Traffic Sign Classifier

Eeshan Deosthale 06/07/2020

Introduction

The objective of the behavior cloning project is to train a neural network to drive a car around the track in the Udacity simulator by providing appropriate steering inputs. The behavioral cloning is done by feeding the neural network with dataset which describes to it how to drive the car around. The dataset consists of images of the track taken from the cameras mounted on the vehicle and corresponding steering angles associated with the position of the vehicle on the track.

This project takes the advantage of the transfer learning technique which involves modifying a pre-existing and already implemented Convolutional Neural Network. The idea is to modify and implement a pre-developed neural network designed for a similar task and which operates on a similar or a different dataset. Important steps in this project are as follows.

1. Dataset exploration
2. Preprossessing training data
3. Neural network architecture selection and modification
4. Training the Model
5. Testing the model on Udacity Simulator
6. Summarize the results

As a first step to develop code for this project, it is important to import necessary packages in python.

Listing 1: Import Packages 1

```
import pickle
import numpy as np
import random
import tensorflow as tf
import csv
import cv2
import glob
```

Next, keras packages and functions are imported which are required to build a neural network model easily.

Listing 2: Import Packages 2

```python
from keras import optimizers
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout, Lambda
from keras.layers import Cropping2D
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D

from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
```

Dataset Exploration

The dataset required to train the behavioral cloning network is a set of images of the environment taken from the front cameras mounted on the car. There are 3 cameras mounted at the left, center and right of the car. Each captured timestamp consist of images from all three cameras. A set of images captured from the three cameras is shown in Fig. 2. The choice of using images taken for one or multiple cameras is up to the user. Each set of images from three cameras is associated with the steering angle at that timestamp. This project comes with a initial dataset of a car drive around the track. A quick summary of the dataset is given in Table 1. The dataset has more than 24000 examples available and each image is color image with RGB channels.

It is interesting to see the number of images available for different steering angles. Since, this is not a classification problem distinct classes of steering angles are not available. The steering angle of range of -1.0 to 1.0 is thus divided into 25 different bins and the number of images available per bin is visualized in Fig. 1.

This quick dataset exploration reveals that there is a large bias towards images with steering angles very close to zero. This means that in while collecting the data it was okay to drive straight most of the times and only apply corrective inputs when the car starts to go off track. It is visible that almost no images are available for steering angles more that 0.5 or less than -0.5. Though some bias towards straight driving is necessary for this track, unavailability of corrective data may cause vehicle to go off track as the network will value driving straight over driving with higher angles as a result of inherent bias in the dataset.

Number of center images	8036
Total number of images	24108
Image data shape	$(160, 320, 3)$

