Behavioural Cloning

Seymur Dadashov

August 25, 2020

Abstract

The purpose of the project is to apply deep learning principles to teach a car to drive autonomously in a simulation after first being provided data from a human driving within the same simulation. The approach is referred to as behavioural cloning and is a formulated as a supervised learning problem. Given some examples of a behaviour, behaviour cloning attempts to generate behaviour that 'matches' the statistics observed from the data given a 'similar' situation. The approach is closely related to *Inverse Reinforcement Learning (IRL)* as the data is a set of demonstrations and the target is the reward function. The project utilizes a Udacity simulation built on the Unity engine and the data will be fed after several pre-processing techniques into a Convolutional Neural Network (CNN) using the Keras framework.

1 Introduction

The object of this project is to apply deep learning principles to effectively teach a car to drive autonomously in a simulated driving application. The simulator includes both training and autonomous modes, and two tracks on which the car can be driven - I will refer to these as the "test track" (which is the track from which training data is collected and on which the output is evaluated for project credit) and the "challenge track" (which includes hills, tight turns, and other features not included in the test track).

In training mode, user generated driving data is collected in the form of simulated car dashboard camera images and control data (steering angle, throttle, brake, speed). Using the Keras deep learning framework, a convolutional neural network (CNN) model is produced using the collected driving data (see model.py) and saved as model.json (with CNN weights saved as model.h5).

Using the saved model, drive.py (provided by Udacity, but amended slightly to ensure compatibility with the CNN model and to fine tune controls) starts up a local server to control the simulator in autonomous mode. The command to run the server is python drive.py model.json; the model weights are retrieved using the same name but with the extension .h5 (i.e. model.h5).

The challenge of this project is not only developing a CNN model that is able to drive the car around the test track without leaving the track boundary, but also feeding training data to the CNN in a way that allows the model to generalize well enough to drive in an environment it has not yet encountered (i.e. the challenge track).

2 Model Documentation

2.1 Initial Structure and Motivation

The project instructions from Udacity suggest starting from a known self-driving car model and provided a link to the nVidia model (and later in the student forum, the comma.ai model) - the diagram below is a depiction of the nVidia model architecture.

The initial step was to reproduce the model depicted in Figure 1 — including image normalization using a Keras Lambda function, with three 5x5 convolution layers, two 3x3 convolution layers, and three fully-connected layers — and as described in the paper text - including converting from RGB to YUV color space, and 2x2 striding on the 5x5 convolutional layers. The paper does not mention any sort of activation function or means of mitigating overfitting, so I began with tanh activation functions on each fully-connected layer, and dropout (with a keep probability of 0.5) between the two sets of convolution layers and after the first

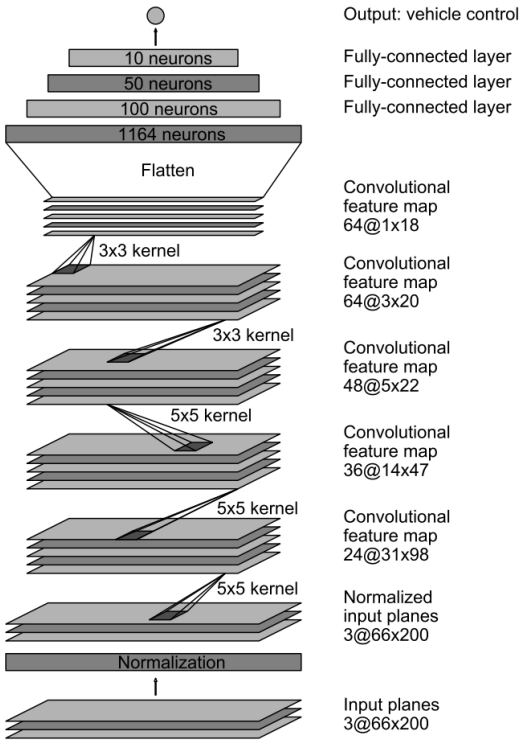

Figure 1: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

fully-connected layer. The Adam optimizer was chosen with default parameters and the chosen loss function was mean squared error (MSE). The final layer (depicted as "output" in the diagram) is a fully-connected layer with a single neuron.

