Writeup | Behavioral Cloning

5 min read

Abstract — this notebook is the writeup of the Behavioral Cloning project as part of the SELF-DRIVING CAR nanodegree program. We apply Deep Learning to clone the behavior of a human driver by getting training data from examples of human driving in a simulator. Then these data feed into a convolutional neural network (CNN) to map features from camera images directly to steering commands. This way the CNN learns to predict the appropriate steering angle when the car drives in autonomous mode in the simulator.

The goals of this project are broken down into the following steps:

- 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

Note:
- the architecture has been inspired by the nvidia neural network (https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf) before being tweaked a bit using the Keras (https://keras.io/) deep learning library with Tensorflow backend. The convolutional neural network has been trained only with the sample data provided by Udacity.
- the code was written on Windows environment. Edit the model.py file for a description in details of the environment.

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.

1. Files Submitted & Code Quality

1.1. 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
- tools.py containing the generator to pull pieces of the data and process them on the fly
- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- writeup_report.pdf summarizing the results
- video.mp4 is a video recording of the vehicle driving autonomously at least one lap around the track one

1.2. Submission includes functional code

Using the Udacity provided simulator and a modified version of the drive.py file, the car can be driven autonomously around the track by executing

```
python drive.py model.h5
```

1.3. Submission code is usable and readable

The model.py and tools.py files contains the code for training, fine tuning and saving the convolution neural network.

The dir_create() function, in model.py, will create the following directory tree:

fig.0.1: *directory tree by dir_create() ,* ***fig.0.2:*** *put the sample folder into the data directory*

```
root (fig.0.1)
|    drive.py
|    model.py
|    tools.py
|
\---data
|    \---h5
\---logs
     \---nn_logs
```

```
root (fig.0.2)
|    drive.py
|    model.py
|    tools.py
|
\---data
|    \---h5
|    +---sample
|         |    driving_log.csv
|         |
|         +---IMG
|                center...jpg
|                left...jpg
|                right...jpg
|                ...
|
\---logs
     \---nn_logs
```

Put the sample folder into the data directory.

The files show the pipeline I used for training and validating the model, and they contain comments to explain how the code works. For more information, execute the following commande:

```
python model.py --help
```

2. Model Architecture

2.1. An appropriate model architecture has been employed

There are the following layers : a normalization layer, 3 convolutional2D layers, 1 flatten layer and 4 fully-connected layers. ELU (Exponential linear unit (https://www.quora.com/How-does-ELU-activation-function-help-convergence-and-whats-its-advantages-over-ReLU-or-sigmoid-or-tanh-function)) activation functions are added at each convolutional2D and fully-connected layers. The dropout regularization technique is inserted between the last convolutional2D layer and the flatten layer.

The final model consisted of the following layers:

Layer	Description
input	32 x 155 x 3
normalization	lambda x: x/255.0-0.5
convolution2D	{ filter: 16 ; stride: 2x2 ; kernel size : 5x5 }
activation function	ELU (Exponential linear unit)
convolution2D	{ filter: 32 ; stride: 2x2 ; kernel size : 5x5 }
activation function	ELU
convolution2D	{ filter: 64 ; stride: 2x2 ; kernel size : 5x5 }
activation function	ELU
regularization	dropout(keep_prob = 0.5)
flatten	
fully connected (Dense)	{ 100 hidden units }
activation function	ELU
fully connected (Dense)	{ 50 hidden units }
activation function	ELU
fully connected (Dense)	{ 10 hidden units }
activation function	ELU
fully connected (Dense)	{ 1 hidden unit }

Note: *I do not use a max pooling layer. For more details, read the Geoffrey Hinton's comments on max pooling (https://mirror2image.wordpress.com/2014/11/11/geoffrey-hinton-on-max-pooling-reddit-ama/)*

2.2. Attempts to reduce overfitting in the model

The model contains one dropout layer in order to reduce overfitting.

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 44-50). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

2.3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 70) except for the transfer learning phase where I changed the default learning rate from 1e-3 to 8.5e-4.

2.4. Appropriate training data

I tried to collect data using the keyboard arrows but it was very challenging and at the end the data were not as good as expected to train properly the neural network. Then the sample data provided by Udacity have been used to train the model. Jitteration and preprocessing techniques have shown to be very effective ways to improve the neural network performances as I have learnt in the Traffic Sign Recognition project. In order to keep the vehicle driving on the road the data have been also preprocessed and jittered before fine tunning the neural network.

