images[]`, `angles[]`.

The second element of the main program is the pre-processing of these arrays in order to analyse and improve the distribution of angles in the dataset. The recording of steering angles in driving activity usually shows a high frequency distribution of 0 degree angles of the steering wheel. As recommended in [2] the adjustment of this distribution on a training dataset is important to remove the bias towards driving straight in the end result. The resampling of the dataset uses the approach from [1]. In Illustration 1 (a) and (c), the distribution of angles captured in the dataset provided by Udacity and in an enriched dataset created manually using the simulator.

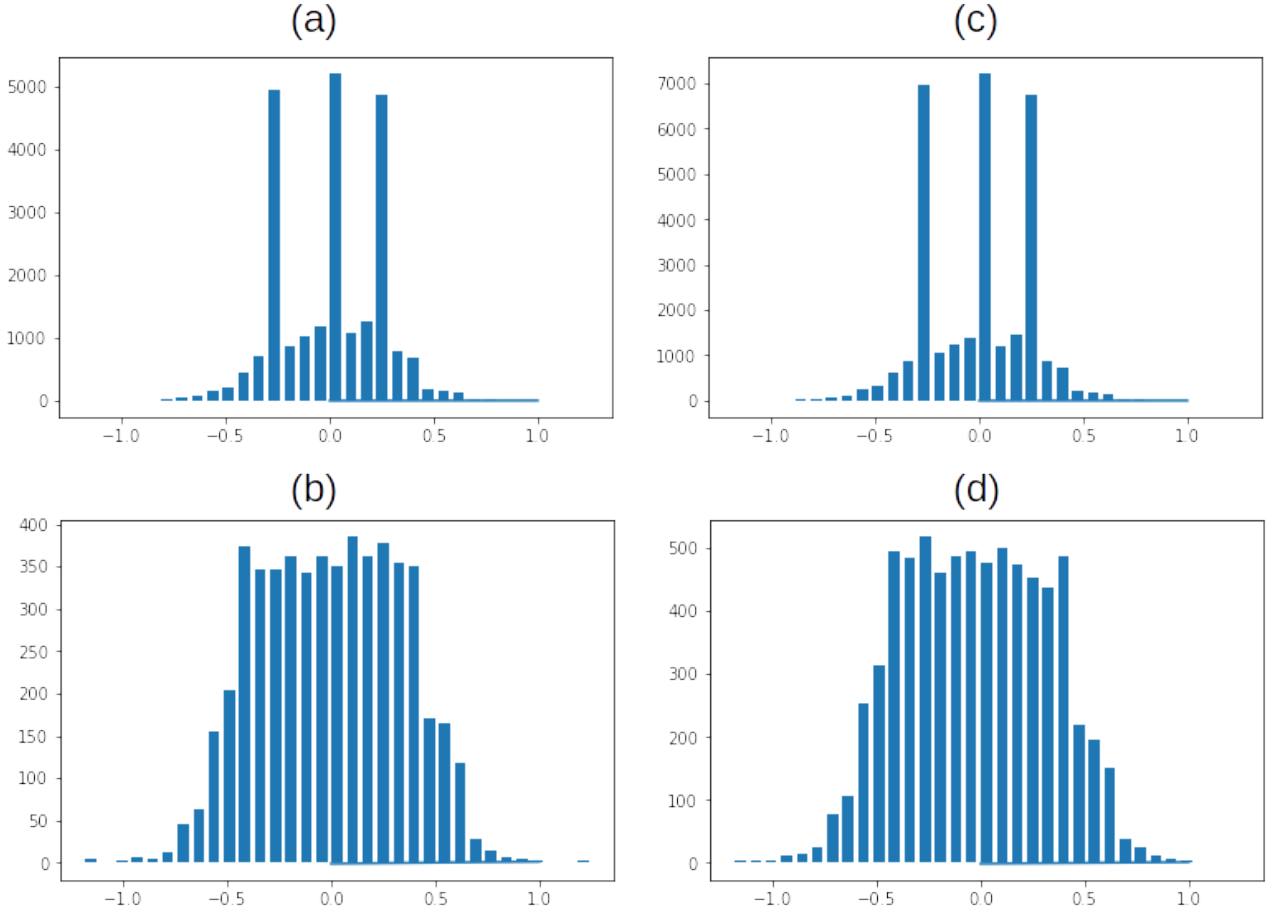


Illustration 1: Distribution of steering angles in the datasets of labelled images: (a) Udacity input dataset, (24105 samples). (b) Udacity postprocessed dataset, (5124 samples). (c) enriched input dataset, (31812 samples). (d) enriched and post processed dataset, (7142 samples)