Table 1: Summary of the Behavioral Cloning Dataset

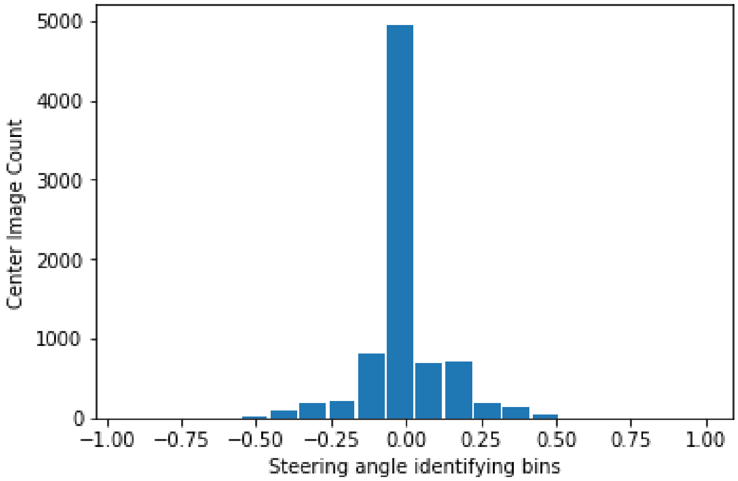

Figure 1: Number of Images Per Steering Angle Bins

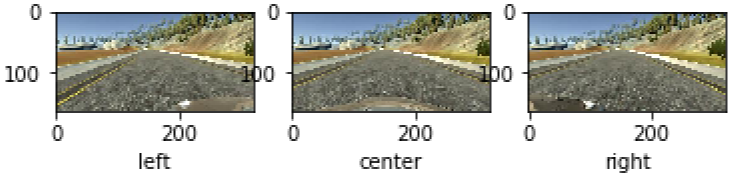

Figure 2: Images taken from Left, Center and Right Cameras

Preprocessing Training Data

The data preprocessing primarily involved three steps.

1. Normalizing or leveling the dataset to reduce bias towards images corresponding to steering angles close to zero.
2. Flipping the images to account for left and right turn and reduce bias towards a specific sign of the steering angle
3. Cropping the images to avoid interference of non-track environment while training and operating the network

Leveling the Dataset

As described in previous section, the dataset has high variability in terms of number of training examples available for different steering angle ranges. To level the dataset, first a average number of images per bin is calculated by dividing the total number of images by total number of bins. A threshold is then set by multiplying this number by a constant and this threshold represents the minimum number of images required per bin before training the network. Now as a starter, the threshold is achieved by randomly duplicating images from that bin. Apart of this method, another way would be to collect additional recovery data to augment the dataset. After leveling, the dataset looks as shown in Fig. 3.

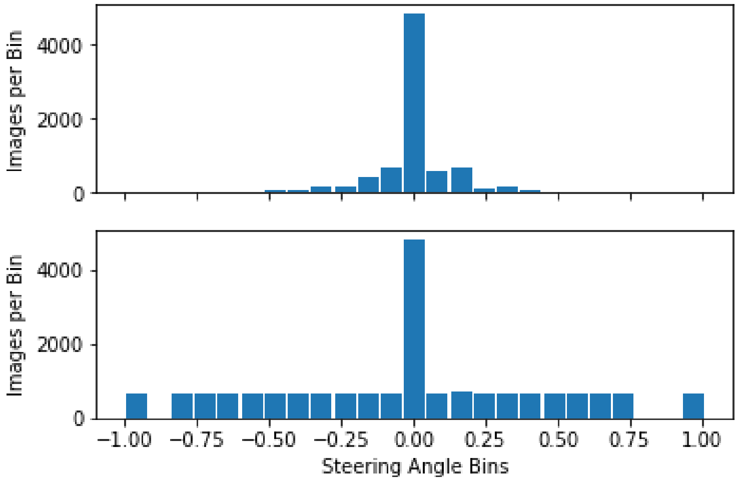

Figure 3: Number of Images Available Per Steering Angle Bins

Listing 3: Dataset Leveling

```
## Bring the number of images per bin to a certain threshold
def make_uniform(samples):
    no_bins = 25
    augmented_samples = []
    count_thresh = int(len(samples)/no_bins)*2
    samples_arr = np.array(samples)
    angles = np.array(list(map(float, samples_arr[:,3])))
    angle_bins = np.linspace(-1., 1.01, no_bins + 1)
    print(len(angles))
    for i in range(no_bins):
        idx = np.where((angles>=angle_bins[i]) & (angles<angle_bins[i+1]))↩
            [0]
        if len(idx) < count_thresh and len(idx) > 0:
            idx_sel = np.random.choice(idx, count_thresh - len(idx))
            samples = samples + samples_arr[idx_sel].tolist()
    samples_arr = np.array(samples)
    angles = np.array(list(map(float, samples_arr[:,3])))
    print(len(angles))
    return samples
```

Flipping and Cropping Images

Most of the collected data appears to be taken by driving following one direction on the track. Since the track in one direction is biased for left-turns the network will likely struggle to generate accurate results when encountering right-turns during the test. Hence, images are flipped while training the network. Flipping images is included as a part of preprocessing routine in the generator function.

