**AUTONOMOUS CAR FOLLOWING THE LANE**

GETTING THE DATA FROM THE CAR SIMULATOR (IMAGES, THROTTLE, SPEED VALUES ETC.,)

PREPROCESS THE IMAGE

CREATE THE NEURAL NETWORK MODEL

TRAIN THE MODEL

TESTING THE SIMULATOR

SPEED, THROTTLE, STEERING VALUES HAS SENT TO THE SIMULATOR VIA THE SOCKET, EVENTLET PORT

AUTOMONOUS MOTION OF THE CAR USING THOSE VALUES

Code

**UTILS.PY**

The utils.py file has all the necessary functions required for the project to use while training and testing the data/simulator.

import pandas as pd  
import numpy as np  
import os  
import matplotlib.pyplot as plt  
from sklearn.utils import shuffle  
import matplotlib.image as mpimg  
from imgaug import augmenters as iaa  
import cv2  
import random  
  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Convolution2D, Flatten, Dense  
from tensorflow.keras.optimizers import Adam  
  
*# we split the path*def getName(filepath):  
  
 return filepath.split(**'**\\**'**)[-1]  
  
*# we read the csv file using the pandas and take only the image and columns features mentioned below*def importDataInfo(path):  
 coloums = [**'Center'**, **'Left'**, **'Right'**, **'Steering'**, **'Throttle'**, **'Brake'**, **'Speed'**]  
 print(os.path.join(path, **'driving\_log.csv'**))  
 data = pd.read\_csv(os.path.join(path, **'driving\_log.csv'**), names=coloums)  
 *#print(data.head())  
 # print(data['Center'][0])  
 #print(getName(data['Center'][0]))* data[**'Center'**] = data[**'Center'**].apply(getName)  
 *# print(data.head())* print(**'Total Images Imported:'**, data.shape[0])  
 return data

*# we balance the steering data and remove the data to a particular samples value*def balanceData(data, display=True):  
 nBins = 31  
 samplesPerBin = 1000  
 hist, bins = np.histogram(data[**'Steering'**], nBins)  
 *# print(bins)* if display:  
 center = (bins[:-1] + bins[1:]) \* 0.5  
 *# print(center)* plt.bar(center, hist, width=0.06)  
 plt.plot((np.min(data[**'Steering'**]), np.max(data[**'Steering'**])), (samplesPerBin, samplesPerBin))  
 plt.show()  
  
 removeindexList = []  
  
 for j in range(nBins): *# we check the steering data with respective bins and remove them* 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))  
 plt.bar(center, hist, width=0.06)  
 plt.plot((-1, 1), (samplesPerBin, samplesPerBin))  
 plt.show()  
  
 return data  
  
*# we return the images path and steering values from the data*def loadData(path, data):  
 imagesPath = []  
 steering = []  
  
 for i in range(len(data)):  
 indexedData = data.iloc[i]  
 *# print(indexData)* 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

*# Here we pan or zoom or brightness or flip the image with the car in it*def augmentImage(imgPath, steering):  
 img = mpimg.imread(imgPath) *# Read the image* if np.random.rand() < 0.5:  
 pan = iaa.Affine(translate\_percent={**"x"**: (-0.1, 0.1), **"y"**: (-0.1, 0.1)}) *# Pan the image* img = pan.augment\_image(img)  
 if np.random.rand() < 0.5: *# Zoom the image* zoom = iaa.Affine(scale=(1, 1.2))  
 img = zoom.augment\_image(img)  
 if np.random.rand() < 0.5: *# increase the brightness* brightness = iaa.Multiply((0.4, 1.2))  
 img = brightness.augment\_image(img)  
 if np.random.rand() < 0.5: *# Flip the image* img = cv2.flip(img, 1)  
 steering = -steering  
 return img, steering

*# convert the image to YUV)(ULtraviolet image changes with change in Y values),blur and resize the image*def preProcess(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  
  
*# This function generates the batches for training*def batchGen(imagesPath, steeringList, batchSize, trainFlag):  
 while True:  
 imgBatch = []  
 steeringBatch = []  
  
 for i in range(batchSize):  
 index = random.randint(0, len(imagesPath) - 1) *# getting the random number* if trainFlag:  
 img, steering = augmentImage(imagesPath[index], steeringList[index])  
 else:  
 img = mpimg.imread(imagesPath[index])  
 steering = steeringList[index]  
 img = preProcess(img)  
 imgBatch.append(img)  
 steeringBatch.append(steering)  
 yield (np.asarray(imgBatch), np.asarray(steeringBatch))

*# create the model using the keras sequential model with convolutions*def createModel():  
 model = Sequential()  
 *# check about the convolutions neural network to understand more For parameters link:https://keras.io/api/layers/convolution\_layers/convolution2d/* model.add(Convolution2D(24, (5, 5), (2, 2), input\_shape=(66, 200, 3), activation=**'elu'**)) *# input shape is the size of the image* 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'**) *# with changing the learning rate the model  
 # performance can vary basically the learning rate is how fast the model can learn about the features in data* return model

**TRAININGSIMULATION.PY**

The training simulation.py file has the steps required to train the model.