2.2 Loading and Pre-processing

n training mode, the simulator produces three images per frame while recording corresponding to left-, right-, and center-mounted cameras, each giving a different perspective of the track ahead. The simulator also produces a csv file which includes file paths for each of these images, along with the associated steering angle, throttle, brake, and speed for each frame. My algorithm loads the file paths for all three camera views for each frame, along with the angle (adjusted by +0.25 for the left frame and -0.25 for the right), into two numpy arrays `image_paths` and `angles`.

Images produced by the simulator in training mode are 320x160, and therefore require preprocessing prior to being fed to the CNN because it expects input images to be size 200x66. To achieve this, the bottom 20 pixels and the top 35 pixels (although this number later changed) are cropped from the image and it is then resized to 200x66. A subtle Gaussian blur is also applied and the color space is converted from RGB to YUV. Because `drive.py` uses the same CNN model to predict steering angles in real time, it requires the same image preprocessing (**Note, however: using cv2.imread, as model.py does, reads images in BGR, while images received by drive.py from the simulator are RGB, and thus require different color space conversion**). All of this is accomplished by methods called `preprocess_image` in both `model.py` and `drive.py`.

2.3 Jitter

Images produced by the simulator in training mode are 320x160, and therefore require preprocessing prior to being fed to the CNN because it expects input images to be size 200x66. To achieve this, the bottom

20 pixels and the top 35 pixels (although this number later changed) are cropped from the image and it is then resized to 200x66. A subtle Gaussian blur is also applied and the color space is converted from RGB to YUV. Because drive.py uses the same CNN model to predict steering angles in real time, it requires the same image preprocessing (Note, however: using cv2.imread, as model.py does, reads images in BGR, while images received by drive.py from the simulator are RGB, and thus require different color space conversion). All of this is accomplished by methods called preprocess_image in both model.py and drive.py.

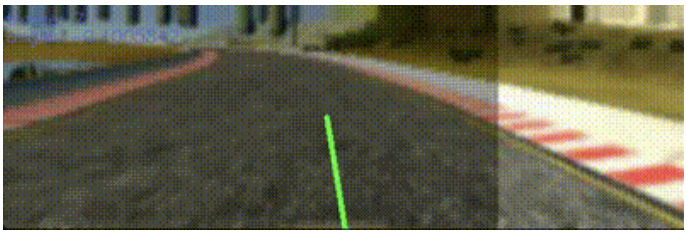

Figure 2: Jitter applied to the image results in a randomized distortion where approximately 75% of the left side has been darkened

2.4 Data Visualization

An important step in producing data for the model, especially when preprocessing (and even more so when applying any sort of augmentation or jitter) the data, is to visualize it. This acts as a sort of sanity check to verify that the preprocessing is not fundamentally flawed. Flawed data will almost certainly act to confuse the model and result in unacceptable performance. For this reason, I included a method 'visualize_dataset', which accepts a numpy array of images X, a numpy array of floats y (steering angle labels), and an optional numpy array of floats y_pred (steering angle predictions from the model). This method calls process_img_for_visualization for each image and label in the arrays.

The process_img_for_visualization method accepts an image input, float angle, float pred_angle, and integer frame, and it returns an annotated image ready for display. It is used by the visualize_dataset method to format an image prior to displaying. It converts the image colorspace from YUV back to the original BGR, applies text to the image representing the steering angle and frame number (within the batch to be visualized), and applies lines representing the steering angle and the model-predicted steering angle (if available) to the image.

2.5 Data Distribution Flattening and Exploring Uniformity

Due to the nature of the test track including long sections with either areas of no curvature or areas of similar curvature, the data captured from it tends to be categorized into 3 mains groups of either left, right or no curvature. Perhaps the initial thought is that this data is good and will address the nature of this problem as intended, but in reality we are creating very heavy biases towards only these three angles and if other angles are introduced, perhaps on a different track, the model will not be able to respond to varying angles it may run into the future. Therefore, in order to make the model more robust the distribution should be adjusted to remove such bias and is represented by the new distribution underneath the black line seen in Figure 3.

