Machine Learning - Microsoft

Internship Project Report

Self Driving Cars

Author:
Sahil Chetan Sorte

Date: 20/10/2022

Table of Contents

1 Abstract

The development of self-driving cars is one of the most trendy and popular directions in the world of AI and machine learning. Humanity has been waiting for self-driving cars for several decades. Thanks to the extremely fast evolution of technology, this idea recently went from "possible" to "commercially available in a Tesla". In this project, a simple and naive self-driving car is developed. This project is the implementation of NVIDIA's "End to End Learning for Self-Driving Cars" paper. It is a supervised regression problem between the car steering angles and the road images in real-time from the cameras of a car. The car uses a trained convolutional neural network (CNN) that predicts all the parameters that the car needs to drive smoothly. These are directly connected to the main steering mechanism and the output of the deep learning model determines the steering angle of the vehicle.

2 Objectives

The objective of our project is as follows:
• To train a model in a manner that would drive a car by itself around a track in a driving simulator.
• To train a model that would determine the rotation angle for steering wheel based on the centre camera view of the car.
• To train a model that would adapt to the changes in the texture of the roads and predict angles accordingly.

3 Introduction

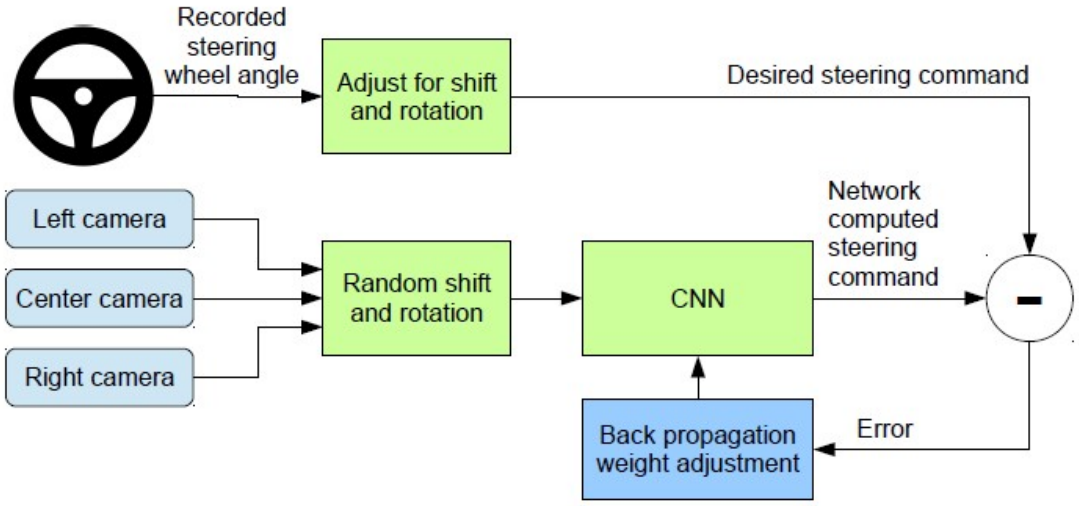
A self-driving car or autonomous car or a driverless car a vehicle that is capable of sensing its environment and navigating without human input. Autonomous cars combine a variety of techniques to perceive their surroundings, using sensors like radar, laser light, GPS and computer vision. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage.

The automation feature of existing cars is insufficient to allow cars to drive itself. There is a constant need for drivers, without it the car is inaccessible. But with self-driving cars, we can make the presence of cars on the road constantly. The Driver constantly needs to monitor signals, road safety signs, barriers, and lanes for traditional cars and make decisions accordingly.

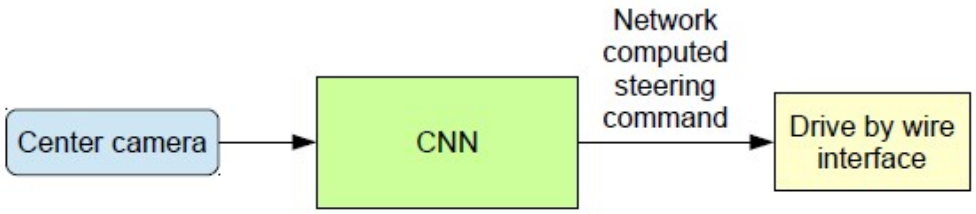
Certainly, autonomous driving can be dangerous to some but it also has its advantages. This would result in reduced traffic congestion, reduced emissions, lower travel costs for all, and a reduction in the cost of new roads and services. It would also immensely improve the mobility of people with old and physical disabilities.

