Behavioral Cloning

Project Report

1. Submission Files

My project submission includes the following files:

- `model.py` – the script used to create and train the final model.
- `drive.py` – the script provided by Udacity that is used to drive the car. I did not modify this script in any way.
- `model.h5` – the saved model file. It can be used with Keras to load and compile the model.
- `report.pdf` – written report, that you are reading right now. It describes all the important steps done to complete the project.
- `video.mp4` – video of the car, driving autonomously on the basic track.

2. Quality of Code

Originally, all of development for this project I did in Jupyter Notebook, because it is very easy to use in a trial-and-error setting. Before submitting the project, I took all the code from the notebook and put it into a file and named it `model.py`.

I split all the functionality used in this project into separate functions and defined them, providing a brief description of what each function does.

Even though the size of raw data is small enought to be stored in RAM on most modern computers, data augmentation techniques can quickly bloat really fast and make a computer slow. This happened to me in the early stages of this project. I quickly realized that this is not a good way to approach this problem and decided to use a generator to sequentially feed data in batches to the model. I made a generator that preprocesses and augments data in real time during training. This approach allows to use for training even large data sets that cannot comfortably fit into memory.

3. Model Architecture and Training Strategy

Before starting coding the model, I decided to do some research and read about architectures used by other people. I quckly noticed, that nVidia's model was really popular. It was also relatively simple and not very expensive to train. So I started with that model. The model has:

1. Two preprocessing layers, which I will describe later when talking about data.
2. Three convolutional layers with `(5,5)` kernels (24, 26 and 48 kernels per layer, correspondingly) with `(2,2)` strides, followed by
3. Two convolutional layers with `(3,3)` kernels (64 kernels per layer) with `(1,1)` strides, followed by
4. Three fully connected layers (with 100, 50 and 10 neurons, correspondingly), followed by
5. Output layer with one output neuron that controls the steering wheel.

I decided to use ELU (Exponential Linear Unit) activation, because there is evidence that it can be slightly better than RELU. I did not notice any significant difference for my model, but it was already training really fast (around 70 seconds for roughly 14000 of images). All of the convolutional layers and all of the dense layers in my model with the exception of the last layer used ELU nonlinearity.

In order to train a model, I used two generators – one for training and one for validation. Validation data generator was used to assess out-of-sample performance. Training generator was performing random data augmentation to improve generalization capabilities of the model, but validation generator was only performing preprocessing without doing any of the augmentation. I will discuss augmentation procedures further below.

Because essentially the model was performing regression, the most appropriate evaluation metric was mean squre error (`'mse'` in Keras). The only problem with that metric is that it is unclear what it means in the context of driving a car autonomously. This metric, in other words, is not very intuitive. The only rule is the smaller, the better. The optimizer used in the model was an adaptive optimizer Adam with the default parameters.

When training the model as described in nVidia paper, I noticed that training error quickly was becoming smaller than validation error, which is the sign of overfitting. To reduce that I introduced dropout layers after each convolutional and each dense layer with the exception of the output layer. After training the model with different values of dropout I stopped at 0.5 for the final model.

The final architecture is presented in the table below.

Layer (type)	Output Shape	Param #
cropping2d_1 (Cropping2D)	(None, 70, 320, 3)	0
lambda_1 (Lambda)	(None, 70, 320, 3)	0
conv2d_1 (Conv2D)	(None, 33, 158, 24)	1824
dropout_1 (Dropout)	(None, 33, 158, 24)	0
conv2d_2 (Conv2D)	(None, 15, 77, 36)	21636
dropout_2 (Dropout)	(None, 15, 77, 36)	0
conv2d_3 (Conv2D)	(None, 6, 37, 48)	43248
dropout_3 (Dropout)	(None, 6, 37, 48)	0
conv2d_4 (Conv2D)	(None, 4, 35, 64)	27712
dropout_4 (Dropout)	(None, 4, 35, 64)	0
conv2d_5 (Conv2D)	(None, 2, 33, 64)	36928
dropout_5 (Dropout)	(None, 2, 33, 64)	0
flatten_1 (Flatten)	(None, 4224)	0
dense_1 (Dense)	(None, 100)	422500
dropout_6 (Dropout)	(None, 100)	0
dense_2 (Dense)	(None, 50)	5050
dropout_7 (Dropout)	(None, 50)	0
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11

Total params: 559,419
Trainable params: 559,419
Non-trainable params: 0

I must admit that architecture-wise, this project is rather easy, especially with Keras that does all the shape

inference for you automatically. The secret sauce in making the car drive itself well is not so much the architecture, but the data.

