**Data.**

The data for this project was acquired by driving a car in Udacity simulation engine. Udacity (<https://github.com/udacity/self-driving-car-sim>) and Microsoft (<https://github.com/Microsoft/AirSim>) provide the most famous simulation engines for training and testing an autonomous vehicle algorithm. The reason for choosing the Udacity engine is simply less training time due to comprising the graphics quality of the images. These engines offer the ability to be connected to an autonomous vehicle code through a local server using …. For which the code is provided in the appendix.

The vehicle was driven for three laps in both direction on the track. The reason being, the lap in the forward direction mostly consist of left turns, so in order to avoid bias in our data, the car was trained in the converse direction as well. It is important to include various possibilities and situations which the vehicle might encounter in the real life situation in the training phase. For example, we don’t want to drive the car without any disturbance on a straight line at the center line of the road, instead, the vehicle should be swerved from left to right on the track as well as driven on different locations on the track in order for the algorithm to be able to generalize when faced such situations. One other considerable occasion would be when the vehicle wanders off the track. This situation can be overcome by driving the car from an off-track location to a stable situation on the track on multiple occasions.

Through the whole process, three images are recorded from the cameras at three different angles namely, center, left and right per each frame. The cameras record the training process with 15+ Frame/sec pace so the final system on which this model is to be mounted should be capable of processing images with such rate.

In addition to the images, the steering angle, which is a floating point value between -1 and 1 is recorded corresponding to -180 and 180 degrees respectively, which would be the label for our data. Moreover, the throttle value (a Boolean indicating whether the vehicle is throttled or not), the reverse value (a Boolean indicating whether the vehicle is moving in the reverse direction or not) and the speed of the vehicle (a floating point value in mile/hour) is recorded as well. The following figure represents the first four instances of data as the head of a pandas table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| center | left \ | right \ | steering | throttle | reverse | speed |
| center\_2019\_02\_18\_14\_33\_39\_945.jpg | left\_2019\_02\_18\_14\_33\_39\_945.jpg | right\_2019\_02\_18\_14\_33\_39\_945.jpg | 0.0 | 0.0 | 0.0 | 0.451 |
| center\_2019\_02\_18\_14\_33\_40\_017.jpg | left\_2019\_02\_18\_14\_33\_40\_017.jpg | right\_2019\_02\_18\_14\_33\_40\_017.jpg | 0.0 | 0.0 | 0.0 | 0.447 |
| center\_2019\_02\_18\_14\_33\_40\_090.jpg | left\_2019\_02\_18\_14\_33\_40\_090.jpg | right\_2019\_02\_18\_14\_33\_40\_090.jpg | 0.0 | 0.0 | 0.0 | 0.444 |
| center\_2019\_02\_18\_14\_33\_40\_160.jpg | left\_2019\_02\_18\_14\_33\_40\_160.jpg | right\_2019\_02\_18\_14\_33\_40\_160.jpg | 0.0 | 0.0 | 0.0 | 0.438 |

Table 1) First four instance of the data

The whole data set is available is uploaded on github at: <https://github.com/CarlMj/SelfDriving.git>

For the sake of simplicity, and given the fact that a there will not be a need for too many data to use in the already time consuming phase of training, only the images from the central camera are used. Furthermore, we only aim for predicting the steering angle as the labels of our instances and will leave the vehicle to drive at a constant pre-specified speed. So far, there are 7458 instances in our pool. This data set is split into a batches of training and test sets with the ratio of 0.2 for the test set.

As it is the pivotal part of any data science or machine learning project, visualizing the data in order to grasp a comprehensive view of the problem is very important. The following shows the histogram distribution of our data base on the steering angles.

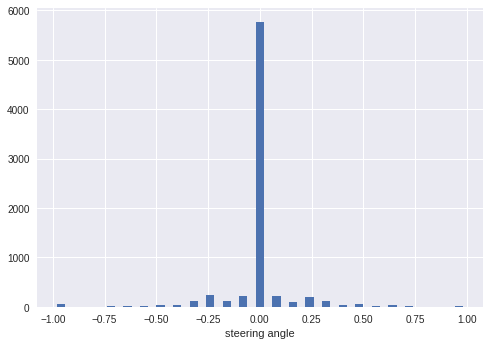


Figure 1)Distribution of steering angel in the raw data

As it is evident, the raw data is blatantly biased towards the 0.0 steering angel. This can be attributed to the fact that most paths on which a vehicle is driven is mostly consisted of straight tracks and not so many turns. A learning algorithm trained on this data will suffer from a very high probability to predict a 0.0 steering angle for any instance of image it faces.

