

Distracted Driver Detection

Introduction

Road Accidents are a cause of major concern in today's age. According to a report by the United Nations,

- *Road traffic injuries are the leading cause of death for children and young adults aged 5-29 years.*
- *Approximately 1.3 million people die each year as a result of road traffic crashes.*
- *Road traffic crashes cost most countries 3% of their gross domestic product.*
- *Between 20 and 50 million more people suffer non-fatal injuries, with many incurring a disability as a result of their injury.*

In India itself, during the year 2020, the total recorded 3,66,138 road accidents caused loss of 1,31,714 persons lives and injured 3,48,279 persons. Unfortunately, the worst affected age group in Road accidents is 18-45 years, which accounts for about 70% of total accidental deaths.

Distracted Driving is one of the leading causes of Road Accidents globally. There are many types of distractions that can lead to impaired driving. The distraction caused by mobile phones is a growing concern for road safety.

- *Drivers using mobile phones are approximately 4 times more likely to be involved in a crash than drivers not using a mobile phone. Using a phone while driving slows reaction times (notably braking reaction time, but also reaction to traffic signals), and makes it difficult to keep in the correct lane, and to keep the correct following distances.*
- *Hands-free phones are not much safer than hand-held phone sets, and texting considerably increases the risk of a crash.*

Objectives

We are planning to create a model that, when feeded with an image from the dashcam, can accurately identify if a driver is focussed on driving or is distracted based on a metric of 10 identifiers.

The 10 classes to predict are:

c0: safe driving

c1: texting - right

c2: talking on the phone - right

c3: texting - left

c4: talking on the phone - left
c5: operating the radio
c6: drinking
c7: reaching behind
c8: hair and makeup
c9: talking to passenger

Our objective is to use Computer Vision / Deep Learning to identify distracted driving real time. The model in reference should pass two quality checks, it should have high accuracy and minimum log loss.

Log loss metric will be used to test our models correctness because unlike other metrics, it highly penalises overconfidence in incorrect predictions.

Software / Hardware Used

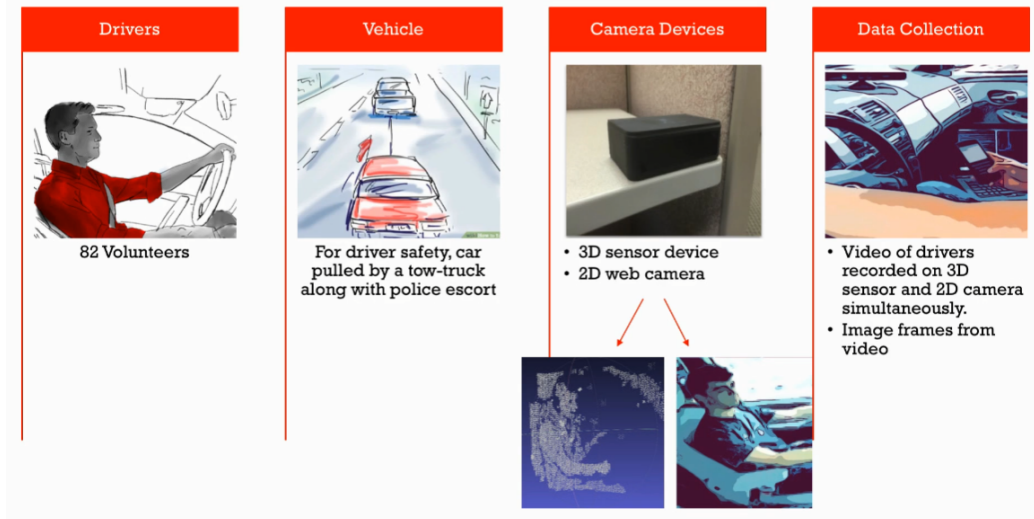
We will be working on Python for creating this model using a set of libraries that include, Numpy, Pandas, Tensorflow, Keras, Matplotlib etc.

The testing and training data used is obtained from Kaggle where an insurance company by the name of State Farm has uploaded a dataset for the same to get the best results from the dev community. The dataset can be found [here](#)

Functionality

The dataset is obtained in strictly monitored conditions to ensure the safety of volunteers and to replicate the real life circumstances revolving around distracted driving

Design of Experiment



The model once trained can be deployed on the web which when feeded with live images of the driver can alert when the driver is distracted or not. This alert can raise an alarm to the driver or co-passengers that they need to focus on the road, thus preventing the possibility of an accident.

