SDC Term 1 Assignment 4, Behavioral Cloning

Alejandro Terrazas, PhD

https://github.com/alejandroterrazas/SDC-Behavioral-Cloning

Note: I requested and received additional grace time from Udacity. I am somewhat flummoxed by the inconsistency of the results; however, I was able to get a few working models. By using the random number seed, I was able to get consistent results for a given set of parameters. I was not satisfied with the results. For example, if I increased my training epochs from three to six, my model performed slightly worse than if I just used three epochs. I wonder if other students were able to achieve consistent results without using the seed. Overall, it felt like a bit of a random draw.

In addition, I was not satisfied with this project because I would have used other approaches such as reinforcement learning or recurrent neural networks to make better predictions. I believe I could tweak things with this model and achieve better results; however, I decided that was not a wise use of time. I think the overall approach is somewhat flawed.

I believe that the inconsistent performance I am receiving is due to a combination of errors—1) possibly from the different versions of the simulator. For example, I used my Mac to generate the data and ran it live on the Udacity platform. As noted in the instructions, different platforms behave differently. I was not able to run the simulator in autonomous mode on my Mac because I would receive a segmentation fault; 2) some error in my data augmentation scheme, and 3) from using smoothing of the steering angles.

I had very high hopes for this project but, again, am not nearly satisfied with the resuts. I did learn a considerable amount of Pandas and Keras programming. I tried many different optimizers, activation functions, and other aspects of Keras.

Overview

This project, at first, seemed like a "slam dunk"; however, the project fell into several "local minima" that prevented strong results. These challenges are listed below.

The challenges fell into several categories that became important learning opportunities (that is, lessons I won't soon forget!). The exact reasons why the model didn't perform as expected remains a mystery. Data augmentation is a likely source; however, a fairly comprehensive effort was made to check the inputs. Included in the submission are a few versions of the model (model.h5,) and the . Attempts were made to test on ten different training sets. Different forms of the model were tried (oldModel.py, model.py) and different versions of the data generation were tried (only one is included, generate_data.py)

1) Batch normalization tends to force all predictions to the same value, resulting in no steering.
2) It is possible that recording the game play on a different platform (Mac OS) than the playback (Ubuntu) can result in poor results. This was not tested systematically.

3) Data preparation, balancing of datasets, and augmentation are far more important than the particular model design.
4) There may be an issue with my **drive.py** or with my training data collection using the simulator. I made edits to drive.py in order to perform multiple regressions (e.g., angles and speed) and other ideas. I tried to reset. I also had several problems with the environment that interfered with my efforts.

Multiple regression model

At the start, the goal was to estimate all four variables and feed them to the model. Some code with two output variables is included in the model (commented out). However, the multiple regression approach was jettisoned as the GPU time dwindled and the project became seriously overdue.

The model has two data preparation steps, one for centering the image values and another for cropping the image. Following these layers, the model uses a series of four convolutional layers, of depth 32, 64, 128, 128. These layers have small kernels of 3x3 with the exception of the fourth convolutional layer with a 2x2 kernel. All of the convolutional layers use a rectified linear unit (RELU) activation function and Xavier Initialization. Finally, all of the convolutional layers have a 2x2 max pooling layer.

Following the four CNN layers, are four fully connected layers with linear activation functions. The fully connected layers started out somewhat wide (128 units) and paired down to 64 and then 16 units. The final output layer of the model is a single unit with linear output that represents the steering angle. Dropout was used to improve generalization, although the images provided are all from the simulator; thus generalization is not so much of an issue. As noted, however, I suspect there is an incompatibility between the simulator running on my Mac (where I generated the training images) and the simulator on the Udacity platform where the model was run.

The model was trained with backpropagation using an Adaptive Moments optimizer (ADAM optimizer). The ADAM optimizer is known to converge quickly for many network types. Because the model was a regression model root mean square error was used as the cost function.

Data Preparation and Augmentation

Most of the game play data (acquired Training Mode) were acquired on a Mac Air. The github repository (https://github.com/udacity/self-driving-car-sim) was installed. Data augmentation was developed on the local machine prior to deploying on the Linux environment provided by the course materials. Note: this may have been an important strategic error. It is not clear whether the different platforms resulted in the lack of a consistent and successful model. Two training datasets were collected on the Udacity environment for comparisons with the MacOS acquired data. The results of those comparisons were not conclusive. It was not possible to run the model on the Mac OS as the eventlet package would not install properly.