The algorithm establishes a number of bins for the distribution (n_bins) and calculates the average amount of samples in the bins ($avg_samples_per_bin$). Based on the deviation from the average samples per bin a multiplication factor to correct the number of samples in each bin. The threshold to establish if bin is oversampled is half of the $avg_samples_per_bin$. For example in the (a) distribution of Illustration 1 the resulting array of shown below:

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.83006198347107452,
0.51440460947503197, 0.073590020698624373, 0.41740259740259739, 0.35666725852272724,
0.30820866896816268, 0.070141592611344866, 0.33911538786190598, 0.29055471179576192,
0.074810994003947723, 0.46172853694977595, 0.53317850033178504, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

As it can be seen, in the transformation of the dataset from (a, c) to (b, d), bins with low amount of samples (far away from zero) have 1.0 factors, whereas highly populated bins will be modified by multiplication factors closer to zero. The `remove_list = []` iteration array is generated by an

iteration through the samples in the bin which randomly removes samples until the bin complies with the `0.5*avg_samples_per_bin` constraint. Examples of the resulting distribution are shown in Illustration 1.

The final part of the `model.py` program is the generation of the training and testing sets using the `sklearn` function.

This section of the code has commented pieces that can be used to plot the distribution histograms before and after preprocessing the dataset.

Convolutional neural network. The last part of uses the Keras 2.0.3 library implement an adaptation of the NVIDIA neural network architecture reported in [2]. This network takes the inputs preprocessed in earlier parts of the code. The implementation uses Keras' `ModelCheckpoint` function to save the model generated in each epoch of the training. It also saves the model and the weights into `h5` and `json` files readable by the `drive.py` program.

1.2 `drive.py`

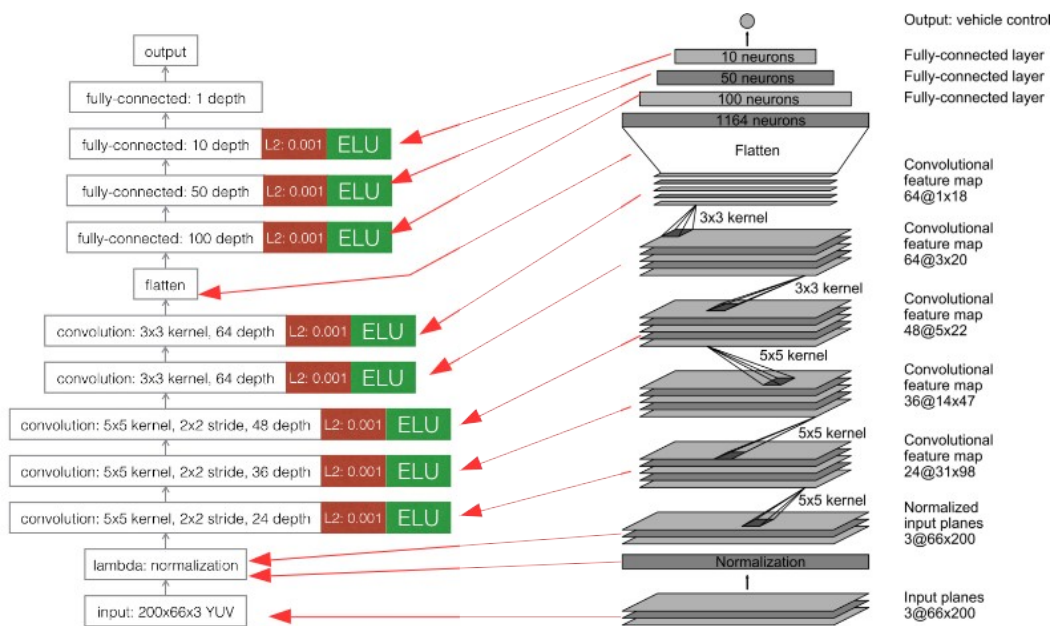
The original `drive.py` program provided by Udacity has been modified to add the `preprocess_img` function described in section 1.1 and used in `model.py`. The only difference between the two is the input image that these functions take in their respective contexts. In `model.py` the function takes images using the BGR color schema utilised by the OpenCV reader object. In the `drive.py` program, the images come specified from the Unity3D simulator using the RGB color space. In both contexts the output images are specified in the YUV color space which is the recommendation by NVIDIA to feed the neural network architecture used in the project [2]. Small modification is also introduced in the `drive.py` code to read the trained model weights from a `json` file.

2. Model Architecture and training strategy

This section answers the questions of the project rubric regarding model architecture and training strategy.

2.1 Appropriate model architecture

I chose for the project the architecture reported in [1] which is an adaptation from [2] as shown in Illustration 2:



Shannon's Adaptation [1]

NVIDIA architecture [2]

Illustration 2: Network architecture compared to NVIDIA model

The architecture recommended by Udacity for this project since it is a well known architecture for the same problem of steering a car in a simulation [2]. All the activations in the architecture use the ELU function. This seems to be commonly recommended by Udacity forum mentors like ayush_jain when students ask for help. The model uses the Keras lambda function to normalise the data.