References

The Dataset and corresponding details were obtained from [Kaggle](#)

The resources used to learn deep learning and computer vision- [Stanford CS231n](#)

The detailed analysis of Dataset provided by [State Farm](#)

Model Analysis

Loss Function

The loss function in a neural network quantifies the difference between the expected outcome and the outcome produced by the machine learning model. From the loss function, we can derive the gradients which are used to update the weights. The average over all losses constitutes the cost.

In this model, we are creating a classifier, which when given an image will perform operations on it and return the corresponding type of that image which in this case is the type of distraction the driver is undergoing.

It increases exponentially as the prediction diverges from the actual outcome. If the actual outcome is 1, the model should produce a probability estimate that is as close as possible to 1 to reduce the loss as much as possible and similar for 0.

Hence, due to the nature of our model, we will be working on Cross-Entropy based loss functions. These can be Binary, Categorical or Sparse Categorical.

Since we have 10 different classifications, the binary classifier can be ruled out. Now, there is not much difference between Categorical and Sparse Categorical loss function. Sparse categorical is used when the output from our model is not one-hot encoded, however when training our model, we will be using softmax function in the final level to ensure that our output is in one hot encoded format, thus we will be using Categorical Cross Entropy.

Activation Function

There are multiple activation functions that can be used for creating a deep learning model. The most commonly used ones are Sigma, TanH and ReLU. Sigmoid and TanH functions are greatly affected by the vanishing gradient problem where the slope of the curve becomes the same for higher values and our model cannot differentiate between them and are also computationally expensive. ReLU on the other hand is a very simple activation function which returns 'x' when the input is greater than 0 and 0 otherwise. This is also affected by the Vanishing Gradient problem but is compensated for by the inexpensive computation required. The solution to the vanishing gradient is present by using a modified format of ReLU called Leaky ReLU which provides a slope for the input that is less than 0 across the activation function.

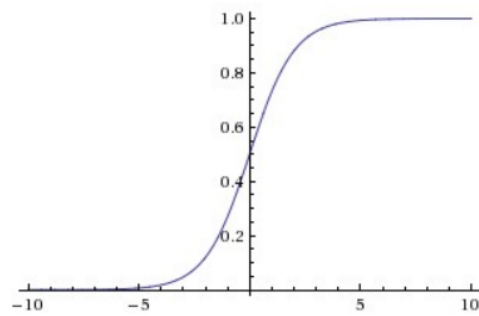
For the more deeper knowledge of these activation functions let's dive in:

SIGMOID FUNCTION:-

It is defined as

$$\sigma(x) = \frac{1}{(1 + e^{-x})}$$

And its graph is as follows

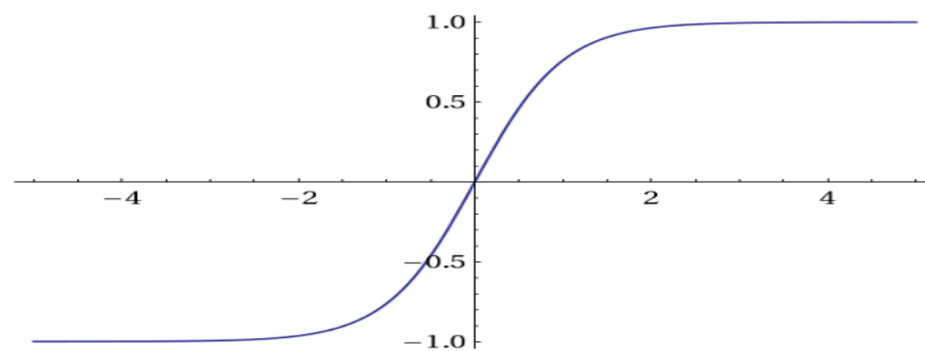


It returns a value between 0 and 1. We didn't make a choice to use it as there is a problem of vanishing gradients with sigmoid function. The output of sigmoid saturates (i.e. the curve becomes parallel to x-axis) for a large positive or large negative number we can clearly see this in graph. Thus, the gradient at these regions is almost zero which is then multiplied with the gradient of this gate during backpropagation. Thus, if the local gradient is very small, it'll kill the gradient and the network will not learn. This problem of vanishing gradient is solved by ReLU.