This issue is taken care of in the preprocessing step.

**Neural Network (Nvidia Model).**

We train our vehicle using the Nvidia neural network model. This vetted model has proven to be highly effective for autonomous driving applications as well as other image processing applications.

The model consists of 10 layers, starting from the convolutional layers to the final output layer which will be the predicted angel.

Convolutional layers.

There are five convolutional layers, the first one of which uses a kernel of size 5\*5 to convolute over the image with strides of (2 , 2), meaning a step of size 2 in the horizontal direction as well as the vertical direction. This first layer trains 24 different filters (kernels).

The second and third convolutional layers have basically the same setting other than the fact that there are 36, 48 filters to be trained in each one respectively. The fourth and the last one contain 64 filters of size 3\*3, each of which traversing the image with a strides of size (3, 3) since by these layers the array would have gotten relatively small to be traversed with the same strides. The activation function used in all these convolutional functions is relu at first (instead of sigmoid to avoid the vanishing gradient problem).

Fully Connected layers.

The convolutional layers are followed by the fully connected layers. There exists a flatten layer to translate the output of convolutional layers which is a three dimensional array, to a vector that can be processed by the dense layers. These fully connected layers contain 100, 50, 10 and 1 units respectively.

Compiling the network.

The network is set up with the Adam optimization function as the optimizer to minimize the loss function for which we choose minimum squared error as is conventional used in regression problems. The summary table for this network is provided in the appendix.

This model is run for 10 epochs with batches of size 100. The learning rate is set to 0.01. The following figure depicts the value of loss function through the training process.

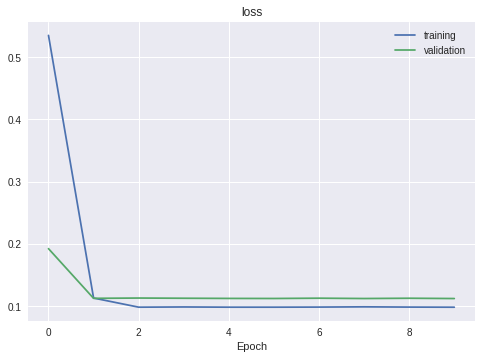


Figure 2) Loss values for relu function

As anticipated, this first network does not yield the desired results. As it is evident, the validation loss function plateaus after a short while and which simply means the network does not learn anything after the third epoch. Both the advantage and the downside of neural networks is the fact that there are many hyper parameters to be tweaked. For the above figure it seems that we are facing the dead nodes problem which is attributed to the activation function, relu. This function activates the nodes based on the behavior shown in the following figure.

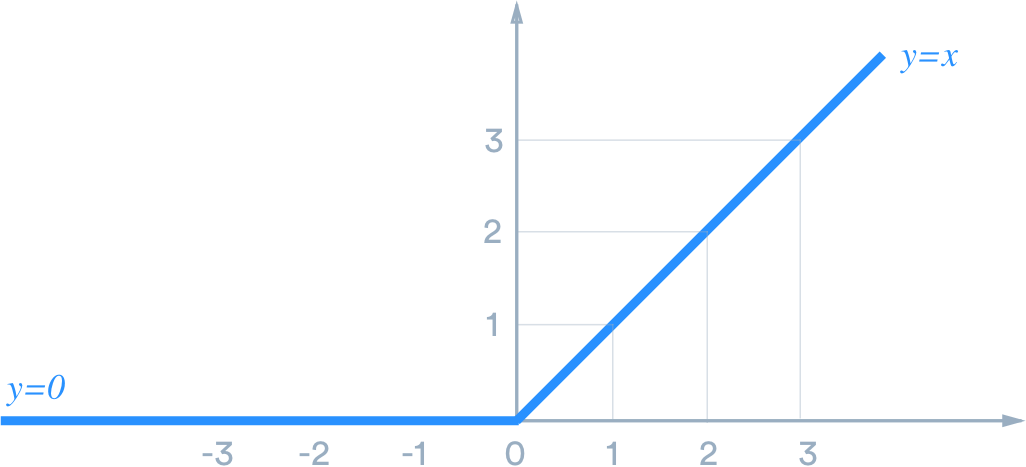


Figure 3) Relu activation function

If the input value of this function becomes negative, the output will simply be zero, resulting in a dying unit. If considerable number of receive a negative value as input, this phenomenon will in effect turn a large portion of the network off. The backpropagation will not be able to change the value the node and thus the network stops leaning. We can simply overcome this issue by changing the relu function to elu activation function. As can be seen in the following figure, the value on the negative side of the elu function is not zero, precluding the dead nodes.