Second type of preprocessing that is done by cropping the images to avoid the influence of non-track environment. This is done by removing the top 75 pixels and bottom 25 pixels from the image. Keras provides the lambda function which can be used to preprocess the images on the go while training the network to save memory. This function is used to crop the images. As a demonstration, the flipped and cropped images are shown in Fig. 4. The code in Listing 4 shows the preprocessing routine that gets called from the generator function to flip the images. The cropping part is directly included with Keras model.

Listing 4: Preprocessing Routine Called from Generator

```
def preprocess_image(images, measurements, to_flip = 0):
    X_train, y_train = images, measurements
    if to_flip == 1:
        flip_measurements = -1.0*measurements
        flip_images = []
        for image in images:
            flip_images += [cv2.flip(image, 1)]
        X_train = np.concatenate((X_train, flip_images), axis = 0)
        y_train = np.concatenate((y_train, flip_measurements), axis = 0)
    return X_train, y_train
\label{list:preprocess}
```

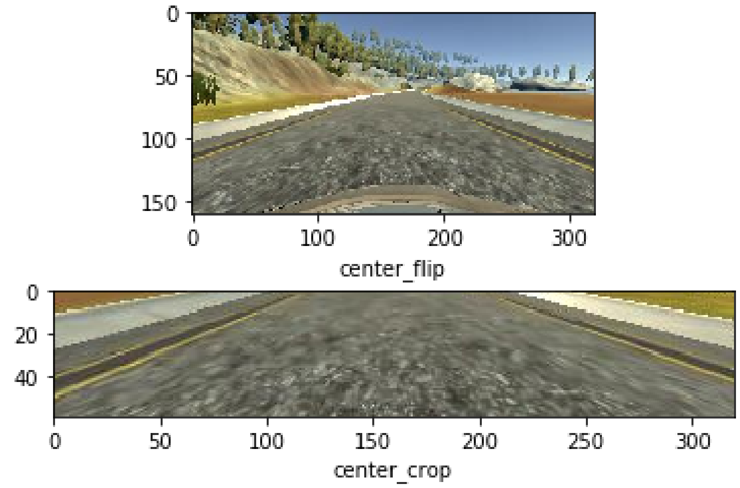

Figure 4: Flipped and Cropped Image of ’Center’ Image from Fig. 2

No further preprocessing is required to be done to train the neural network that works on the first track. A generator function is used send the training data to the network in batches.

Listing 5: Generator Definition

```
def generator(samples, batch_size = 32, is_validation = 0, include_side = 0,↩
    to_flip = 0, sample_size = 2000):
    samples = random.sample(samples, k=sample_size) if is_validation == 0 ↩
       else shuffle(samples)
    num_samples = sample_size
    while  True:
        shuffle(samples)
        for offset in range(0, num_samples, batch_size):
            batch_samples = samples[offset:offset+batch_size]
            center_images, left_images, right_images = [], [], []
            center_measurements = []
            for batch_sample in batch_samples:
                center_images += [cv2.cvtColor(cv2.imread('./data/IMG/'+↩
                   batch_sample[0].split('IMG/')[-1]), cv2.COLOR_BGR2RGB)]
                center_measurements += [float(batch_sample[3])]
                if include_side == 1:
                    left_images += [cv2.cvtColor(cv2.imread('./data/IMG/'+↩
                       batch_sample[1].split('IMG/')[-1]), cv2.COLOR_BGR2RGB↩
                       )]
                    right_images += [cv2.cvtColor(cv2.imread('./data/IMG/'+↩
                       batch_sample[2].split('IMG/')[-1]), cv2.COLOR_BGR2RGB↩
                       )]
            images = np.array(center_images)
            measurements = np.array(center_measurements)*5.
            if include_side == 1:
                images = np.concatenate((images, left_images, right_images),↩
                    axis = 0)
                measurements = np.concatenate((measurements, measurements + ↩
                   0.2*5., measurements - 0.2*5.), axis = 0)
            if is_validation == 0:
                X_train, y_train = preprocess_image(images, measurements, ↩
                   to_flip = to_flip)
            else:
                X_train, y_train = images, measurements
            yield shuffle(X_train, y_train)
```