Tanh(Hyperbolic Tangent) FUNCTION:-

It is defined as

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



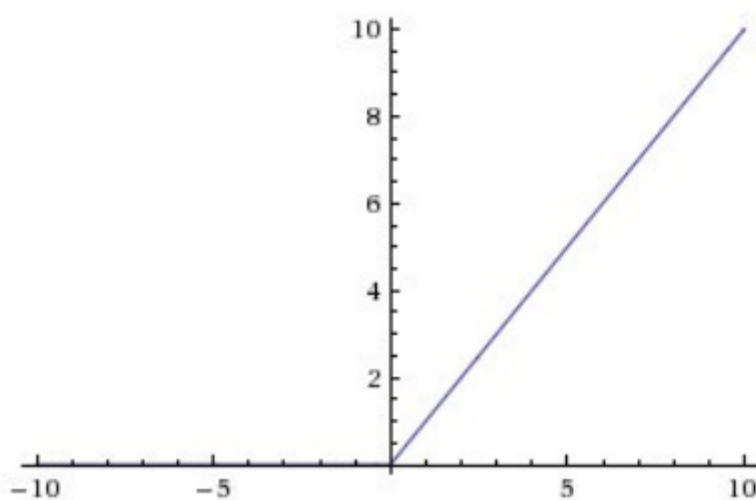
Tanh is just a mathematically shifted version of sigmoid function thus it works little better than sigmoid function but still the problem of vanishing gradients has not been resolved here as you can see again at large positive or large negative number the gradient reduces to zero hence giving out same problem as of sigmoid function. Now to further solve the problem of vanishing gradients ReLU came up as a revolution in activation functions.

ReLU (Rectified Linear Unit) function:-

It is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. So it can be written as

$$f(x) = \max(0, x)$$

And hence its graph can be represented as



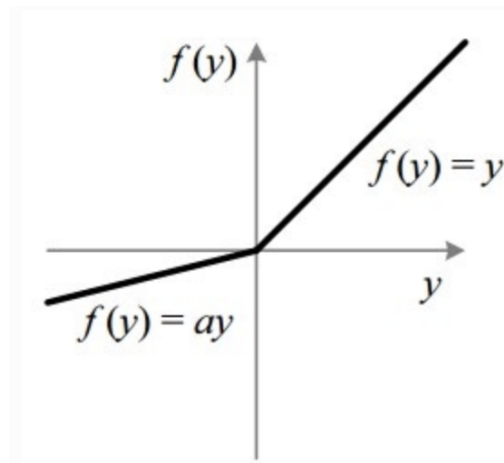
Leaky ReLU:-

In ReLU we have seen that we have solved the problem of vanishing of gradients to some extent as for negative values still gradients are vanishing hence Leaky ReLU is introduced as a variant of ReLU, here instead of being 0 when $z < 0$, leaky ReLU allows a small, non-zero,, constant gradient α (Normally, $\alpha=0.01$).

The function is defined as:-

$$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$$

Here is the graph for Leaky ReLU



Data Exploration :

The dataset from Kaggle provided by StatesFarm has images taken in controlled environments of drivers seated in cars that were being towed to ensure safety of the volunteers. The Dataset downloaded contains the following files/folders :

- imgs.zip - zipped folder of all (train/test) images

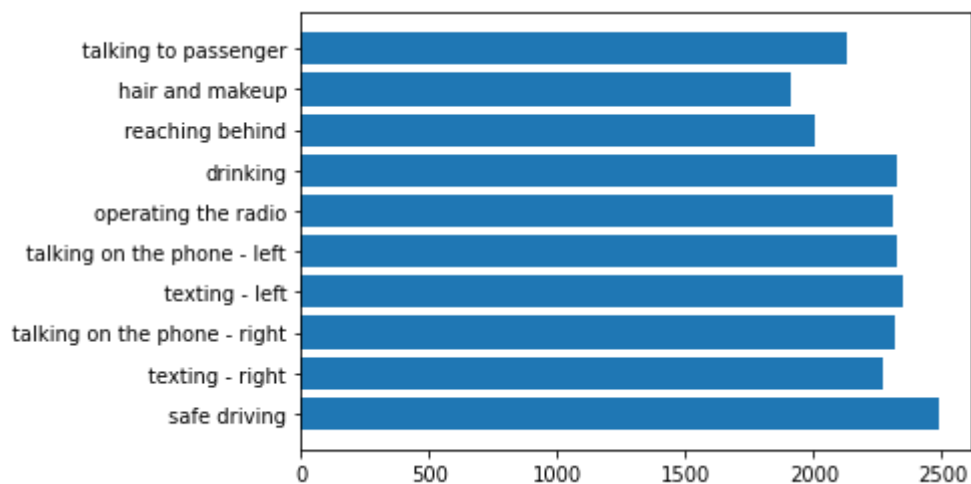
- *sample_submission.csv* - a sample submission file in the correct format
- *driver_imgs_list.csv* - a list of training images, their subject (driver) id, and class id