The data obtained in training mode were highly unbalanced with most angular values falling just to the minus side of zero. The track model was a combination of mostly slight left

turns with a few straight aways and a few sharp left- and right- hand curves. The small left bias is due to the left side bias of the training data. One training dataset was acquired by driving in the opposite direction (by turning the car around in a "dirt parking area") and then acquiring the training data (these data were skewed slight positive). Several example images are shown in Figure 1.

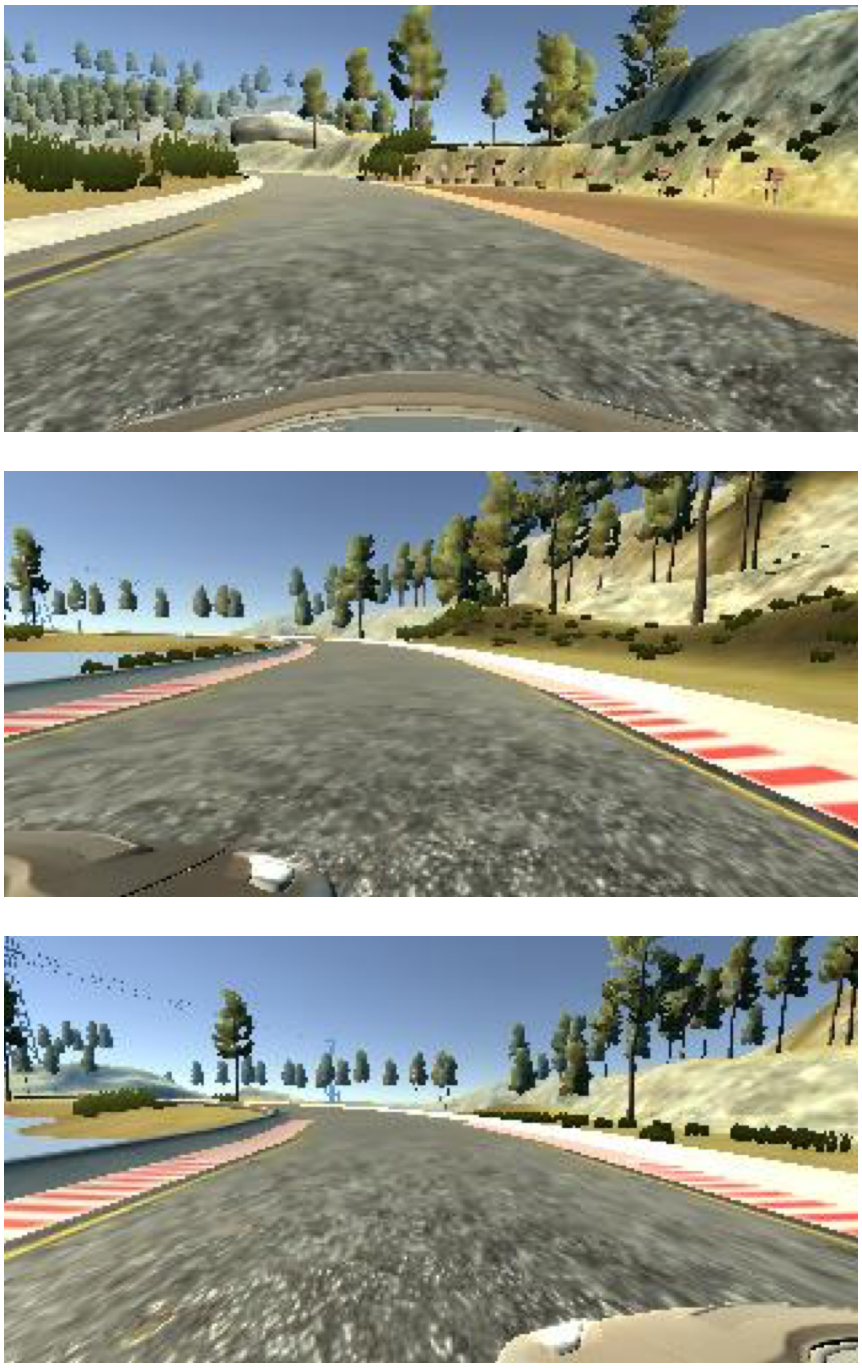

Figure 1: Sample images from the Udacity Simulator*. The cameras were used: center, right and left. The top panel shows an image from the center camera that was primarily used for training the network. The image shows the front of the simulated car in the foreground. The car is approaching a curve and is well centered. The middle panel shows a view from the right camera when the car is pointing to the left. Thus, as training image, this view should be considered as a center image; thus, .25 was added to the angle value for this training image. The bottom panel is from the left camera (note hood of car off to the right. Again, as a training image, the angle should be adjusted by -.25.*

A standard approach to data augmentation was used. Many individual runs in training mode were acquired and kept in separate folders. One goal for the project was to average models between different drivers and training sessions. That goal was not achieved due to the complications that occurred.

The .csv files that hold the names of the images and the values of the regressors (angles, speeds, throttles and brakes) were read in separately (selected with folders = ['folder1', 'folder2' etc.] in order to select different subsets of the training data. Separate row were output for center, left and right images with duplicate regressors copied into each row. In the final analysis, creating separate rows for center, right and left was not necessary; however, this choice was maintained throughout the remainder of the project.

Once the csv files were merged, the combined dataset was shuffled. Several approaches to selected specific images were tried. For example, left- and right-camera images were selected and fixed values (e.g., .1:.25) were added or subtracted (added for left camera, subtracted for right). Another approach was to only accept a fraction of the left- and right-camera images. All of these efforts were tried but, as noted, the results were not consistent.

Following the creation and shuffling of the data, the angular values were binned in the range -3.5:3.5 inclusive using 20 bins. These values were then normalized through division by number of elements in the most frequent value (again, near zero). These percentages determined the likelihood of augmentation such that training batches would be balanced in terms of a uniform distribution of angular values.