In this project, we are going to learn how to train a self driving car using Convolution neural networks CNN. We will be using the open source Self driving car simulator provided by Udacity that is used in their Self driving car Nano degree program. Using this simulator we will first drive the car and collect data. Then we will train a CNN model to learn this behavior and then test it back on the simulator. The model we will use was proposed by NVIDIA. They used this model to train a real car data and got promising results when they drove it autonomously.

We first get our dataset by running a car in Udacity's Self Driving Car Simulator. The dataset consists of images from the camera and the steering angle. Images are fed into a CNN which then computes a proposed steering command. The proposed command is compared to the desired command for that image and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. A block diagram of our training system as shown below:



Once trained, the network can generate steering from the video images of a single centre camera.



4 Methodology

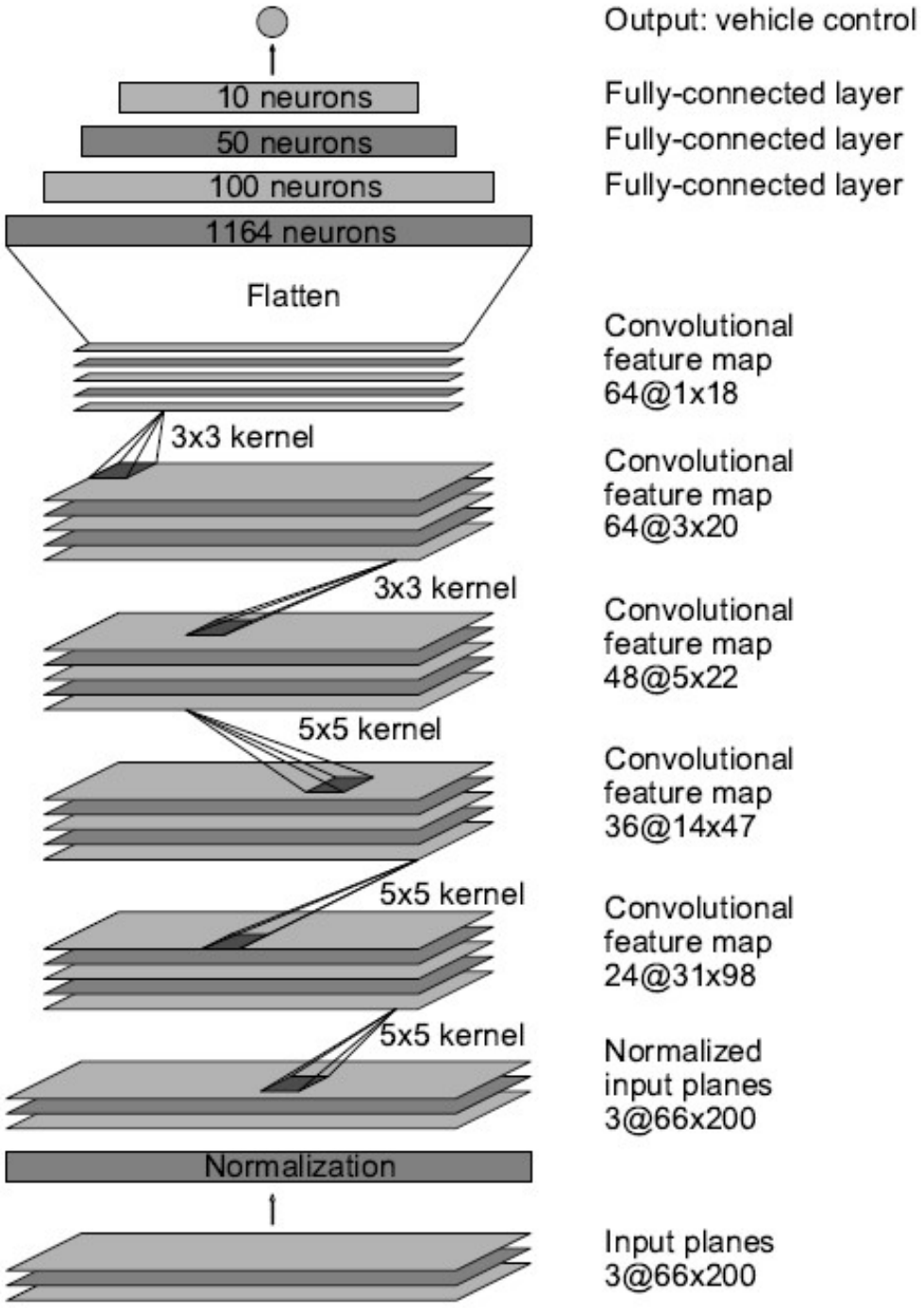
1. Data Collection: Training data was collected by recording the driving in the training mode of Udacity's Self Driving Car Simulator. Drive the car on the same route in 3-4 laps and then reverse the car and drive again. The data collected is stored in the form of images and .csv file. Images are taken from three cameras – center, left and right. The .csv file contains the addresses of three images of an instance while driving. It also includes its corresponding values for steering, throttle, reverse and speed.