I trained model for 20 epochs. I tested the model after 5, 10 and 15 epochs and found car behaviour after 20 epochs the most satisfactory. In addition, training and validation errors were steadily declining throughout all training. Training outpout provided by Keras is presented below. I think I could have continued to train for more epochs, but results were already good and I decided to stop training.

```
model.fit_generator(train_gen,
                    epochs=20,
                    validation_data=valid_gen,
                    validation_steps=number_valid_steps,
                    steps_per_epoch=steps_per_epoch, initial_epoch=0)
```

```
Epoch 1/20
80/80 [==============================] - 106s - loss: 0.0930 - val_loss: 0.0349
Epoch 2/20
80/80 [==============================] - 96s - loss: 0.0445 - val_loss: 0.0320
Epoch 3/20
80/80 [==============================] - 97s - loss: 0.0397 - val_loss: 0.0334
Epoch 4/20
80/80 [==============================] - 97s - loss: 0.0362 - val_loss: 0.0300
Epoch 5/20
80/80 [==============================] - 95s - loss: 0.0349 - val_loss: 0.0293
Epoch 6/20
80/80 [==============================] - 97s - loss: 0.0337 - val_loss: 0.0307
Epoch 7/20
80/80 [==============================] - 97s - loss: 0.0322 - val_loss: 0.0269
Epoch 8/20
80/80 [==============================] - 95s - loss: 0.0317 - val_loss: 0.0280
Epoch 9/20
80/80 [==============================] - 96s - loss: 0.0305 - val_loss: 0.0253
Epoch 10/20
80/80 [==============================] - 97s - loss: 0.0306 - val_loss: 0.0254
Epoch 11/20
80/80 [==============================] - 96s - loss: 0.0303 - val_loss: 0.0250
Epoch 12/20
80/80 [==============================] - 95s - loss: 0.0293 - val_loss: 0.0245
Epoch 13/20
80/80 [==============================] - 96s - loss: 0.0289 - val_loss: 0.0258
Epoch 14/20
80/80 [==============================] - 97s - loss: 0.0288 - val_loss: 0.0228
Epoch 15/20
80/80 [==============================] - 94s - loss: 0.0275 - val_loss: 0.0238
Epoch 16/20
80/80 [==============================] - 96s - loss: 0.0274 - val_loss: 0.0231
Epoch 17/20
80/80 [==============================] - 97s - loss: 0.0278 - val_loss: 0.0238
Epoch 18/20
80/80 [==============================] - 96s - loss: 0.0269 - val_loss: 0.0223
Epoch 19/20
80/80 [==============================] - 96s - loss: 0.0267 - val_loss: 0.0198
Epoch 20/20
80/80 [==============================] - 96s - loss: 0.0264 - val_loss: 0.0226
```

4. Data

A machine learning model is only as good as the data you put in. This is why data collection and processing is one of the most important parts of a successful machine learning application. In the following part I will address the issues related to data collection, preprocessing and augmentation. They were the key part of achieving good results in simulation.

4.1 Data Collection

As per Udacity suggestions, I collected two laps of "good smooth driving" in the center of the lane and one lap of "recoveries", where I was placing the car in an undesired position away from the center of the road and then recorded the part when the car steers back to the center.

