Write-up Report of Project 3

Deep Learning for Driving Behavior Cloning

jiyang_wang@yahoo.com

1. The goals / steps of this project

- 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

2. Rubric Points

Files Submitted & Code Quality

My project includes the following files:

- **model.py** containing the script to create and train the model
- **drive1.py** for driving the car on track1 and **drive2.py** on track2, in autonomous mode
- Two convolution neural network models, **model1.h5** and **model2.h5**, each trained for track 1 and track 2 respectively, used together with drive1.py and drive2.py as their respective controller.
- Two mp4 video files **runtrack1.mp4** and **runtrack2.mp4** (a video recording of your vehicle driving autonomously around the track for at least one full lap)
- **writeup_report.pdf** summarizing the results
- I also included Jupiter Notebook version of model.py ("P3 Behavior Cloning Training - Good for Track1.ipynb")

Submission includes functional code Using the Udacity provided simulator and drive1.py/drive2.py file, the car can be driven autonomously around track 1 by executing

```
python drive1.py model1.h5
```

and around track 2 by executing

```
python drive2.py model2.h5
```

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

My initial model was a scaled-down version of PilotNet (“*Explaining How a Deep Neural Network Trained with End-to-End Learning*”, by Mariusz Bojarski, et al) shown in Figure 1.

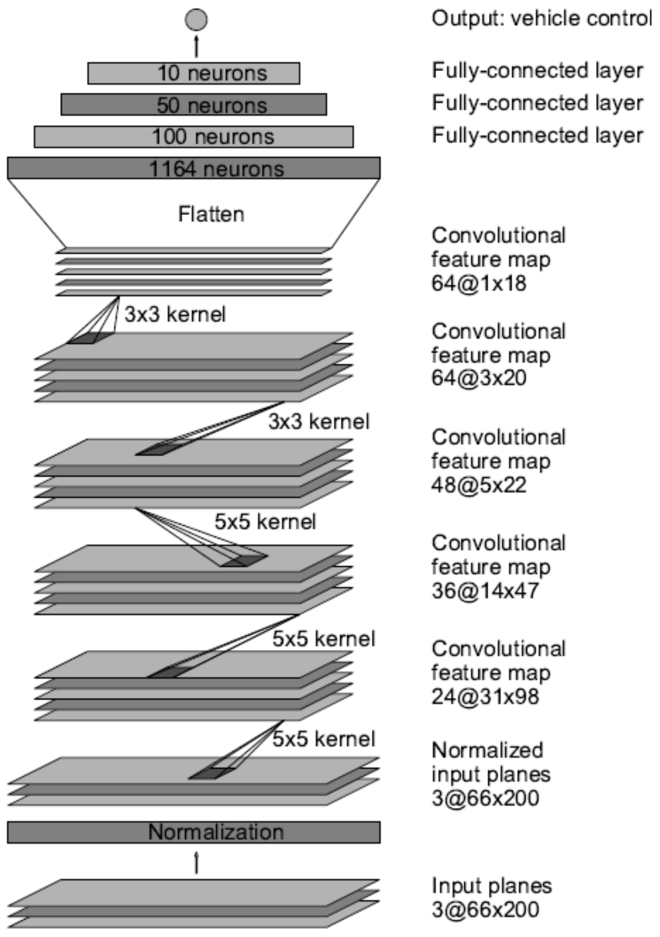


Figure 1. PilotNet Architecture

I made two changes to PilotNet:

1. The 2nd convolution layer (36@14x47) is removed
2. The 1st full-connection (FC) layer has 200 instead of 1164 neurons

All the rest of my initial model is the same as PilotNet.

The reason for the two changes is that I want to reduce the training time while still keep the model sophisticated enough to capture all the features required for steering the car on track, for which I kept two 5x5 and two 3x3 convolution layers. Meanwhile I reduced the number of neurons in the 1st FC layer to 200, which I believe is enough for non-linear feature mapping for this project (the paper referenced above does not explain why PilotNet chose the particular number 1164, even though empirically more neurons render higher dimensions to the model that can search more easily the optimums within relatively lower dimensions of the problem, i.e., finding the steering angles based on the images).

The convolution layers

The input image has the size of 160x320x3, i.e., it has 3 channels, and then each channel is **normalized** by a lambda layer (model.py line 225) and then filtered by a 5x5 kernel (model.py line 228), which is exactly the same as PilotNet, followed by more convolution layers of 5x5 and 3x3 kernels. Each convolution layer uses RELU as its activation function to introduce **non-linearity**.