2.2 Overfitting reduction

The project uses Keras additional regularization hyperparameter to help manage overfitting. This is oposed the other main technique explained in the course linked to stop learning when the loos starts to decrease.

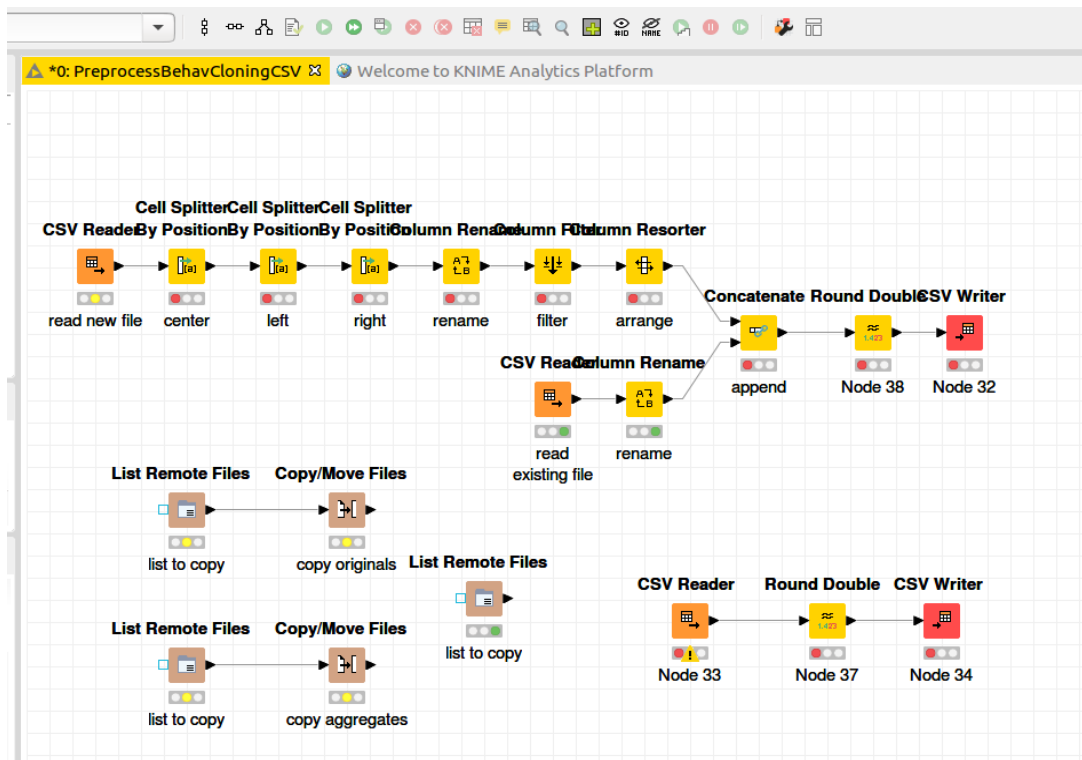
2.3 Model parameters tuning

My general approach to learning rate so far (looking at previous execise) is to keep it low. The model uses Adam optimiser and I just realised that the use of Adam removes the need to adjust the learning rate. The network uses MSE as the indicator of loss. However, is well reported in the forums and in the course material that MSE does not necessarily corrlate to performance in this project.

2.4 Training data

Up to four captures of data driving the car in training mode were performed in the project. 2 of these data collection exercises focused on driving straight and the remaining two focused on recovery. Recovery was trained by taking driving diagonally in the road until the car is almost out

of it and then recover by driving back. For efficiently processing the datasets coming from the simulator without making any changes to the code, I used an external data processing tool called Knime. With Knime I could process the local paths of the output `driving_log.csv` file coming from the simulation and adjust its content to my local images and csv file storage paths. With this method it did not matter what data I obtained, I could automate the renaming of the csv files and the fast copy of the images to a desired local directory so the program always points to an images path starting with `IMG/...` as it is coded. I found this method efficient due to my limited expertise in python. However, others may be more confident to write a helper function to do the same.



Drawing 1: screenshot of Knime

3. Achitecture and training documentation

Using Keras 2 specified the execution of the model in the following way:

```
model.fit_generator(train_prep,
                    validation_data=val_prep,
                    epochs= n_epochs,
                    validation_steps=(len(images_test)/batch_size),
                    steps_per_epoch=(len(images_train)/batch_size),
                    verbose = 1,
                    callbacks = [checkpoint])
```

After several attempts to understand the and specify a correct value for `validation_steps` and `steps_per_epoch`, I found help in the Udacity forum explaining that the recommended approach is to

use the size of the training set divided by the batch size. Until then, I was choosing different values and not being able to drive the car within the road limits. The specification of these values as described in the above code snippet is what made me achieve the first complete lap of the project.