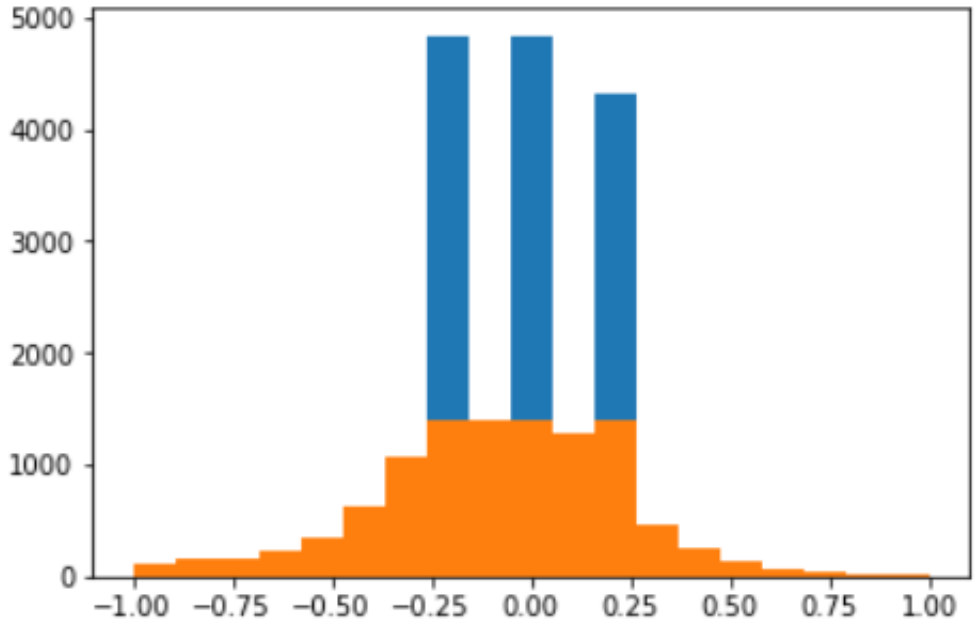
An important point to pay attention to is using keyboard vs. using controller. Analog controller allows to kee steering angle steady at some arbitrary fixed angle and that improves the quality of the data. I used controller to record laps of smooth driving but it was too cumbersome to record recoveries using controller, so I used keyboard instead.

4.2 Data Cleaning

If you look at the histogram below, you will see that raw data is extremely imbalanced. Most of the recoded images have steering angle of 0, 0.25 and -0.25, while all other values are not represented so well. This creates a bias for the model to learn very well these 3 staes and disregard all the other.

A way to address this issue is by removing some of the images with over represented values. I created a script that does just that. It splits the range of all values in bins and counts how many data points blong in each bin. Then it selects the largest count allowed per bin and iteratively randomly removes data points in over represented bins in such a way, that at most only the allowed number of observations are left in each bin.

In the figure below you can see histogram of the data. Together, orange and blue areas make up distribution of original data, whereas the orange area represents distribution of data after cleaning. Blue areas were discarded during data cleaning and as result, most of the steering angles in the central area are similarly distributed.



4.3 Data Preprocessing

Data preprocessing consisted of four steps.

The **first** step is to apply **Gaussian blur** with small value of 3. This step allows the network to not focus on very small details of the image that can sometimes be noise not related to general features of image that we would like the network to learn.

The **second** step was to **convert BGR color scheme to YUV** (the default format used by CV2 is BGR). I did this because nVidia paper mentions it and also because it is easy to change brightness of picture just by adjusting the Y channel. Had I used BGR, it would be more challenging to change brightness. I mention changing brightness, because it will be one of augmentation steps, which I will address later.

The **third** step was to **crop "irrelevant" parts of image**, as per Udacity's suggestions. I cropped 65 upper and 25 lower pixels of each image. I could use CV2 but went with Keras's `Cropping2D`, because it uses GPU to perform computations, and hence can be done in parallel much faster then on CPU.

The **fourth** was **data normalization**. This step consists of dividing each pixel value by 255 and then subtracting 0.5. As result, values are centered at zero and have standard deviation of 1 at most. This preprocessing step was achieved by the meand of Keras's `Lambda` layer with custom function `normalize_pixels()` passed to the layer. Because this step was implemented in the second layer of the network, it was also handled by GPU with all the benefits of fast processing.

During training and validation, all images went through all of these steps.