2. Data Balancing: Visualize the obtained data by plotting the histogram for distribution of the steering wheel angles. One can observe lots of redundant data. As one can see there is a huge spike near zero which means that most of the times the car is driving straight. Remove this redundant data by eliminating all the data above the decided threshold value.

3. Split for training and validation sets: The next step was to split the data using the 80 20 rule which means using 80% of the data for training while the rest for testing the model on unseen images. In machine learning, data splitting is typically done to avoid overfitting. That is an instance where a machine learning model fits its training data too well and fails to reliably fit additional data.

4. Data Augmentation: After selecting the dataset, we augment the data by adding artificial shifts and rotations to teach the network how to recover from a poor position or orientation. The magnitude of these perturbations is chosen randomly from a normal distribution. The distribution has zero mean, and the standard deviation is twice the standard deviation that we measured with human drivers.

5. Creating the model: The first step to training a neural network is selecting the frames to use. To train a CNN to do lane following we only select data where the driver was staying in a lane and discard the rest. A higher sampling rate would result in including images that are highly similar and thus not provide much useful information. To remove a bias towards driving straight the training data includes a higher proportion of frames that represent road curves. The CNN architecture used is as follows:



6. Training the dataset: Finally, train the model for certain epochs batch-wise. Also plot the loss the training and validation as the function of epochs. If it converges, then it means that it is learning a fairly good policy to steer a car on unseen road environments.

7. Testing: Test the model on the test dataset and get root mean square error (RMSE). Also, run the model in autonomous mode in the simulator.

5 Code

The source code of the project along with the dataset is available here.

First of all, all the libraries and dependencies must be included:

```
import numpy as np
import pandas as pd
import os
from matplotlib import pyplot as plt
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import matplotlib.image as mpimg
import cv2
from imgaug import augmenters as iaa
import random
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Convolution2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam
```

Create an object and load the entire data in driving_log.csv. Also, remove the prefix for the location of images.

```
def getName(filePath):
    return filePath.split('\\')[-1]

def importDataInfo(path):
    columns = ['Centre', 'Left', 'Right', 'Steering', 'Throttle', 'Brake', 'Speed']
    data = pd.read_csv(os.path.join(path, 'driving_log.csv'), names = columns)
    print('Total images imported are: ', data.shape[0])

    data['Centre'] = data['Centre'].apply(getName)
    return data

path = 'SimulationData'
data = importDataInfo(path)
```

Now, plot the distribution for the steering wheel angle and also, balance it out as mentioned in the methodology. After balancing, total remaining images were 1444, and the one which were removed were 5128.

```
def balanceData(data, display=True):
    nBins = 31
    samplesPerBin = 350
```

```
    hist, bins = np.histogram(data['Steering'], nBins)
    #print(hist)
    #print(bins)
    if display:
        centre = (bins[:-1] + bins[1:])*0.5
        #print(centre)
        plt.bar(centre, hist, width = 0.06)
        plt.plot((-1,1), (samplesPerBin, samplesPerBin))
        plt.show()

    removeIndexList = []
    for j in range(nBins):
        binDataList = []
        for i in range(len(data['Steering'])):
            if data['Steering'][i] >= bins[j] and data['Steering'][i] <= bins[j+1]:
                binDataList.append(i)

        binDataList = shuffle(binDataList)
        binDataList = binDataList[samplesPerBin:]
        removeIndexList.extend(binDataList)

    print('Removed Images:' ,len(removeIndexList))
    data.drop(data.index[removeIndexList], inplace=True)
    print('Remaining Images:',len(data) )

    if display:
        hist, _ = np.histogram(data['Steering'], nBins)
        #print(centre)
        plt.bar(centre, hist, width = 0.06)
        plt.plot((-1,1), (samplesPerBin, samplesPerBin))
        plt.show()

    return data

data = balanceData(data, display=True)
```

from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Convolution2D, Flatten, Dense from tensorflow.keras.optimizers import Adam

