Udacity Self-Driving Car Nanodegree
Project 3: Behavioral Cloning

Mark Helms, January 7, 2018

Goal/Summary

Train a neural network to steer a car in a simulation using Python and Keras. Success is judged upon whether or not an entire lap can be driven without any tires leaving the drivable area. This has been achieved through the use of a convolutional network. In addition, the stand-out task of having the car navigate the mountain road was performed, including maintaining a position in the right lane. This report summarizes the activity involved with the project.

Approach

Step 1: Data Collection

Data was collected by driving two laps around the track both in a clockwise and a counter-clockwise direction. An Xbox 360 controller was used as input in order to provide smooth control. Best attempts were made to stay in the center of the lane the entire time and avoid jerky turning motions.

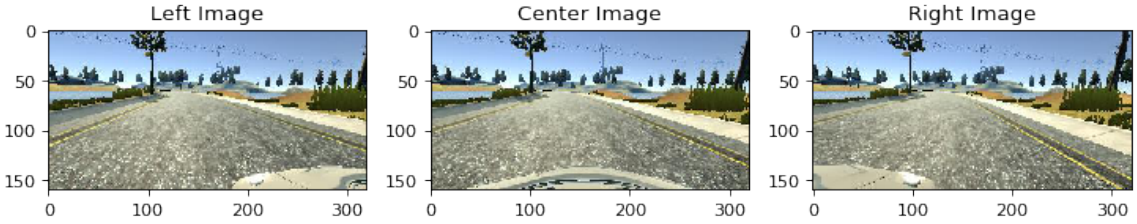
The simulator provided by Udacity records continuous snapshots that include images for the left, center and right cameras (each mounted on the hood of the car and facing forwards), as well as information about the speed, throttle position and steering angle. When the model is being evaluated, the only input available will be the center camera image and the only output required will be the steering angle.

The images are 320x160 (width x height) and are saved as BGR format. They are converted to RGB upon being loaded, due to the fact that the test images will be in RGB format. Steering values range from -1.0 to 1.0. Throttle ranges from 0.0 to 1.0. Speed ranges from 0.0 to 30.0. Brake is always 0.0.

Example data are shown in the following figures. Note that the steering number shown on the left corresponds to the center image.