4.4 Data Augmentation

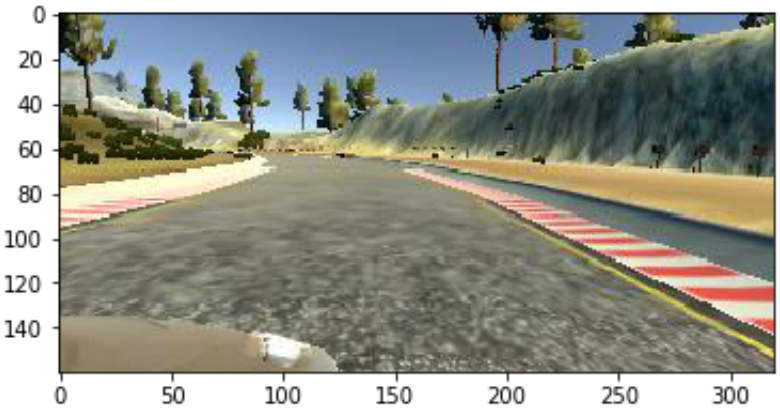
Data augmentation also consisted of four techniques. Data augmentation was used to make more data without having more data. It also allowed the model to generalize better. Only images that were used for training were augmented using techniques described below. There was no augmentation applied to validation data.

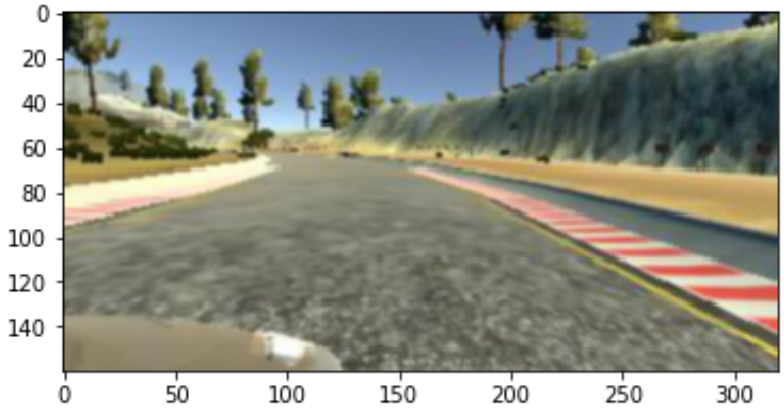
One approach is random **horizontal flipping** of the image with corresponding reversal of the steering angle. During training, the data generator "flips a coin", and with probability 50% will flip an image. The reason for this augmentation technique is to reduce left or right steer bias during training. The training loop is a closed circuit and has relatively more left turns. We do not want the model to learn turning to the left more than turning to the right and therefore we want the model to see all turn directions and magnitudes in equal amounts. This technique allows for that.

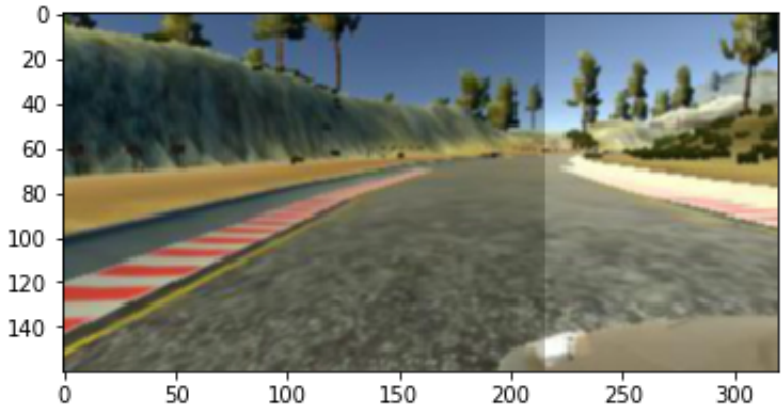
The other 3 approaches were inspired by reading about how other CarND students approached their solutions. The goal of these approaches is to improve model's ability to generalize and learn relevant features. The first technique is **random change of brightness** to allow the model to learn both in dark and bright conditions. The second technique is **random "darkening" of a random area of an image** to simulate shadows on the road and to make model not be confused by the shadows. And the third technique is **random horizont shift** up or down to simulate hilly terrain.