A **cropping** layer (model.py line 226) is inserted right behind the normalization layer to crop the input image so that the trees, hills, lakes, etc., that are not relevant to the track are cropped off from the input, avoiding the confusion to the model and reducing the training time (because the dimension of the input vector is reduced).

After each convolution layer, there is an **average-pooling** layer. At the beginning I used **max-pooling** but later on I found that average-pooling performed better when running the car on track 1 by the model.

Model parameters

Between the full-connection layers there are **dropout** layers to reduce overfitting. The dropout rate is picked differently for track 1 and for track 2 and it is somehow related to the number of epochs because high number of epochs may also lead to overfitting. The values of these hyper-parameters are tuned according to the test results on track. The dropout rate is 0.5, 0.3 and 0.1 for the 3 FC layers in model1 for track 1 (line 244, 247, 250 in model.py), while it is set to 0.3, 0.2 and 0.1 in model2 for track 2.

The model used **Adam** optimizer, so the learning rate was not tuned manually, and it used **MSE** loss function.

The final model

I used the initial network described above and eventually trained the model to successfully drive the car on track 1. But I found that it took much longer to train the model for the 2nd track (2nd track is much longer than the 1st track and thus generates much more input images). So I further simplified the model by:

1. Removing the 2nd FC layer with 100 neurons, leaving only 3 FC layers (with 200, 50, 10 neurons, respectively)

2. Converting the input images to **YUV** and leaving only Y channel in the inputs. So the input images now have only one channel.

The Y channel provides enough detailed information to the model to extract the features in the images. The U and V channel have some kind of blurred images that are not very useful.

