Guiding Self -Driving Car Using Behavioral Cloning

Madhu Dev Just now · 7 min read

This project illustrates how deep learning can be used to clone driving behavior to guide Self-Driving cars as part of Project # 4 of Udacity Self-Driving Nanodegree.

Convoluted Neural Networks (CNNs) are widely adopted for pattern recognition tasks in image analysis. These algorithm captures the features in images using convolution operations with relatively fewer parameters in comparison to total number of operation. Additionally, they can implemented on massively parallel processing units like GPU's to accelerate learning and inference. This project, implemented with CNN's, illustrates the concept of transferring human driving behavior to machines or Self-Driving cars.

The Objective of the project is to train a neural network to guide a Self-Driving car around a track in a simulator. During the training Phase, the model learns the steering angles for different position of car in relation to track. The acquired knowledge will then be used to predict the steering angles during the drive phase. The implementation is based on the Nvidia paper and uses Keras / Tensorflow deep learning library.

The code for this project is available here

1. Data collection strategy using Simulator

The data for the training was collected using the simulator provided by Udacity. This simulator has two modes. Training mode and Autonomous mode. In the training mode, its possible to control the car with help of mouse and guide it around the track. The simulator additionally gives control to start or stop recording of images and the corresponding steering angles as the car drives around the track.In Autonomous mode, trained model can be tested. I recorded the following simulated drive around the track.

1. Drive one lap around the track forward and in opposite direction keeping the car at the track center. This remove the bias to towards left turn or right turn.
2. Recovery laps. I drove the car to the edge or out of the road. Then record the recovery of the car to the center of the track. This process was repeated again for drive forward and reverse direction of the track.This results in extreme steering angles and gives a uniform distribution of recorded steering angles in training data. Essentially it removes the bias towards straight driving.
3. To avoid the model memorizing the Track, i recorded a single lap of drive in a more challenging track Track2 provided by the same simulator. Track2 has sharp driving curves resulting in extreme angles and sharp lighting changes. Adding data from the Track2 will lead to generalization of the neural network.

The complete datasets recorded were 18747. After randomly shuffling this data set, the total data set is then split into training and validation datasets in the ratio 4:1 for the use in training and validation. This approach ensures the same data is not used for training and validation.

2. Preprocessing and Augmentation of training data

Preprocessing and Augmentation of images are required to enhance convergence, minimize overfitting and speed up the training process. The image processing pipeline was implemented in python using Computer Vision Library. The code for data processing, network model and training is available in the file *model.py*

2.1 Loading the Data in memory

The training data set is a gigantic set of images created during the data collection stage. Loading them into the memory for processing in one shot will require huge amount of RAM. Instead i used a generator function to process only a set of images with size equal to user-defined batch size. The generator functions like a indefinite iterator that processes batches of images and delivers the processed data for training the model just when its needed. Such a functionality is accomplished using infinite loop in conjunction with Python keyword *yield*. This is implemented in *(line 182)*

2.2 Left and right camera Images

The Simulator records images with left, right and center cameras. While the center camera image corresponds to the recorded steering angle, the steering angles for left and right camera images are derived by adding and subtracting center steering angle by a calibrated value 0.21 respectively. The addition of two more data points or images for every center image augmented the training datasets by a factor three. This is implemented in the function *Choose_image_from_camera() (line 107)*

Image from Left , Center and Right camera

2.3 Randomly flip images

The training data may have bias either to left turn or right turn. Consequently, the trained network may also show bias towards left or right turn. To avoid this scenario, the images are randomly flipped in vertical direction and their steering angle multiplied by -1. This is implemented in the pipeline function *im_preprocess() (line 90)*

Image on Left is flipped along height to get Right image

2.4 Cropping of images

The images recorded by the simulator were of dimensions 160x320x3. This included irrelevant details like skies, mountains and trees. Those details were cropped out of the training images to prevent the network from learning irrelevant features. The functionality to crop an image is added as layer to the network model*(line 218)* instead of adding to the image processing pipeline. Advantage here is, we can still feed the raw images from simulator directly to model without resizing.

Raw image and cropped image

2.5 Image Normalization

Image normalization ensures that each input parameter, pixels in this case, has a similar data distribution. This makes convergence faster while training the network. Furthermore, it delivers a well-conditioned input data to the model that is easier for network to learn the features. Normalization is integrated as a Lambda Layer in the network model. Its code can be found in *line 215*

3. Developing and Training the model

The entire network is implemented and trained using Keras and Tensor flow as Backend. Code for building network model and training is available in model.py (lines 212 and above).

3.1 Network Architecture

The Neural Network Architecture i used, is based on the paper published by NVIDIA, developed to map the raw pixels to steering commands. The network consists of 9 layers, including a normalization layer, 5 convolution layers and 3 fully connected layers.

The Image is normalized in Normalization layer. According to the Paper, performing normalization in the network allows the normalization scheme to be altered with the network architecture and to be accelerated via GPU processing.

There are 5 convolution layers designed to extract the features from the images. The first 3 Convolutions are performed with strided convolutions with 2x2 strides and a 5x5 Kernel. The last two convolution layers are obtained through non-strided convolutions performed with 3x3 kernel.

The convolution layers are followed by 3 full connected layers leading to the output steering value. The model is implemented in Keras with Tensorflow as backend*(line 212:237)* . My final model architecture is shown in the image below.

Final neural network for predicting steering angles

Adam Optimizer was used to update the network weights. This optimizer required no tuning as it adapts the learning rate during training*(line 248)*

3.2 Optimizing the Model

The model was optimized by implementing the following measures.

3.2.1 Initially the model was underfitting and the car showed a bias towards straight driving, failing to turn at curves. By adding recovery laps, more data samples using data augmentation discussed earlier, a drive on Track2, the bias towards straight drive was eliminated.

3.2.2 Later the car was taking wrong turns or driving along the edges of the track. I noted that the validation losses were significantly higher than the training losses. I identified that the problem was due to overfitting. This was minimized by adding dropout layers after all layers with drop probability at 0.4

3.2.3 To save the model with least validation loss, i used the ModelCheckPoint () function in Keras which saves the best model while continuing to run for 20 Epochs. I added a early stopping with the condition to stop if five consecutive epochs show no improvement to save time.

4. Testing the model on track1

The final step is to let the car in the simulator to drive autonomously. The images from the simulator as the car drives are opened by the python file *drive.py* . Using the above trained model the steering angles are predicted for the image sent by the simulator. The predicted angles are then fed back to the simulator to move the car to next position. This technique is based on the concept of Deep learning Inference. With the trained model, i could make the car drive through the track without leaving its marked edges. The video below shows the images from the center camera during the autonomous drive.

5. Conclusion

This project of Udacity is a challenging one specially for beginners in machine learning. Unlike the previous projects, this one involves collecting the training data by self. The task of collecting training data, devising a strategy for it can itself be daunting at first. Next comes the difficulty in setting up the model, processing the input data and optimizing the model parameters. More often i might have concluded that its impossible to solve. The project is based on deep learning but it provides scope for us to learn deeply. That's the only way to solve this problem. Once i saw the car navigating the track with help of my trained model, the entire struggle to solve this problem vanished and i was filled with sheer joy. I gave me a great deal of confidence that i can solve challenging machine learning problems !

Self Driving Cars Machine Learning Computer Vision Behavioral Cloning Keras