print(**'Setting UP'**)  
import os  
os.environ[**'TF\_CPP\_MIN\_LOG\_LEVEL'**] = **'3'**from utlis import \*  
from sklearn.model\_selection import train\_test\_split  
  
  
*#### STEP 1  
# the path of the data where it is stored after running the car simulator*path = **"D:**\\**Simulator**\\**beta\_simulator\_windows**\\**mydata"***# we get the data from the specified path*data = importDataInfo(path)  
print(data)  
  
*#### STEP 2  
# we balance the data*data = balanceData(data,display=True)  
  
*#### STEP 3  
# we get the image path and steering value which is basically the direction*imagesPath, steerings = loadData(path,data)  
  
*#### STEP 4  
# Splitting the data to train and validation sets with test\_size = 20 percent of the data*xTrain, xVal, yTrain, yVal = train\_test\_split(imagesPath, steerings, test\_size=0.2,random\_state=5)  
print(**'Total Training Images: '**,len(xTrain))  
print(**'Total Validation Images: '**,len(xVal))  
  
*####STEP 5  
  
####STEP 6  
  
####STEP 7  
  
####STEP 8  
# create the model*model = createModel()  
model.summary()  
  
*#### STEP 9  
# we train the data for 10 epochs by changing the epochs the performance can vary*history = model.fit(batchGen(xTrain,yTrain,100,1),steps\_per\_epoch=300,epochs=10,validation\_data=batchGen(xVal,yVal,100,0),validation\_steps=200)  
  
*#### STEP 10  
# saving the model to an h5 format*model.save(**'model\_new1.h5'**)  
print(**'Model Saved'**)  
*# plotting the train loss and validation loss*plt.plot(history.history[**'loss'**])  
plt.plot(history.history[**'val\_loss'**])  
plt.legend([**'Training'**,**'validation'**])  
plt.ylim([0,1])  
plt.title(**'Loss'**)  
plt.xlabel(**'Epoch'**)  
plt.show()

**Testsimulation.py**

This file has the code for the preprocessing the image, telemetry calculations and sending those values such as throttle, speed, steering values to drive the car.

print(**'Setting UP'**)  
import os  
os.environ[**'TF\_CPP\_MIN\_LOG\_LEVEL'**] = 3 *# to hide any runtime warnings given by tensorflow*import socketio  
import eventlet  
import numpy as np  
from flask import Flask  
from tensorflow.keras.models import load\_model  
import base64  
from io import BytesIO  
from PIL import Image  
import cv2  
   
*#### FOR REAL TIME COMMUNICATION BETWEEN CLIENT AND SERVER*sio = socketio.Server()  
*#### FLASK IS A MICRO WEB FRAMEWORK WRITTEN IN PYTHON*app = Flask(\_\_name\_\_) *# '\_\_main\_\_'*maxSpeed = 10

*# we convert the image to YUV , blur and resize the image*def preProcess(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  
   
  
*# getting the telemetry values such as steering values, speed, throttle*@sio.on(**'telemetry'**)  
def telemetry(sid, data):  
 print(**"telemetry ON"**)  
 speed = float(data[**'speed'**])  
 image = Image.open(BytesIO(base64.b64decode(data[**'image'**]))) *# decoding the image from bytes* image = np.asarray(image)  
 image = preProcess(image)  
 image = np.array([image])  
 steering = float(model.predict(image)) *# taking the preicted steering values from the image* throttle = 1.0 - speed / maxSpeed  
 print(**'{} {} {}'**.format(steering, throttle, speed))  
 sendControl(steering, throttle)  
   
  
@sio.on(**'connect'**)  
def connect(sid, environ):  
 print(**'Connected'**)  
 sendControl(0, 0)  
   
*# sending the steering and throttle values to car by server*def sendControl(steering, throttle):  
 sio.emit(**'steer'**, data={  
 **'steering\_angle'**: steering.\_\_str\_\_(),  
 **'throttle'**: throttle.\_\_str\_\_()  
 })  
   
   
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 model = load\_model(**'model\_new1.h5'**)  
 app = socketio.Middleware(sio, app)  
 *### LISTEN TO PORT 4567* eventlet.wsgi.server(eventlet.listen((**''**, 4567)), app)

**NOTE:**

***Socket.IO***:

This library is a transport protocol that enables real-time bidirectional event-based communication between clients and a server. If you are using the Python client and server, the easiest way to ensure compatibility is to use the same version of this package for the client and the server. If you are using this package with a different client or server, then you must ensure the versions are compatible.  the versions of the Socket.IO and Engine.IO protocols must be of same version.

**Eventlet**:

Eventlet is a concurrent networking library for Python that allows you to change how you run your code, not how you write it. It uses epoll or libevent for highly scalable non-blocking i/o. coroutines ensure that the developer uses a blocking style of programming that is similar to threading, but provide the benefits of non-blocking i/o. the event dispatch is implicit, which means you can easily use eventlet from the python interpreter, or as a small part of a larger application.

**Imgaug**:

imgaug is a library for image augmentation in machine learning experiments. it supports a wide range of augmentation techniques, allows to easily combine these and to execute them in random order or on multiple cpu cores, has a simple yet powerful stochastic interface and can not only augment images, but also keypoints/landmarks, bounding boxes, heatmaps and segmentation maps.