Following this, I run a series of experiments with the network architecture that had worked with various conditions of the number of epochs and datasets. I saved the output models to test their performance on the road. The following table summarizes these tests:

scenario	Dataset	Epochs	Driving assesment
1	Udacity_provided dataset	20	Consistent drive around the circuit
2	Udacity_provided dataset	30	Consistent drive around the circuit
3	Udacity_provided dataset	50	Undecisive driving and crash
4	Udacity_provided + manually generated dataset	20	Undecisive driving and oversteering (wobbly drive)
5	Udacity_provided + manually generated dataset	50	Undecisive driving and oversteering (wobbly drive)


The best results I managed to obtain are the ones of scenario 1 and 2. Training time in both cases is under three minutes on my laptop. I interpret the oversteering as a case of overfitting.

4. Conclusions and discussion

A convolutional neural network to drive a simulated car has been developed and reported in this document. The architecture choices and the training conditions of the scenarios 1 and 2 enabled the car to consistently drive around the simulation track. None of these completed the challenge track but covered about a third of its length before crashing on a very steep curve. The objective of my implementation as stated in section 0 was to test whether if the chosen architecture could be able to drive the track with minimum transformation of the images and no jittering. The results shows the achievement of this objective by demonstrating that the care can be driven around the track with a low amount of training and data, (5124 angle labelled images in the post processed training dataset). This corroborates the conclusions made by Shannon [1], who pointed out of the importance of the data. The work here extends those insights by testing their validity on an even simpler setup in terms of data and training. Although [1] does not specify the epochs used for training, his work reports the use of a larger dataset of 16482 labelled samples, (post processed). This findings have implications in the strategy to approach a deep neural network implementation: give high importance to the quality of the data.

The results described here also challenge some of the recommendations 4 and 6 from Paul Heraty (Illustration 1). However, it can be concluded that the guide is still accurate in most of its statements.

Behavioral Cloning Cheatsheet

 Paul Heraty • Dec 08, 2016

112 There are about 10 things that I wish I knew before I started this exercise.

Note: These points below are not the 'only' way of solving this problem. Think of them as pointers and feel free to pick and choose as you see fit.

1. How to use Python generators in Keras. This was critical as I was running out of memory on my laptop just trying to read in all the image data. Using generators allows me to only read in what I need at any point in time. Very useful.
2. Use a GPU. This should almost be a prerequisite. It is too frustrating waiting for hours for results on CPU. I must have run training 100 times over the past 3 weeks and it was driving me crazy. Using a GTX980M was around 20x faster in training than a quad-core Haswell CPU.
3. Use an analog joystick. This also should be a prerequisite. I'm not sure if its even possible to train with keyboard input. I think some have managed it, but for me it's a case of garbage in, garbage out.
4. Use successive refinement of a 'good' model. This really saves time and ensures that you converge on a solution faster. So when you get a model working a little bit, say passing the first corner, then use that model as a starting point for your next training session (kinda like Transfer Learning). Generate some new IMG data, lower the learning rate, and 'fine tune' this model.
5. Use the 50Hz simulator. This generates much smoother driving angle data. Should be the default. You can find a link to download this on the Slack channel. Choose the fastest graphic quality and lowest screen resolution has helped the model to perform better.
6. You need at least 40k samples to get a useful initial model. Anything less was not producing anything good for me.
7. Copy the Nvidia pipeline. It works :) And it's not too complex.
8. Re-size the input image. I was able to size the image down by 2, reducing the number of pixels by 4. This really helped speed up model training and did not seem to impact the accuracy.
9. I made use of the left and right camera angles also, where I modified the steering angles slightly in these cases. This helped up the number of test cases, and these help cases where the car is off center and teaches it to steer back to the middle.
10. Around 5 epochs seems to be enough training. Any more does not reduce the mse much if at all.
11. When you're starting out, pick three images from the .csv file, one with negative steering, one with straight, and one with right steering. Train a model with just those three images and see if you can get it to predict them correctly. This will tell you that your model is good and your turn-around time will be very quick. Then you can start adding more training data.

Hopefully the above will help you converge quicker. It can be a pretty frustrating project at the beginning, but worth it when you have it working!

Most of these ideas have come from the great folks on the Slack channel, which reminds me:

12. Use the Slack channel. The people on there are awesome!

Illustration 3: Paul Heraty's recommendations for Project 3.

In summary, I think that I have achieved my learning objectives with this project. There are still some uncertainties about the tweaking of paramters. I leave those for future projects.

References

- [1] <https://github.com/jeremy-shannon/CarND-Behavioral-Cloning-Project>
- [2] M. Bojarski et al. End to end learning for self-driving cars. <https://arxiv.org/pdf/1604.07316.pdf>