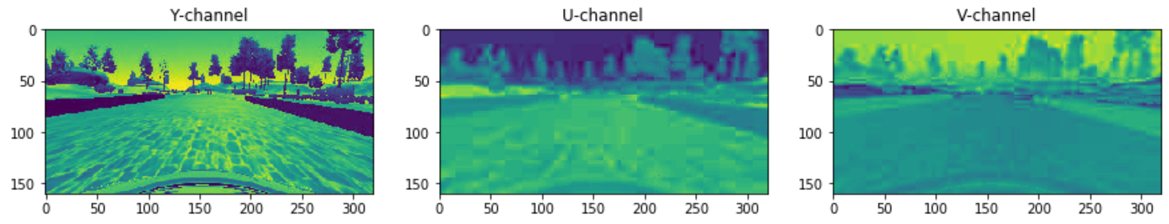


Figure 2. Image in YUV color space

This new model is much smaller than PilotNet, though. Is it still possible for it to be trained so that it is able to drive the car on both track 1 and track 2? I believe so but also know that the quality and quantity of dataset as well as the training strategy are crucial to achieving the goal. Here is the final model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_1 (Lambda)            (None, 160, 320, 1)       0
_________________________________________________________________
cropping2d_1 (Cropping2D)    (None, 100, 320, 1)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 96, 316, 24)       624
_________________________________________________________________
average_pooling2d_1 (Average (None, 48, 158, 24)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 44, 154, 48)       28848
_________________________________________________________________
average_pooling2d_2 (Average (None, 22, 77, 48)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 20, 75, 64)        27712
_________________________________________________________________
average_pooling2d_3 (Average (None, 10, 37, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 23680)             0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               4736200
_________________________________________________________________
dropout_1 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                10050
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                510
_________________________________________________________________
dropout_3 (Dropout)          (None, 10)                0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 11
=================================================================
Total params: 4,803,955
Trainable params: 4,803,955
Non-trainable params: 0
```

The Dataset and Its Split

The Udacity simulator provides a very efficient way to build up the dataset. Captured images and their associated steering angles are automatically stored after running the car in the simulator as many laps as you want, and are easy to access.

Every primary dataset I use for **training** and **validation** is the result of running the car for at least two laps, plus accumulated recovery driving images. It is then split into training set and validation set by the ratio of 80:20 (model.py line 215).

The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

Dataset Augmentation

There may be three approaches to augment the dataset, which are implemented in model.py:

1. Adding the images captured by the left and the right camera with the steering angles adjusted from the angles by the central camera (model.py line 77 – 100).

2. Flipping the images with the angles reversed (model.py line 102 – 113)

3. Translating the images (model.py line 117 – 133), shifting the images, e.g., in random range (-20, 20) pixels in x and y directions.

However, I didn't use them in the actual training because it is just so easy to get enough examples from running the car in "Training" mode in the simulator.

Data Generator

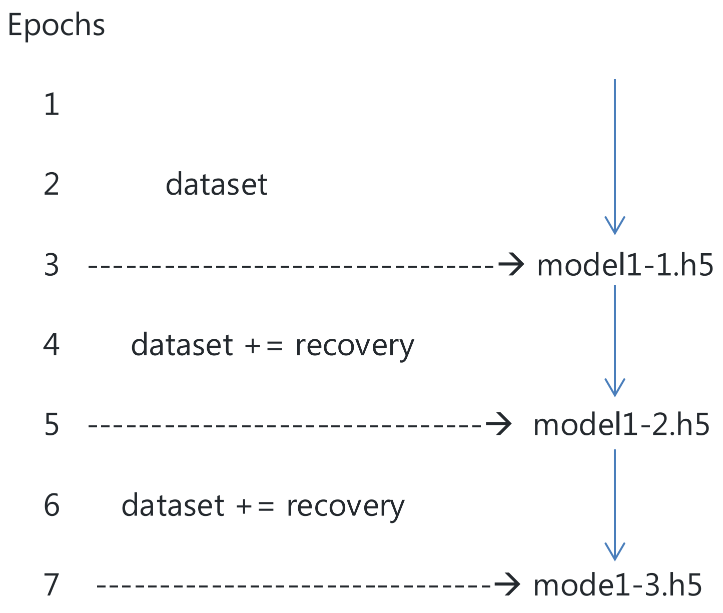
Using Keras generator in model training is extremely important to me because I'm using a modest laptop PC to run the programs. Generator avoids loading the whole dataset to main memory but instead it loads just a batch of data during the epochs of one training round. Both training dataset generator and validation dataset generator are provided (model.py line 158 – 209).

Training Strategy

As mentioned earlier, training strategy is crucial to a small model to successfully reach the point where the model can drive the car in simulator without getting out of the tracks or hitting any obstacles.

My training strategy is to set the training to two stages. The first stage trains the model with a primary dataset for a few epochs, say, 3 or 4. Then I'll see how it goes in the simulator, and jot down the places where the car fails to keep on track (which I call "foul lines"). In the 2nd phase, I collect more normal driving and recovery driving data around the foul lines and add the data to the existing primary dataset used in the 1st phase. Then I use the accumulated data to train the model again (not from scratch). I code the program in the way that makes this kind of "**accumulative training**" very easy (model.py line 272 – 301), following the **transfer learning** paradigm. So the model is improved continuously until it runs through the whole track without hitting the foul lines.

It is important to note that the number of epochs should not be large in the 1st phase. For example, if overfitting is expected to take place at epochs = 8, then it is necessary not to set epochs higher than 4 in the 1st phase, leaving another 3 or 4 to the 2nd phase. This can be illustrated as below.


Epochs
1
2 dataset
3 --------------------------------→ model1-1.h5
4 dataset += recovery
5 --------------------------------→ model1-2.h5
6 dataset += recovery
7 --------------------------------→ mode1-3.h5

If at this moment the model is still not good enough, it is time to continue the training by new dataset (probably just some recovery driving data) for a couple of epochs.

This training strategy proves to be efficient. I've got the models this way that run the car in simulator all along track 1 and track 2.

The controller drive.py

Initially I didn't intend to do anything to drive.py. But converting the input images to YUV and leaving only Y channel in the dataset asks me to modify drive.py because now the model makes prediction of steering angle based on Y-channel images (before it is based on RGB images). This change is made by adding line 78 to 80 to telemetry() in drive.py.

This is OK for track 1. But when I saw that the car on track 2 may sometimes stop on a descending slope or may seem not to have enough power to climb an ascending slope, and that the car made the turn too late, I realized that I might have to do something more to the controller after I double checked the steering angles output from the model during these failures and could confirm that the angles were correct in these cases.

I decided to increase the proportional gain **Kp** so that the controller can give the car higher power, and to add the derivative gain **Kd** so that the controller has faster response. Meanwhile, I reduced the integral gain **Ki** to mitigate possible system oscillation due to addition of Kd. So basically I turned the original PI controller to PID controller. See newly added lines 32, 36, 53, 54 and the change in line 57 in drive.py.

The original values of these parameters are:

Kp = 0.1, Ki = 0.002, Kd = 0 (actually there is no Kd in the controller).

After about 30 trials with different values of Kp, Ki and Kd, I nailed them down to:

Kp = 0.14, Ki = 0.0001 and Kd = 0.0001 , for track 2 (file drive2.py)

For track 1, I used the original Kp = 0.10, but Ki = 0.001, and Kd = 0.00001 (file drive1.py)

The speed is set to 9 in all the cases above.