To reduce the occurrence of low and zero angle data points, I first chose a number of bins (I decided upon 23) and produced a histogram of the turning angles using numpy.histogram. I also computed the average number of samples per bin (avg_samples_per_bin - what would be a uniform distribution) and plotted them together. Next, I determined a "keep probability" (keep_prob) for the samples belonging to each bin. That keep probability is 1.0 for bins that contain less than avg_samples_per_bin, and for other bins the keep probability is calculated to be the number of samples for that bin divided by avg_samples_per_bin (for example, if a bin contains twice the average number of data points its keep probability will be 0.5). Finally, I removed random data points from the data set with a frequency of (1 - keep_prob).

The resulting data distribution can be seen in the chart below. The distribution is not uniform overall, but it is much closer to uniform for lower and zero turning angles.

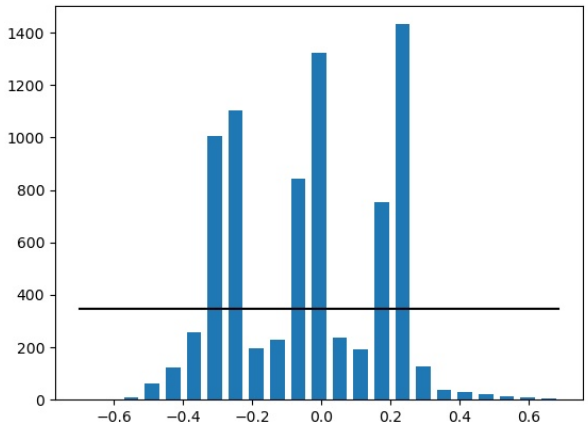

Figure 3: Initial distribution of observed steering angles from raw dataset.

At one point, I had decided I might be throwing out too much of my data trying to achieve a more uniform distribution. So instead of discarding data points until the distribution for a bin reaches the would-be average for all bins, I made the target twice the would-be average for all bins. The resulting distribution can be seen in the chart below. This resulted in a noticeable bias toward driving straight (i.e. problems with sharp turns), particularly on the challenge track.

The consensus from the nanodegree community was that underperforming on the challenge track most likely meant that there was not a high enough frequency of higher steering angle data points in the set. I once again adjusted the flattening algorithm, setting the target maximum count for each bin to half of the would-be average for all bins. The histogram depicting the results of this adjustment can be seen in the chart below.

Figure 4 and 5 show the new distribution of steering angles.

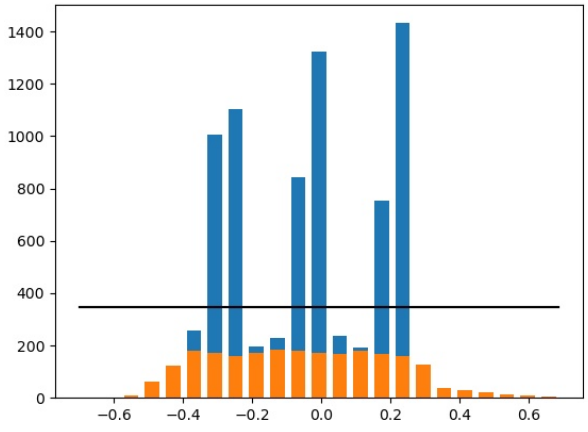

Figure 4: Distribution in orange shows new distribution after removing angle biases.