Neural Network Architecture Selection and Modification

There are various neural network architectures available in the public for the problem of behavioral cloning as this field has been in research since quite a few years and the research in the field of automated vehicles has been growing rapidly. There are architectures available from industry as well as academia. Though each architecture is unique in its own sense most of the architectures share a common framework of having a few convolutional layers and few dense layers. The most common activation functions used in these architectures are 'ELU' and 'ReLU'. After reading a few papers and articles the NVIDIA End-to-End deep learning architecture is chosen for this project. Some of the architectures are, comma ai [1], NVIDIA [2].

The architecture consists of five convolutional layers followed by four fully connected layers. 'ELU' activation is used for all the layers except the last fully connected layer. Dropout is used for all layers to avoid overfitting the training data. The final layer has a single output which represents the steering angle. The loss function used here is mean squared error. The final set of hyperparameters is represented in Table 2.

Hyperparameter	Value
Sample Size	3000
Batch size	32
Dropout (keep_prob)	0.5
Epochs	15
Learning rate	0.001
Optimizer	Adam
Loss function	Mean Squared Error

Table 2: Neural Network Model Parameters

Listing 6: Model Definition

```
## Define Model Architecture
def model_nvidia(train_samples, validation_samples):
    batch_size = 32
    sample_size = 3000
    train_generator = generator(train_samples, batch_size = batch_size, ↩
       is_validation = 0, include_side = 1, to_flip = 1, sample_size = ↩
       sample_size)
    validation_generator = generator(validation_samples, batch_size = ↩
       batch_size, is_validation = 1, include_side = 0)

    model = Sequential()
    model.add(Lambda(lambda x: (x/255.0) - 0.5, input_shape=(160,320,3)))
    model.add(Cropping2D(cropping=((70,25),(0,0))))
    model.add(Conv2D(24, 5, strides = (2,2), activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Conv2D(36, 5, strides = (2,2), activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Conv2D(48, 5, strides = (2,2), activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Conv2D(64, 3, activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Conv2D(64, 3, activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(100, activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Dense(50, activation=’elu’))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation=’elu’))
    model.add(Dense(1))

    opt = optimizers.Adam(lr=0.001)
    model.compile(loss='mse', optimizer=opt)
    model.fit_generator(train_generator,
                        steps_per_epoch = sample_size//batch_size,
                        validation_data = validation_generator,
                        validation_steps = len(validation_samples)//↩
                           batch_size,
                        epochs = 15, verbose = 1)
    model.save('model.h5')
```

Layer	**Description**
Input	$160 \times 320 \times 3$ RGB Image
Convolution 5×5	2×2 stride, 'Valid' padding, 24 output layers
ELU	
Dropout	keep_prob $= 0.5$
Convolution 5×5	2×2 stride, 'Valid' padding, 36 output layers
ELU	
Dropout	keep_prob $= 0.5$
Convolution 5×5	2×2 stride, 'Valid' padding, 48 output layers
ELU	
Dropout	keep_prob $= 0.5$
Convolution 3×3	1×1 stride, 'Valid' padding, 64 output layers
ELU	
Dropout	keep_prob $= 0.5$
Convolution 3×3	1×1 stride, 'Valid' padding, 64 output layers
ELU	
Dropout	keep_prob $= 0.5$
Flatten	
Fully Connected Layer	100 output neurons
ELU	
Dropout	keep_prob $= 0.5$
Fully Connected Layer	50 output neurons
ELU	
Dropout	keep_prob $= 0.5$
Fully Connected Layer	10 output neurons
ELU	
Dropout	keep_prob $= 0.5$
Fully Connected Layer	1 output neurons

Table 3: NVIDIA Model Architecture for Behavioral Cloning

Training CNN Model

The model is trained on training data and the model hyperparameters are tuned base on the loss obtained on training and validation dataset. A consistently lower loss on training dataset than validation dataset indicates the model overfitting the data. After a few such tries, a dropout is introduced in the model to prevent overfitting. Changes were then made to the hyperperparameters such as batch size, size of random sample, dropout percentage and learning rate etc. After training and testing the model for a few times it is noticed that the model did not perform very well on the portion of the track where the road texture changes; e.g. Fig 5. Additional recovery data is then collected to train the model which improves driving performance. The performance of the final model is indicated in Table 4.