Then I made a function to load all the images as well as the steering wheel angle values in a numpy array. Also, split the dataset into training and validation set. Training images are 1155 and validation images are 289.

```
def loadData(path, data):
    imagesPath = []
    steering = []
    for i in range(len(data)):
        indexedData = data.iloc[i]
        imagesPath.append(os.path.join(path,'IMG',indexedData[0]))
        steering.append(float(indexedData[3]))

    imagesPath = np.asarray(imagesPath)
    steering = np.asarray(steering)
```

```
    return imagesPath, steering

imagesPath, steering = loadData(path, data)
xTrain, xVal, yTrain, yVal = train_test_split(imagesPath, steering, test_size = 0.2,
print('Total training Images:', len(xTrain))
print('Total validation Images:', len(xVal))
```

As stated in the Nvidia's paper, we augmented the images by either panning or zooming or brightening or flipping or their random combinations. We continued by doing some image processing. Crop the image to remove the unnecessary features, change the images to YUV format, use gaussian blur, decrease the size for easier processing and normalize the values.

```
def augmentImage(imagePath, steering):
    img = mpimg.imread(imagePath)

    ##PAN
    if(np.random.rand()<0.5):
        pan = iaa.Affine(translate_percent={"x":(-0.1, 0.1), "y":(-0.1, 0.1)})
        img = pan.augment_image(img)

    ##ZOOM
    if(np.random.rand()<0.5):
        zoom = iaa.Affine(scale=(1,1.2))
        img = zoom.augment_image(img)

    ##BRIGHTNESS
    if(np.random.rand()<0.5):
        brightness = iaa.Multiply((0.5,1.2))
        img = brightness.augment_image(img)

    ##FLIP
    if(np.random.rand()<0.5):
        img = cv2.flip(img, 1)
        steering = -steering

    return img, steering

def preprocessing(img):
    img = img[60:135,:,:]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    img = cv2.GaussianBlur(img, (3,3), 0)
    img = cv2.resize(img, (200,66))
    img = img/255

    return img

def BatchGenerator(imagesPath, steeringList, batchSize, trainFlag):
    while True:
        imgBatch = []
        steeringBatch = []

        for i in range(batchSize):
            index = random.randint(0, len(imagesPath)-1)
```

```
            if trainFlag:
                img, steering = augmentImage(imagesPath[index], steeringList[index])
            else:
                img = mpimg.imread(imagesPath[index])
                steering = steeringList[index]
            img = preprocessing(img)
            imgBatch.append(img)
            steeringBatch.append(steering)

        yield(np.asarray(imgBatch), np.asarray(steeringBatch))
```

Create the model on the basis of CNN architecture given in the figure in methodologies.

```
def createModel():
    model = Sequential()

    model.add(Convolution2D(24, (5, 5), (2, 2), input_shape=(66, 200, 3), activation=
    model.add(Convolution2D(36, (5, 5), (2, 2), activation='elu'))
    model.add(Convolution2D(48, (5, 5), (2, 2), activation='elu'))
    model.add(Convolution2D(64, (3, 3), activation='elu'))
    model.add(Convolution2D(64, (3, 3), activation='elu'))

    model.add(Flatten())

    model.add(Dense(100, activation = 'elu'))
    model.add(Dense(50, activation = 'elu'))
    model.add(Dense(10, activation = 'elu'))
    model.add(Dense(1))

    model.compile(Adam(lr=0.0001),loss='mse')
    return model

model = createModel()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 31, 98, 24)        1824

 conv2d_1 (Conv2D)           (None, 14, 47, 36)        21636

 conv2d_2 (Conv2D)           (None, 5, 22, 48)         43248

 conv2d_3 (Conv2D)           (None, 3, 20, 64)         27712

 conv2d_4 (Conv2D)           (None, 1, 18, 64)         36928

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 100)               115300

 dense_1 (Dense)             (None, 50)                5050

 dense_2 (Dense)             (None, 10)                510

 dense_3 (Dense)             (None, 1)                 11

=================================================================
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
_________________________________________________________________
```