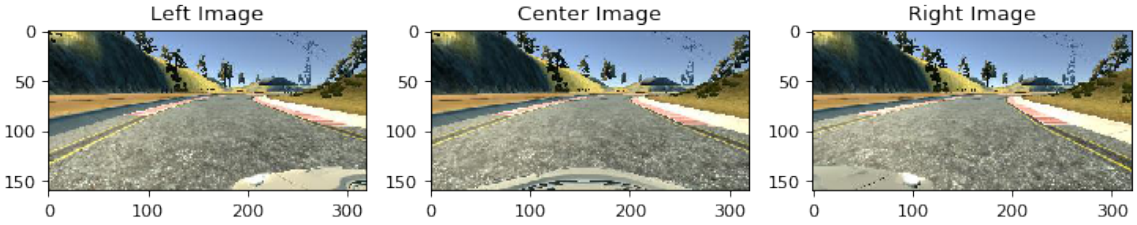
Index 509

Steering: -0.12
Throttle: 1.00
Speed: 30.19
Brake: 0.00



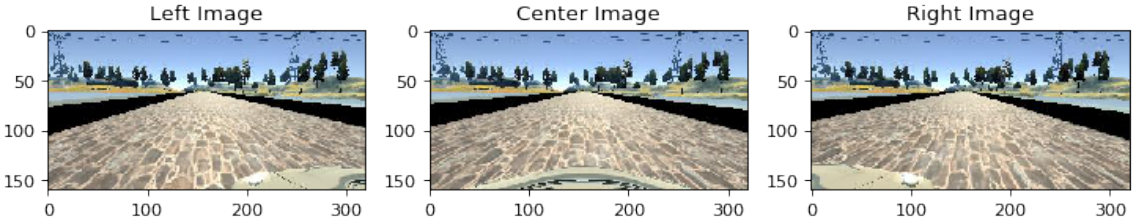
Index 3908

Steering: 0.00
Throttle: 1.00
Speed: 30.19
Brake: 0.00



Index 5721

Steering: 0.00
Throttle: 1.00
Speed: 30.68
Brake: 0.00

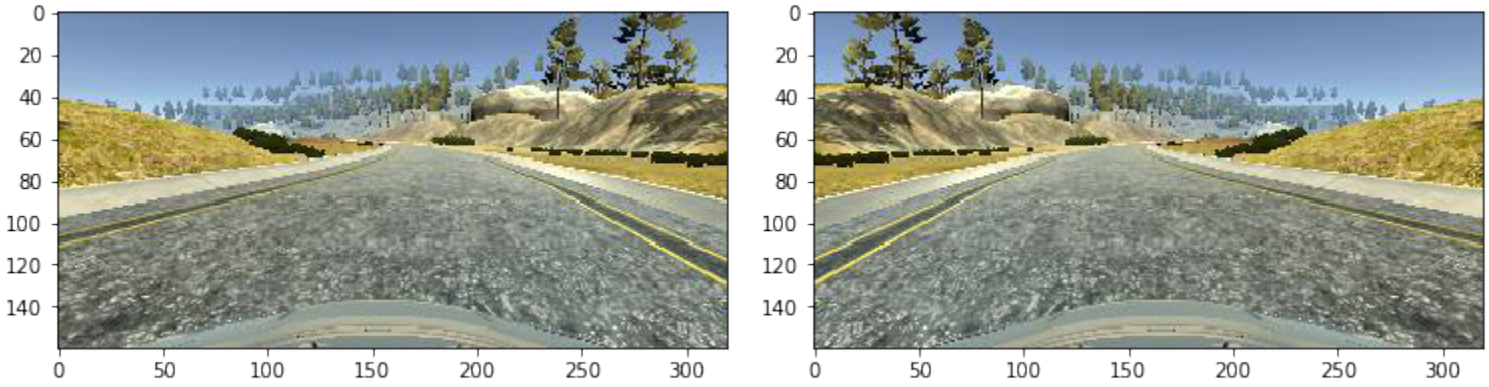


Step 2: Data Augmentation

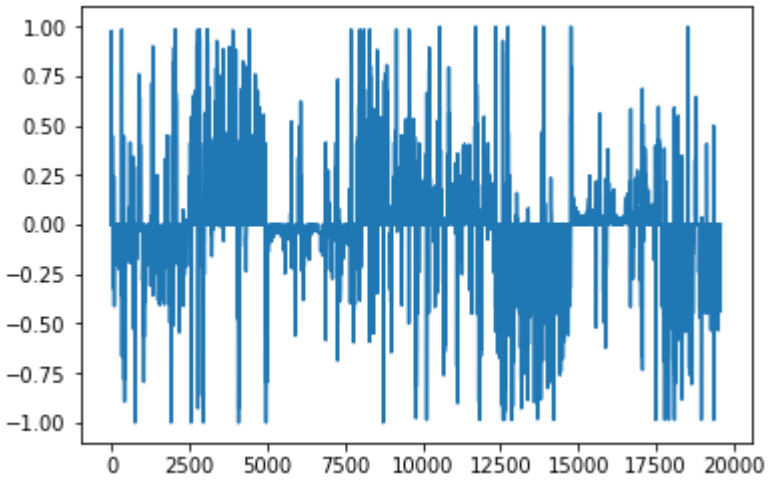
By default, three camera angles are provided, representing left, center and right cameras mounted on the hood and facing forward. These extra angles are used as recommended by Udacity in order to augment the main, center camera image dataset. The left and right cameras are a snapshot of what the car would see if it had wandered too far to the left or right. In order to use them in a regression, the steering angle corresponding to this new perspective should be slightly adjusted by the use of a correction factor. The correction factor chosen, 0.4, was double the factor recommended by Udacity. This adjustment was necessary to make sure that quick corrections would be made if the car wandered too far off to either side.