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

3. Training Strategy

3.1. Solution Design Approach

In summary, it was an iterative approach with much trial and error. Initially, I have trained a convolution neural network (CNN) model similar to the Nvidia CNN with the original sample data set. Next, I modified the CNN model and tweaked a few parameters. Then I preprocessed and artificially augmented data to fine tune the model using the transfer learning technique.

3.1.1. The first attempt: Nvidia convolution neural network

Initially I used a convolution neural network model similar to the Nvidia CNN as discribed below:

fig.1: first adaptation of the Nvidia CNN

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 64, 64, 3)	0
conv2d_1 (Conv2D)	(None, 30, 30, 24)	1824
conv2d_2 (Conv2D)	(None, 13, 13, 36)	21636
conv2d_3 (Conv2D)	(None, 5, 5, 48)	43248
conv2d_4 (Conv2D)	(None, 3, 3, 64)	27712
conv2d_5 (Conv2D)	(None, 1, 1, 64)	36928
dropout_1 (Dropout)	(None, 1, 1, 64)	0
flatten_1 (Flatten)	(None, 64)	0
dense_1 (Dense)	(None, 100)	6500
dense_2 (Dense)	(None, 50)	5050
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11

Total params: 143,419
Trainable params: 143,419
Non-trainable params: 0

The Nvidia model might be appropriate because it learns to drive in track with or without lane markings or in areas with unclear visual guidance such as on unpaved roads. It automatically learns to detecte useful road features with only the steering angle as the training signal.

At this stage, the vehicle could stay on the track one when the model was tested by running it through the simulator. **However, the car tends to drive on the left lane line as it is shown in the video.1.**

video.1 - result 1

(https://youtu.be/uxNCMQ3gk-Q)

3.1.2. Overfitting and countermeasure

In order to gauge how well the model was working, I split the image and steering angle data into a training and validation set. This first model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting.

To combat the overfitting, I used the following techniques:

- train the model with more data (augmentation and jitteration)
- stop the training when the loss has stopped improving
- add a dropout layer as a regularization technique

I also removed all images and the related steering angles driving the car off the track from the sample dataset.

If I would have more time, I would implement the cross-validation technique (https://github.com/keras-team/keras/issues/1711) as well. But the vehicle could stay on the track one when the model was tested by running it through the simulator:

3.1.3. Fine tuning

The video.1 shows the vehicle drove on the lane line of the track one. To improve the driving behavior, I first modified the convolutional neural network by removing a few convolutional2D layers and then I tweaked a few parameters like the initial learning rate in the transfer learning phase, the number of convolution layers and their related parameters, the regulation functions, to name but a few.

From that point, I have fine tuned the model twice using the transfer learning technique: first with more training data, and then with a different compensation rate of the steering angle. For details about how I created the training data, see the next section.

At first sight, the video.2 shows the fine tuning with more training data improved dramatically the car behavior by driving much less on the lane lines, more in the center of the track from the beginning, then over the brige, passing successfully the first turn with its dirty border but **it drove off the track in the second turn**:

video.2 - result 2

(https://youtu.be/_FEcHhhy9lc)

In order to overcome the case of the vehicle leaving the road, I have fine tuned a second time the model by increasing the compensation rate of the steering angle. Finally, the video.3 shows the vehicle can drive autonomously around the track one without leaving the road:

video.3 - result 3

(https://youtu.be/McrmB3ZIA18)

3.2. Creation of the Training Set & Training Process

There are a few challenges. The car has to drive in the center of the track from the beginning, then over the brige, passing successfully a few sharp turns and some with dirty border instead of clean lane lines, not to mention a few cast shadows on the track.

Despite this, I chose to train the model with the sample data provided by Udacity without collecting additional data.

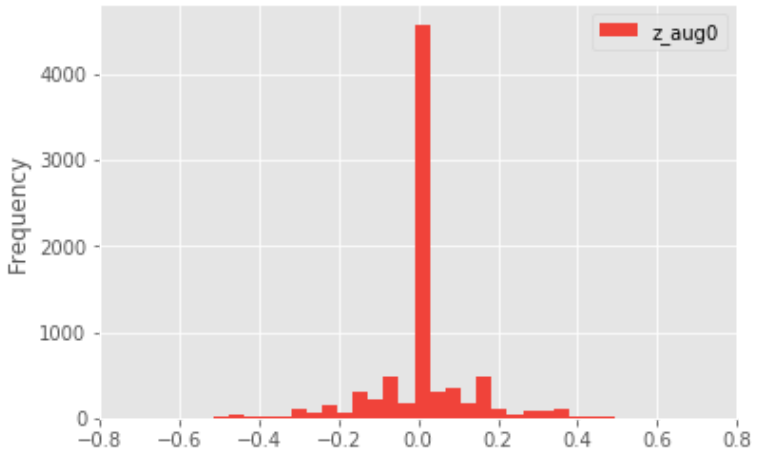
3.2.1. Data exploration

Initially, the sample data is comprised of 8036 units. Each unit is made of a steering angle value and the associated images from the center, left and right cameras. Here is a video based on all images from the center camera found in the sample data set cleaned of a few units that drove the car off the track: video.4 (https://youtu.be/YzftbxF7-_w)

The throttle, the brake and the speed values were set aside in this project. I will use them at a later date when I have more time.