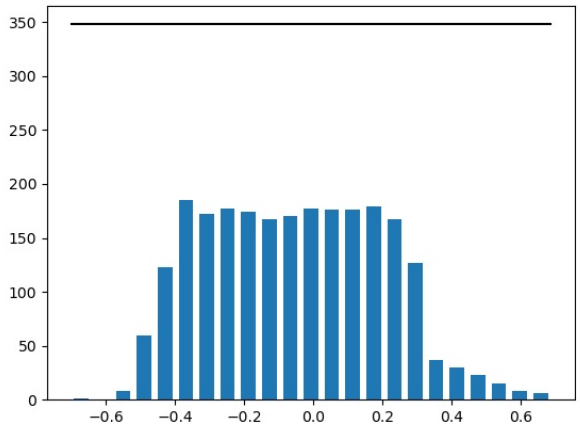

Figure 5: Exploding the orange distribution of Figure 4.

2.6 Implementing Python Generator in Keras

When working with datasets that have a large memory footprint (large quantities of image data, in particular) Keras python generators are a convenient way to load the dataset one batch at a time rather than loading it all at once. Although this was not a problem for my implementation, because the project rubric made mention of it I felt compelled to give it a try.

The generator generate_training_data accepts as parameters a numpy array of strings image_paths, a numpy array of floats angles, an integer batch_size (default of 128), and a boolean validation_flag (default of False). Loading the numpy arrays image_paths (string) and angles (float) from the csv file, as well as adjusting the data distribution (see "Data Distribution Flattening and Exploring Uniformity," above) and splitting the data into training and test sets, is still done in the main program.

generate_training_data shuffles image_paths and angles, and for each pair it reads the image referred to by the path using cv2.imread. It then calls preprocess_image and random_distort (if validation_flag is False) to preprocess and jitter the image. If the magnitude of the steering angle is greater than 0.33, another image is produced which is the mirror image of the original using cv2.flip and the angle is inverted - this helps to reduce bias toward low and zero turning angles, as well as balance out the instance of higher angles in each direction so neither left nor right turning angles become overrepresented. Each of the produced images and corresponding angles is added to a list and when the lengths of the lists reach batch_size the lists are converted to numpy arrays and yielded to the calling generator from the model. Finally, the lists are reset to allow another batch to be built and image_paths and angles are again shuffled.

generate_training_data runs continuously, returning batches of image data to the model as it makes requests, but it's important to view the data that is being fed to the model, as mentioned above in "Data Visualization." That's the purpose of the method generate_training_data_for_visualization, which returns a smaller batch of data to the main program for display. (This turned out to be critical, at one point revealing a bug in my implementation of cv2.flip causing the image to be flipped vertically instead of horizontally).

2.7 Cleaning the Data

Another mostly unsuccessful attempt to improve the model's performance was inspired by David Brailovsky's post describing his competition-winning model for identifying traffic signals. In it, he discovered that the model performed especially poorly on certain data points, and then found those data points to be mislabeled in several cases. I created clean.py which leverages parts of both model.py and drive.py to display frames from the dataset on which the model performs the worst. The intent was to manually adjust the steering angles for the mislabeled frames, but this approach was tedious, and often the problem was with the model's prediction and not the label or the ideal ground truth lay somewhere between the two. A sample of the

visualization (including ground truth steering angles in green and predicted steering angles in red) is shown below.

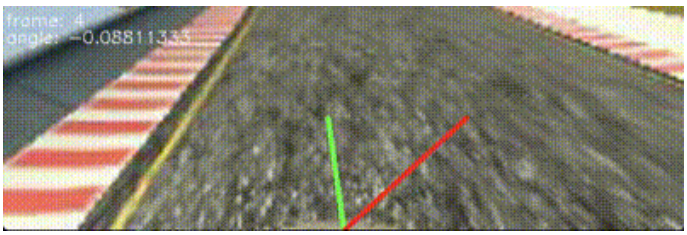

Figure 6: Image displaying incorrect steering angle vector in *red* and the manual correction of the vector in *green*.

2.8 Fine-Tuning and Overfitting

Some other strategies implemented to combat overfitting and otherwise attempt to get the car to drive more smoothly are (these were implemented mostly due to consensus from the nanodegree community, and not necessarily all at once):

- Removing dropout layers and adding L2 regularization (lambda of 0.001) to all model layers - convolutional and fully-connected
- Removing tanh activations on fully-connected layers and adding ELU activations to all model layers - convolutional and fully-connected
- Adjust learning rate of Adam optimizer to 0.0001 (rather than the default of 0.001)

These strategies did, indeed, result in less bouncing back and forth between the sides of the road, particularly on the test track where the model was most likely to overfit to the recovery data.