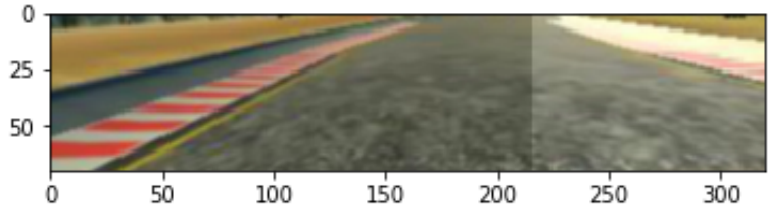
Images below represent all steps related to data processing and augmentation. For convenience, I chose to represent all images in RGB color, whereas starting from step 2 and after that, all images in the network are in YUV colors. Steps are as follows:

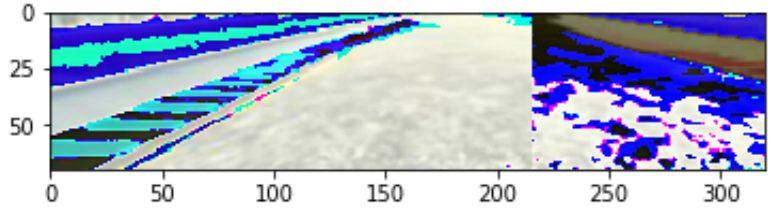
1. Original image
2. Blurred image
3. Augmented image: flip, brighten, shadow, horizon shift
4. Cropped image [layer 1 in network]
5. Normalized image [layer 2 in network]











There was additional approach that helped to increase amount of data without collecting more data. Udacity's simulator car had not one, but three cameras mounted on the car. So there were actually 3 images for each moment of time corresponding to each steering angle measurement. As proposed by Udacity instructor, used this opportunity to improve the quantity of data. I used an angle correction factor of 0.2. For the left images, I added the factor and for the right images, I subtracted the factor. This allowed to simulate recovery behavior of the car. When the car drives in the center of the line, the image from the left camera will appear closer to the left side of the road. This means the car needs to correct by steering to the right, and this is exactly what

adding the correction factor did. The opposite sitution if with the right camera.

This effectively allowed to triple amount of data without collecting any more data, which is really cool technique in my opinion.

4.5 Data Summary

There were `8634` lines in the data log. This means there were `8634 x 3 = 25902` images before cleaning from all 3 cameras and after steering angle correction. After cleaning, when over represented classes were trimmed, there were `12816` images left (the orange area in the histogram above. Then these images were passed to the generator and augmented during training process. I chose each epoch to have twice as many images (`12816 x 2 = 25632`), so that each image, on average, will be presented to the network in its original, as well as flipped form.

Challenges of the Project

Originally I thought that `steps_per_epoch` and `validation_steps` parameters of `.fit_generator()` method require the number of training examples. When I provided numbers of training examples, the training went extremely slow even though I was using a high-end GPU for training. At first, I thought I was hitting the hard drive read bottleneck, because my hard drive is old and slow. I tried to solve this problem by pre-loading all cleaned data into memory and then using that loaded data to pass to train generator and perform augmentation on the fly. I think that sped things up but just a little bit. After some time of frustration I finally realized that I was using `steps_per_epoch` and `validation_steps` parameters all wrong. I then adjusted the values of these parameters and the training started to be as fast as I expected given the speed of my GPU. I learned my lesson and will never forget what these two parameters mean.

I used generators of Keras before, for example `flow_from_directory()` , but I have never written my own custom generator. For some reason, I thought that it is too difficult and my level of Python was not advanced enough to write my own generators. I was mistaken and realized that generators are not that difficult. This was a really good practice and not only for Keras. Generators are widely used in Python and now I feel more confident in my ability to use and create them.

The most important conclusion that I made after competing this project is that you cannot simply take a model and throw data on it and expect it to learn. Data engineering, preprocessing, cleaning, augmentation are all very important to get good results.

I think that Udacity projects are very useful because they stimulate independent research and problem solving, without explicitly telling what to do.