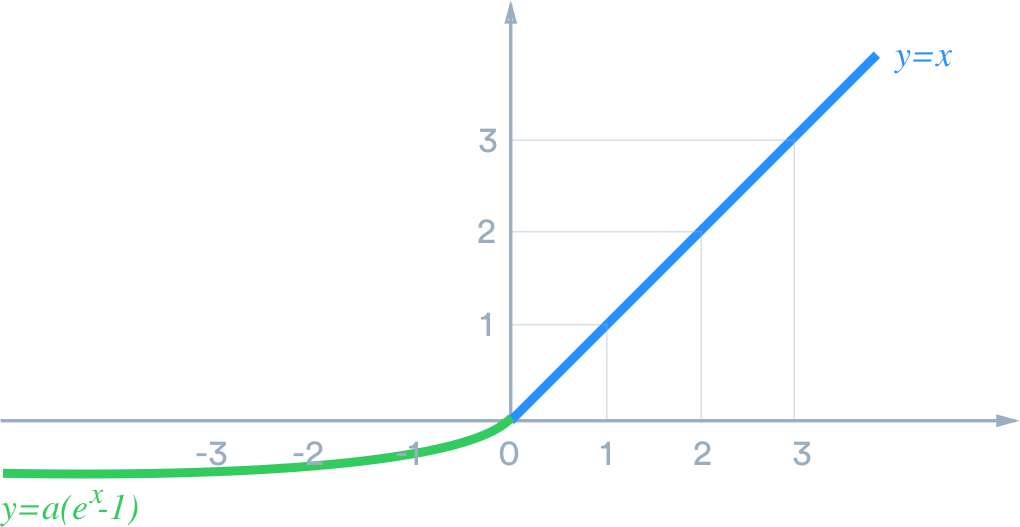


Figure 4) elu activation function

The next attempt to train the network after the new choice of activation function results in a more sensible loss value. However, when launched on the test mode for the Udacity engine, the model wandered off the track after a short while. Given the fact that we have chosen the Nvidia model as our network, our guess was the lack of data is the culprit this time. The strength of the Nvidia model will be more palpable when a large pool of images is fed through.

Data augmentation.

Data augmentation is a technique generally used to resolve the issue of lack of data. These techniques are proven to be very helpful especially when the cost of acquiring more data is very high or when simply no more data may be available. These techniques are context specific, but are intuitive enough to be easily make sense. The idea is to use the currently available data to generate new batches.

As an example, consider image classification. You can imagine rotating a given image by a small value would not result in a considerable change to the essence of the problem. The label for that image will remain the same as well as the values for each pixel. However, the rotated image is a new data by which the model can learn better. Similar techniques of this form are introduced to the data in order to increase the size of our data set.

Having an augmented data set came at a price and that is, the training set and the test set was now too big to be fed to the network at once. It is worth mentioning that the google.colab notebook environment is used to train the model for the free GPU available in this environment. The RAM in this environment, and even many computers, cannot work with such a large amount of data. This problem was obviated the by building a batch generator, which would divide the data in small batches, preprocess each batch and then feed it to the network to be learned.

Note that we so far we are equipped with a data augmentation routines, and a batch generator which can provide the infinite amount of data using the augmentations. Therefore, what needs to be determined as the hyper parameters of the training will be the batch size, the new parameter “steps per epoch” which indicates the number of batches to be generated per each epoch, as well as the number of epochs. The batch size and the steps per epoch for training and validation data are set to 100, 300 and 200 respectively. .The resulting figure after 10 epochs is as follows:

…………………………………………………………………………………………………

The problem here is the overfitting. When there are too many parameters for the network to exploit to adjust itself to the data, the network becomes too dependent on the training data such that it prevents it from generalizing to new data reasonably .As one solution, we added dropout layers to our network to reduce the number of learnable parameters at random. The function of a dropout layer with a dropout ration is to turn the output of portion of the preceding layer off at random, thus excluding those nodes from the learning cycle in effect. After adding a dropout layer after each fully connected layer with ratios 0.5 for each the loss function looks as follows.

………………………………………………………………………………..

Another critical hyper parameter inducing drastic changes in the result of backpropagation is the learning rate of our optimizer. From the following equation,

A very small value for the learning rate would cause slow learning and need for more data while a large value would cause the optimizer to leap out of the desired region in which the optimum value exists (figure i).