2.9 Results

These strategies resulted in a model that performed well on both test and challenge tracks. The final dataset was a combination of Udacity's and my own, and included a total of 59,664 data points. From these, only 17,350 remained after distribution flattening, and this set was further split into a training set of 16,482 (95%) data points and a test set of 868 (5%) data points. The validation data for the model is pulled from the training set, but doesn't undergo any jitter. The model architecture is described in the paragraphs above, but reiterated in the image below:

2.10 Discussion

This project - along with most every other exercise in machine learning, it would seem - very much reiterated that it really is all about the data. Making changes to the model rarely seemed to have quite the impact that a change to the fundamental makeup of the training data typically had.

I could easily spend hours upon hours tuning the data and model to perform optimally on both tracks, but to manage my time effectively I chose to conclude my efforts as soon as the model performed satisfactorily on both tracks. I fully plan to revisit this project when time permits.

One way that I would like to improve my implementation is related to the distribution flattening scheme. As it is currently implemented, a very large chunk of data is thrown out, never to be seen again. I find this bothersome, and I feel that wasting data like this (even if it is mostly zero/near-zero steering angles) is a missed opportunity to train a model that can better generalize. In the future, I would like to pass the full dataset (image_paths and angles) to the generator and allow it to flatten the distribution itself, throwing out a different portion of the data each epoch or batch.

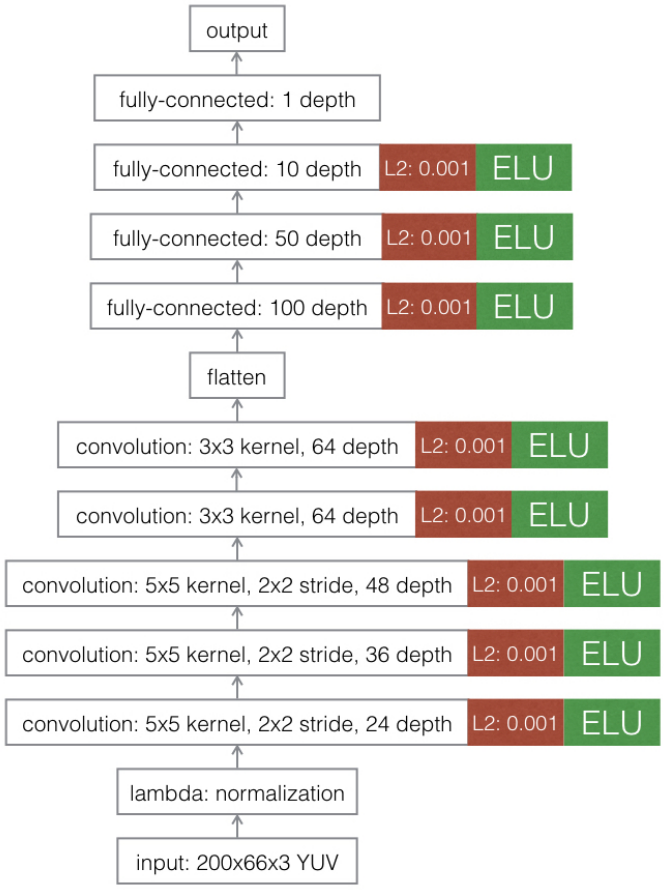

Figure 7: Model diagram

I would also like to revisit implementing a more agressive crop to the images before feeding them to the neural net. Another nanodegree student achieved good model performance in a neural net with only 63 parameters. Such a small neural net is desirable because it greatly reduces training time and can produce near real-time predictions, which is very important for a real-world self-driving car application.

I enjoyed this project thoroughly and I'm very pleased with the results. Training the car to drive itself, with relatively little effort and virtually no explicit instruction, was extremely rewarding.