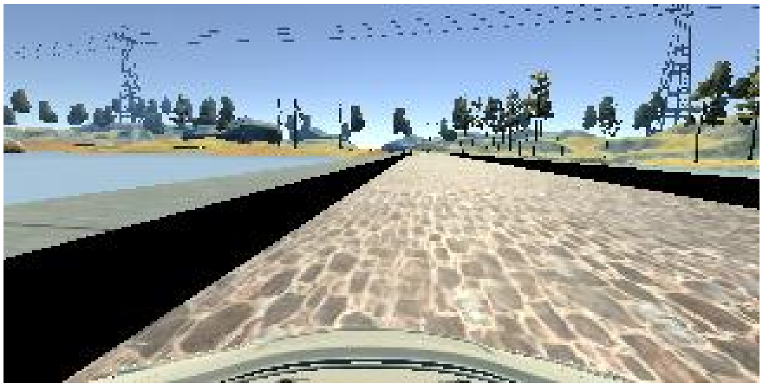

Figure 5: Example Image of Recovery Area on Track

Dataset	Loss
Training	0.035
Validation	0.022

Table 4: Neural Network Model Performance

A couple more things needed to be done to generalize the model enough to avoid overfitting and get the model performance described in Table 4. First, the number of center images in the dataset are 8036. After augmenting the dataset with images with steering angles which are significantly larger or smaller than 0, the total number of images goes upwards of 18000. At the start, the number of steps per epoch were calculated by dividing the total number of images by batch size. This creates a very large number of steps per epoch. That means each epoch is being trained on all the images which can lead to overfitting. To address this issue another hyperparameter is creates as indicated in Table 2 called sample size which is some fraction of the whole dataset. By the use of this parameter, each epoch is trained on number of images indicated by sample size randomly selected from the complete dataset. This ensured that, each epoch is being trained on different set of images and avoids overfitting and also speeds up the process of training.

The second thing which was done to improve the model performance is to multiply the steering measurement by 5 so as to increase loss. Without the multiplication, steering angles are very small and that results in even smaller loss. This idea was taken from one of the papers published online on Behavioral Cloning [3]. The drive.py file is eventually modified and the steering prediction is divided by 5.

Testing CNN Model on Simulator

After getting satisfactory performance on the training and validation dataset, next step is to test the model on the driving simulator provided by Udacity. The simulator has two modes, manual and autonomous. Udacity also provides a python script to load the trained model output the steering angles required to drive the car around the simulator track. The car is allowed to run on the track for two complete laps to test the performance of the model on the simulator. The video of the simulation is saved with the name video.mp4.

Summary and Conclusion

In conclusion of this project, it can be said that the Deep Learning neural network model based adapted from NVDIA End-to-End Deep Learning Neural Network did a good job of cloning the behavior of manual driving on a simulated driving environment. The major takeaway from this project is that the performance of the model depended heavily on the dataset used to train the model. It is found that the model is sensitive to the variation in the dataset provided. Specifically, the model is found to be sensitive to the bias between images corresponding to straight driving versus images corresponding to recovery driving and to the sample size of training images per epoch. Significant effort is made in this project to avoid model overfitting by adding dropouts, sampling training images randomly etc. Some more effort will be required to make this model more robust.

References

[1] E. Santana and G. Hotz, "Learning a driving simulator," *arXiv preprint arXiv:1608.01230*, 2016.

[2] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars. arxiv 2016," *arXiv preprint arXiv:1604.07316*, 2016.

[3] S. Sharma, G. Tewolde, and J. Kwon, "Behavioral cloning for lateral motion control of autonomous vehicles using deep learning," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0228–0233.