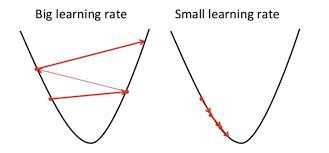


Figure 5) effect of learning rate

Currently as our learning process is not desirably stable and the model is not learning constantly, we can try the model we smaller values for learning rate. The following figure is correspoinding to the learning rate = 0.001.

From this point on, we tried to adjust the hyper parameters the learning rate, the number of epochs and the dropout ratios of the layers. Our final model which yielded the best results has the following properties (the summary table for this models is provided in the appendix).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Learning\_rate | Epochs | Train\_batch | Validation\_batch | Dropout1 | Dropout2 | Dropout3 |
| 0.0001 | 33 | 100\*300 | 100\*200 | 0.0 | 0.0 | 0.2 |

This model achieved the following acceptable loss values:

Training Loss = 0.0371

Validation Loss = 0.0362

**Conclusion.**

This project was an attempt to build a dummy autonomous drive car. It is very important to analyze the data prior to anything and prevent potentially untraceable flaws in the model. It is almost never the case that the raw data would be perfect regardless of how carefully it is obtained. Furthermore, it helps to have a general idea of the mission of the neural network in order to tinker with the hyper parameters accordingly.

To improve this autonomous driver to generalize even better, we can include the images from left and right cameras as well. We may also crop the images better to mostly focus on the track itself and not the unimportant views from all around the car. We can integrate a sign detector to the model as well to learn the labels of the signs on the road and make decision for the speed of the vehicle based on that.

Please consult the appendix for the codes.

**Appendix**

Code for the machine learning model.

First the necessary libraries are imported

import random

import numpy as np

import matplotlib.pyplot as plt

import keras

import matplotlib.image as mpimg

import cv2

import pandas as pd

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Convolution2D, MaxPooling2D, Dropout, Flatten

from keras.optimizers import Adam

from imgaug import augmenters as ia

import os, ntpath

Load the data with the desired names for the columns. Since we will work with the center images manipulate only center’s images addresses to be work with them more easily in future.

data = pd.read\_csv("SelfDriving/driving\_log.csv", names = ["center", "left", "right", "steering", "throttle", "reverse", "speed"])

print(data.head())

data['center'] = data['center'].apply(lambda x: ntpath.basename(x))

print(data.head())

Draw a histogram of the images in each steering angel bin.

hist, bins = np.histogram(data['steering'], 25)

centered\_bins = (bins[:-1] + bins[1:]) / 2.0

plt.bar(centered\_bins, hist, width = 0.04)

plt.xlabel("steering angle")

threshold = 300

plt.plot((-1, 1), (threshold, threshold))

plt.show()

Remove the a number of images beyond the desired threshold at random

from sklearn.utils import shuffle

to\_be\_removed = []

for i in range(25):

lst = []

for j in range(len(data['steering'])):

if data['steering'][j] <= bins[i+1] and data['steering'][j] >= bins[i]:

lst.append(j)

lst = shuffle(lst)

to\_be\_removed.extend(lst[threshold:])

Draw a histogram of images after the bins are curtailed to the threshold.

data.drop(data.index[to\_be\_removed], inplace=True)

hist, b = np.histogram(data['steering'], 25)

centered\_bins = (b[:-1] + b[1:]) / 2.0

plt.bar(centered\_bins, hist, width = 0.04)

threshold = 300

plt.plot((-1, 1), (threshold, threshold))

plt.show()

Split the data into train and test sets

from sklearn.model\_selection import train\_test\_split

imagepath = np.array(data['center'])

steer = np.array(data['steering'])

X\_train, X\_test, y\_train, y\_test = train\_test\_split(imagepath, steer, test\_size = 0.2, random\_state = 1)

print(X\_test.shape, X\_test[0])

Define the augmenting functions

def zoom(image):

zoom = ia.Affine(scale=(1, 1.3))

image = zoom.augment\_image(image)

return image

def pan(image):

pan = ia.Affine(translate\_percent={"x": (-0.1, 0.1), "y":(-0.1, 0.1)})

image = pan.augment\_image(image)

return image

def darken(image):

darken = ia.Multiply((0.2, 1.2))

image = darken.augment\_image(image)

return image

def flip(image, steering):

image = cv2.flip(image, 1)

steering = -1\*steering

return image, steering

With the help of the augmenting functions, build an augmenter which will apply randomly some (in some cases may never apply any) of those function to each image received as input.