To begin with, the figure 2 reveals the data is imbalanced:

fig.2 - number of occurence per steering angle value



As we can observe, there is a concentration of steering angles of 0 radian. The track one is straight driving and most of the time with light steering and a few sharpe turns. In this case, the value of mean_squared_error metric might be excellent on paper but it is only reflecting the underlying steering angle distribution. Consequently, the model will have the tendency to decide that the best thing to do is to often predict a steering angle of 0 degree. In other words, the sample data has too much straight driving data with few turning data. It explains why the car went off the track at the first turns when the model is trained with the original sample data *(see video.1)*.

In order to remedy this problem, I have learnt in the previous project that the best way is to rebalance the data. For those purposes, there are two options:

- either increase right/left turning data
- or decrease the straight driving data

I chose to increase right/left turning data to both rebalance the data set and to reduce overfitting in the model.

3.2.2. Rebalance the data set - increasing right/left turning data

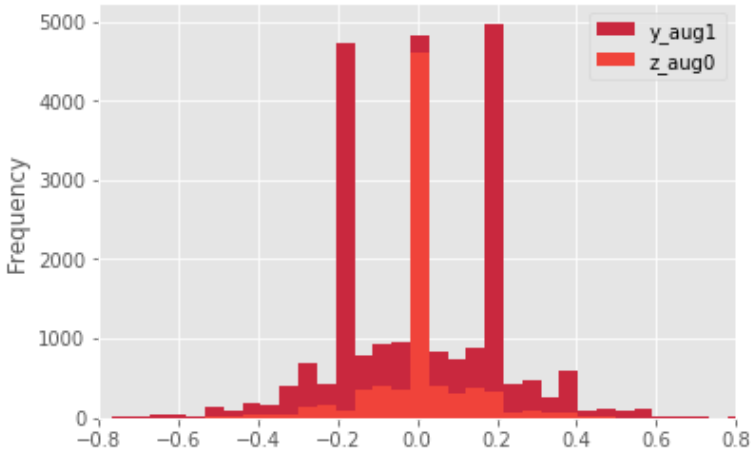
For a start, I have at my disposal images from the left and rigth cameras that I can use to create some artificial 'recovery' data, i.e. data that enable the model to learn how to recover when the vehicle drives away from the center of the track. For example, if we look at the images from the left camera in the figure 3, the vehicle seems to be closer to the left lane than it actually is.

fig.3 - example of images from the center, the left and rigth cameras

If it was an image from the center camera, we would like to teach the model to steer the vehicle back on the center of the track. To increase the left turning data, I used the left image as an input of the model and I added a readjustment angle to the current steering angle value as the desired output value. I repeated it with the right images but this time subtracting the readjustment angle to the steering angle.

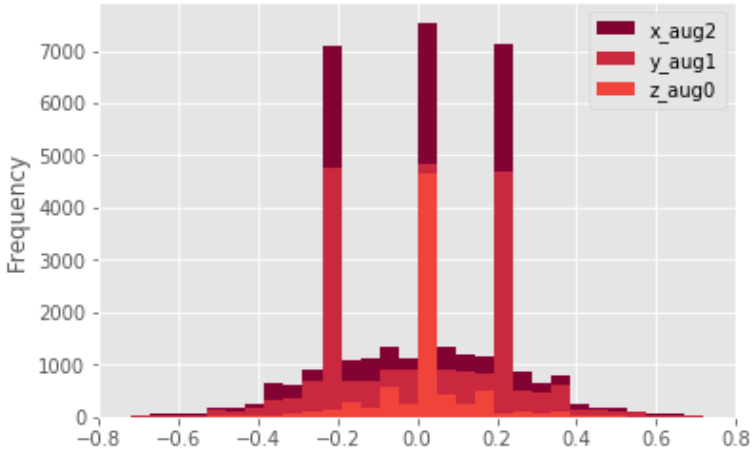
The figure 4 shows the steering angle distribution after using the left and right images:

fig.4 - rebalance - part1 / number of occurence per steering angle value



Secondly, I randomly flipped images and angles in order to augment the data set and rebalance right and left images. At the end of this increase of data, I had **68,430 number of data points**. The figure 5 shows the new steering angle distribution:

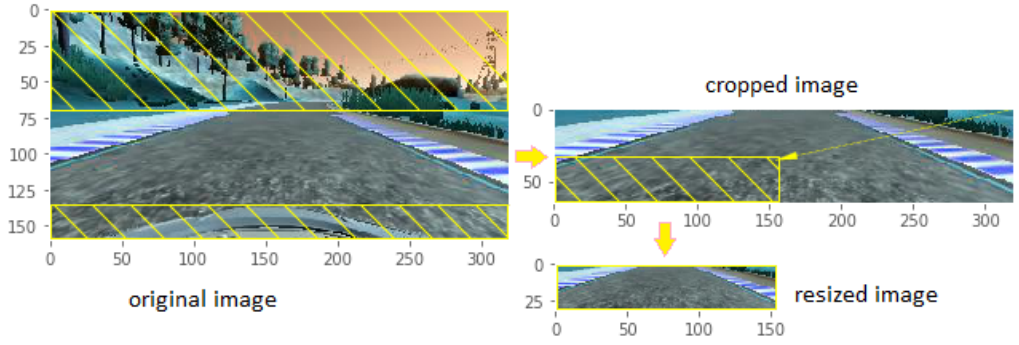
fig.5 - rebalance - part2 / number of occurence per steering angle value



3.2.3. Preprocess the images - cropping, resizing, changing brightness

Next, I preprocessed this data by cropping and resizing images.

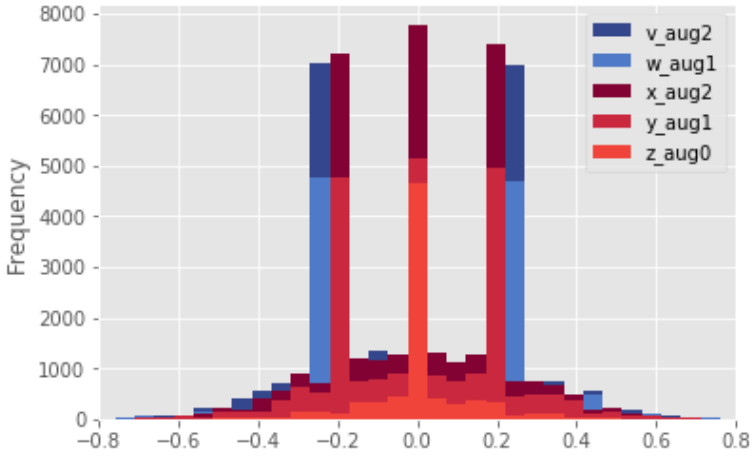
fig.6 - cropped and resized images



Then, I randomly shuffled the data set and put 20% of the data into a validation set. And I used this augmented data for training the model. The training is interrupted when the validation loss isn't decreasing anymore by using an EarlyStopping callback. And if by chance it does not interrupt itself, the maximun number of epochs has been set between 5 and 10. I also used an adam optimizer so that manually training the learning rate wasn't necessary. As a result, the car drove off the track in the second turn after passing the first bend, the one with the dirty border *(see video.2)*.

In order to address this shortcoming, I fine tuned the model using a greater reajustment angle value. I have changed the 0.20 radian by 0.25 radian of reajustment. At the end, I had **128,604 number of data points in total** *(see figure 6)*. And it solved the problem. The vehicle could finally drive the whole track one *(see video.3)*.

fig.6 - fine tune - number of occurence per steering angle value



4. Does the model generalize well?

4.1. Track two

There are additional challenges. Unlike the track one, the track two is not flat at all. There are several steep inclines and much more sharp turns. What is more, there are a few short or dark segments of the road. Not to mention, there are much more tremendous cast shadows on the track.

The model does not generalize well. The video.4 shows the vehicle drove off having a tendency to get around the cast shadows on the track.

video.4 - result 4

(https://youtu.be/4f8VXsyo5E0)

4.2. What's next?

If I would have enough time, I would fine tune the model once I would have implemented the following ideas to get around these challenges:

- dark segment: *(done)* change randmly the brightness of images to simulate day and night conditions
- steep incline: *(to be done)* vertical translation of images and compensate for steering angles accordingly
- sharp turn: *(to be done)* both the same technique as for the track one and collect more specific data

---

ANNEXE

What I have learnt in this project:

- keras
- argparse
- pandas
- the top-level script environment: main()
- git/github workflow: master, develop, release, hotfixes, feature branches