Horizontal flipping was also recommended by Udacity and was employed. This allows the dataset to be generalized in a left-to-right sense and avoids biases, such as those associated with driving a looped course, whereby by definition one turning angle must occur more frequently than the other.

An example of an image and its flipped version are shown below.



The total history of steering angles vs time can be seen below. Notice how the second half is a negative version of the first half.



Step 3: Model Building

It was required to build the model using Keras. Because the "test" data must be handled on the fly, all preprocessing must be performed in the neural network. This is advantageous, even if not required, as the GPU can parallelize many otherwise costly modifications, if performed on the CPU. The preprocessing steps incorporated were, as recommended by Udacity, clipping and normalization.

Clipping was performed such that the entire width was maintained, while eliminating upper and lower portions of the screen. This forces the network to focus on the most informative section of the image, as pertains to steering.

Normalization was performed in a standard fashion for RGB images, in that each pixel was centered about 128 and then normalized so that the range of pixel values for each channel ranges from -0.5 to 0.5. Interestingly, a Lambda function was used, which is a way to author short functions to implement as Keras layers. Normalization of the images has an incredibly strong effect on the performance of the network.

After preprocessing, what might be thought of as the actual neural network begins. After attempting simpler networks, a network consisting of a triple layer of convolutional layers followed by a triple layer of fully connected (dense) layers was chosen, which is an architecture that is fairly common and resembles the early LeNet architecture. Each convolutional layer is followed by a maxpooling layer, which has a default pooling size of 2x2. A dropout layer is included between the convolutional layers and the fully connected layers, where it can have maximal effect on allowing the fully connected portion to diversify its basis for making decisions based on features seen. The dropout layer wasn't necessary for the entire track, but was necessary in keeping the car from drifting into the dirt section where there were no right-side markings for the road – a clear sign that the network wasn't allowing the left-side markings alone to correct the steering. Having multiple fully connected layers allows for the network to create a complicated decision-making process based on the features visible to the convolutional layers.

The neural network layers are shown in the table below:

Layer	Output Shape / Features	Parameters
Input	160 x 320 x 3	Train/Validation Split = 0.8/0.2
Cropping2D	90 x 320 x 3 (output)	
Normalization	90 x 320 x 3	
Convolution2D + Activation	86 x 316 x 8	Kernel = 5x5, Features = 8, ReLU
MaxPooling2D	43 x 158 x 8	
Convolution2D + Activation	39 x 154 x 8	Kernel = 5x5, Features = 8, ReLU
MaxPooling2D	19 x 77 x 8	
Convolution2D + Activation	15 x 73 x 8	Kernel = 5x5, Features = 8, ReLU
MaxPooling2D	7 x 36 x 8	
Flatten	2016	
Dropout	2016	Keep Prob = 0.5
Dense + Activation	120	ReLU
Dense + Activation	84	ReLU
Dense	1	

Total Trainable Parameters: 256,113

The loss used was the mean squared-error and the optimizer used as the Adam optimizer. All weights and biases were initialized using Keras default settings. The number of epochs performed was 3 and batch size was 32, however only the weights after the first epoch were retained, as they produced the lowest validation loss. Each epoch took about 30 seconds to train over 47,000 samples using an NVIDIA 1080 Ti.

Step 4: Performance

In order to test the performance, the drive.py script was run using the model generated after training the network. Performance on track one was excellent at the 9 mph default speed limit set in drive.py. Performance is mostly stable up to 20 mph. Around 30 mph, a resonance becomes apparent, as the vehicle tries hard to stay in the exact center, while often over-running its target. This could probably fixed one of three ways: 1) retraining with better training data, 2) using a dampening low-pass filter or 3) using a higher steering calculation rate. It's like that option 1 is the best, due to the fact that the dampening filter may hurt the performance during hard turns and a higher steering calculation rate may be too costly.

Step 5: Standing Out / Mountain Course

The same network design was trained using data collected and augmented in the same fashion as for the primary course. The vehicle was only driven on the right side of the road, due to the existence of a certain lane. The performance was surprisingly great. The car occasionally touches the lines with a tire, but not often, and it never leaves the road. The video was taken with the car operating at a target speed of 20 mph.