def augmentor(image, steering):

image = mpimg.imread(os.path.join('SelfDriving', 'IMG', image))

if np.random.rand() > 0.5:

image = zoom(image)

if np.random.rand() > 0.5:

image = pan(image)

if np.random.rand() > 0.5:

image = darken(image)

if np.random.rand() > 0.5:

image, steering = flip(image, steering)

return image, steering

The preprocessor is defined to modify each augmented (or maybe even not augmented) instance to best suit the Nvidia specifications.

def preprocess(image):

image = image[60:137, :, :]

image = cv2.cvtColor(image, cv2.COLOR\_RGB2YUV)

image = cv2.GaussianBlur(image, (3, 3), 0)

image = cv2.resize(image, (200, 66))

image = image / 255

return image

The following is a batch generator, using the augmenter and the preprocessor would generate batches of data to be fed to the network. Note that the while loop is indefinite as we are able to produce as much images as we want to train or network on. This number is determined when calling the Nvidia model to fit the data.

def batch\_generator(images, steerings, batch\_size, train):

while True:

imagebatch = []

steerbatch = []

indices = np.random.randint(len(images), size=batch\_size)

if train:

maps = list(map(augmentor, images[indices], steerings[indices]))

imagebatch, steerbatch = zip(\*maps)

else:

imagebatch = list(map(mpimg.imread, 'SelfDriving/' + 'IMG/' + images[indices]))

steerbatch = steerings[indices]

imagebatch = list(map(preprocess, imagebatch))

yield (np.asarray(imagebatch), np.asarray(steerbatch))

Define the Nvidia network (adding the convolutional layers and the dense layers and etc.)

def nvidia():

model = Sequential()

model.add(Convolution2D(24, (5, 5), subsample=(2, 2), input\_shape=(66, 200, 3), activation='elu'))

model.add(Convolution2D(36, (5, 5), subsample=(2, 2), activation='elu'))

model.add(Convolution2D(48, (5, 5), subsample=(2, 2), activation='elu'))

model.add(Convolution2D(64, (3, 3), activation='elu'))

model.add(Convolution2D(64, (3, 3), activation='elu'))

model.add(Dropout(0.0))

model.add(Flatten())

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

model.add(Dropout(0.0))

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

model.add(Dropout(0.0))

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

model.add(Dropout(0.2))

model.add(Dense(1))

model.compile(optimizer=Adam(0.001), loss='mse')

return model

Fit the model to the data with desired hyper parameters as defined below.

model = nvidia()

print(model.summary())

history = model.fit\_generator(batch\_generator(X\_train, y\_train, batch\_size=100, train=1), steps\_per\_epoch=300, epochs=10, validation\_data = batch\_generator(X\_test, y\_test, batch\_size=100, train=0), validation\_steps=200, verbose=1, shuffle=1)

Plot the training and validation loss.

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.legend(['training', 'validation'])

plt.title('loss')

plt.xlabel('Epoch')

Save this model and download it to be used in the code for the local server.

model.save('NvidiaModel.h5')

from google.colab import files

files.download('NvidiaModel.h5')

Code for the local server.

This local server connects the learned model to the local simulator to predict the steering angels while driving.

import socketio

import numpy as np

from flask import Flask

import eventlet

from keras.models import load\_model

import base64

from io import BytesIO

from PIL import Image

import cv2

sio = socketio.Server()

app = Flask(\_\_name\_\_)

speed\_limit = 10

def preprocess(image):

image = image[60:137, :, :] #first we cut the crap from our image. Like color of the sky is not important for us.

image = cv2.cvtColor(image, cv2.COLOR\_RGB2YUV) #We change the color channel since Nvidia model that will be implemented will work best with this channel

image = cv2.GaussianBlur(image, (3, 3), 0) #reduce the noise and smooth out the image

image = cv2.resize(image, (200, 66)) #this is not necassary but Nvidia model has been trained using this size of images so will probably perform better on this size

image = image / 255

return image

@sio.on('telemetry')

def telemetry(sid, data):

speed = float(data['speed'])

image = Image.open(BytesIO(base64.b64decode(data['image'])))

image = np.array(image)

image = preprocess(image)

image = np.array([image])

steer = float(model.predict(image))

throttle = 1.0 - speed/speed\_limit

send\_steer(steer, throttle)

@sio.on('connect')

def connect(sid, environ):

print('Connected')

send\_steer(0, 1)

def send\_steer(angle, throttle):

sio.emit('steer', data={

'steering\_angle': angle.\_\_str\_\_(),

'throttle': throttle.\_\_str\_\_()

})

if \_\_name\_\_ == '\_\_main\_\_':

model = load\_model('NvidiaModel3.h5')

app = socketio.Middleware(sio, app)

eventlet.wsgi.server(eventlet.listen(('', 4567)), app)