Sampling from a Bernoulli distribution was used to determine whether to augment in general (again, lower frequency angles would be more often augmented while higher frequency angles were less often augmented). Within the augmentation scheme, further sampling from the Bernoulli distribution (often simply a "coin flip") was use to add noise, flip (left-right) and adjust the brightness of the augmented image. In order to find the most appropriate candidate image for flipping, the image to flip was searched from among the opposite images closest to the -1*angle. In other words, if an image at -0.5 was desired, an image near +0.5 was selected.

Playback

The GPU time expired on this project before a movie could be created. The best model wasn't saved, unfortunately. The remedy is to create an Ubuntu instance with GPU and use that. Once the dust settles, I will finish this effort. Several models are included; however, none worked perfectly. Some were close but were, unfortunately, not backed up because I thought they could be easily replicated going forward.

As noted, a great deal of GPU time was wasted with Batch Normalization, which resulted in only a single predicted value regardless of the input. I learned to print out the predictions iteratively so that this situation could be detected early but that didn't occur until substantial GPU credits were used. These predicted numbers were coften lose to zero and, therefore, the car would drive straight ahead and quickly off the road.

Several other conditions resulted in a constant prediction output. For example, using bias would sometimes result in a constant value for all input images. Otherwise, the number of

layers, model complexity and other aspects of the model design were not critical (at least as far as I could tell). Again, the main issue was likely the quality of the input images and balance augmentation of the batches. The reason remains a mystery.

Train_on_batch versus fit_iterator

Both train_on_batch and fit_iterator versions of the model were used. This was because it was not immediately obvious what the issue was with batch normalization. The results from these two approaches should be more or less the same. Train_on_batch provides more control over parameters like batch size and intermediate results on accuracy. Fit_iteratory, however, offers several advantages.

angle

Model Design

Both train_on_batch and fit_iterator versions of the model were used. These are shown in model.py which is based on using Sequential() and fit_iterator and oldModel.py, which is based on adding layers using the postfixed (model) approach and train_on_batch.

The precise model chosen is less important. To see if there was something about the form of the model I had chosen, I copied the architecture from a previous student' submission https://github.com/windowsub0406/SelfDrivingCarND/tree/master/SDC_project_3

The oldModel.py is my original model architecture. I have stored my models, data, and python code at my github account.