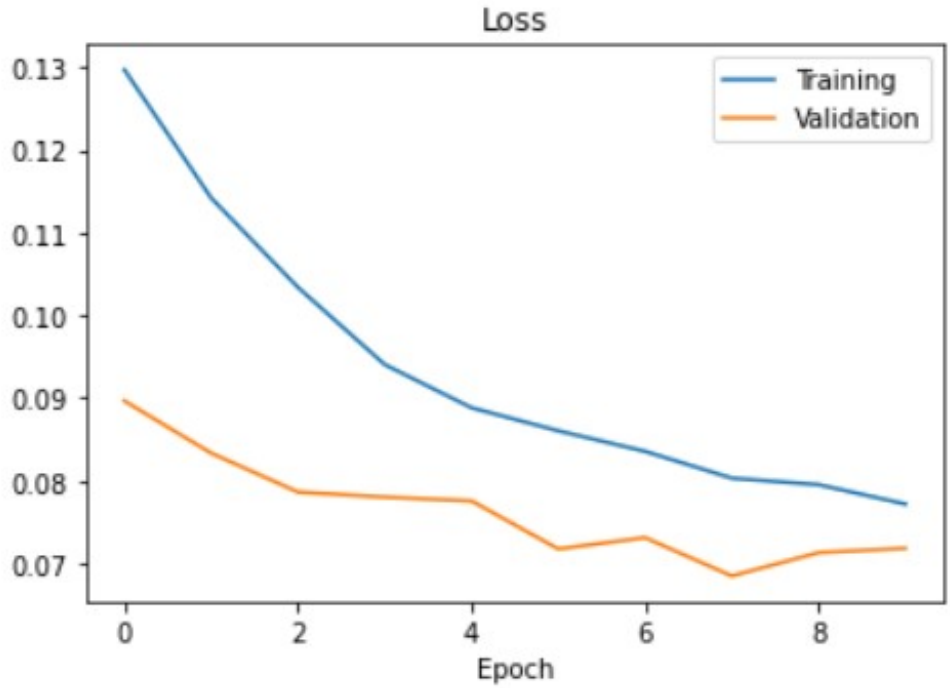
Finally, I trained the model for 10 epochs with a batch size of 100. Also, I plotted the training and the validation loss as a function of epochs.

```
history = model.fit(BatchGenerator(xTrain, yTrain, 100, 1), steps_per_epoch=300, epoch

model.save('model.h5')
print('Model Saved')

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['Training', 'Validation'])
plt.title('Loss')
plt.xlabel('Epoch')
plt.show()
```

```
Epoch 1/10
300/300 [==============================] - 214s 714ms/step - loss: 0.1297 - val_loss: 0.0897
Epoch 2/10
300/300 [==============================] - 213s 710ms/step - loss: 0.1142 - val_loss: 0.0834
Epoch 3/10
300/300 [==============================] - 212s 709ms/step - loss: 0.1034 - val_loss: 0.0787
Epoch 4/10
300/300 [==============================] - 218s 727ms/step - loss: 0.0941 - val_loss: 0.0780
Epoch 5/10
300/300 [==============================] - 216s 723ms/step - loss: 0.0888 - val_loss: 0.0776
Epoch 6/10
300/300 [==============================] - 216s 722ms/step - loss: 0.0861 - val_loss: 0.0718
Epoch 7/10
300/300 [==============================] - 215s 717ms/step - loss: 0.0836 - val_loss: 0.0732
Epoch 8/10
300/300 [==============================] - 214s 716ms/step - loss: 0.0803 - val_loss: 0.0685
Epoch 9/10
300/300 [==============================] - 213s 711ms/step - loss: 0.0795 - val_loss: 0.0713
Epoch 10/10
300/300 [==============================] - 216s 720ms/step - loss: 0.0772 - val_loss: 0.0719
```



6 Conclusion

On testing, the RMSE value turned out to be 0.24 approximately, which is the indicator of the fact that our model is trained properly. Also, both the training and validation losses converges after 6 epochs. It is really interesting to know that a small amount of training data was sufficient to train the car. The CNN is able to learn meaningful road features from a very sparse training signal (steering alone). This system is work perfect in the simulated environment, but becomes difficult when the tracks are changed. Also, it doesn't know the recovery methods, i.e., what if it hits the barrier. It needs to be trained for all these possibilities. However, it was really interesting to work on the project.