The *imgs.zip* file contains two directories for testing and training and the training directory has further 10 directories labelled 'c0' to 'c9' where each of the labels corresponds to a unique class for identifying the action of the driver.

The 10 classes to predict are:

- c0: safe driving*
- c1: texting - right*
- c2: talking on the phone - right*
- c3: texting - left*
- c4: talking on the phone - left*
- c5: operating the radio*
- c6: drinking*
- c7: reaching behind*
- c8: hair and makeup*

There are a total of 1,02,150 images of which 22,424 and 79,726 are testing images. The training images are further classified into 10 directories and each of these has nearly 2000 images

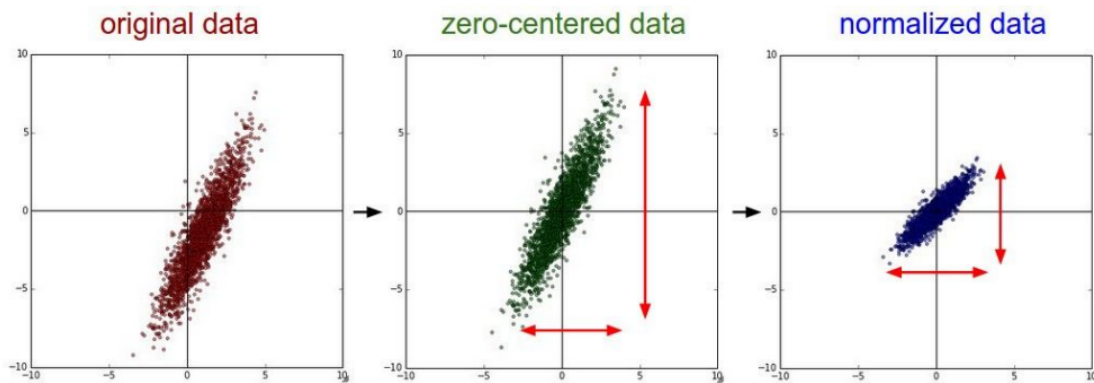


The images are coloured with three channels (rgb) and have 640 x 480 pixels each.

When loaded from the directory, the training data is divided into two parts: training and validation on a 25% split amounting to 16,822 images in the training dataset and 5602 images in the testing dataset.

Data Preprocessing :

The images in their raw format are not ready for training and testing purposes. They need to be processed accordingly for better results and faster computation. The standard techniques revolving preprocessing of data involves, resizing the data, normalising it and making it zero-centred i.e. subtracting a value of 0.5 to ensure that mean becomes 0. The effect of these techniques can be well illustrated in the given example diagram.



To pre process our data, first the data is divided accordingly to training and testing and then further divided to training and validation using `'flow_from_directory'` functionality of `'ImageDataGenerator'`

The images are then resized. This depends entirely on the architecture of the underlying model. We have used different kinds of modelling techniques and hence the size varies as we move across the models.

Since the images provided to use are in RGB format (3 channel), all these are kept and the formatting of images is not changed.

To normalise our images we divide the pixel values of images by 255.0 to get all values in a range of 0 to 1

Finally for have zero-centred data, we can use samplewise or featurewise centring techniques of ImageDataGenerator

The mean calculated on the training dataset as feature-wise centering, requires that the statistic is calculated on the training dataset prior to scaling.

This is different to calculating the mean pixel value for each image, which Keras refers to as sample-wise centering and does not require any statistics to be calculated on the training dataset.

Explanation with CODE:

The first step included importing the required libraries and setting the directories for training and testing data

```
: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten
import pickle4 as pickle
```

```
from numba import cuda
device = cuda.get_current_device()
device.reset()
```

```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

```
train_dir = 'imgs/train'
test_dir = 'imgs/test'
```

The next step involved analyzing the data, and the various classes of input present

```
attribute = {'c0' : "safe driving",
            'c1' : "texting - right",
            'c2' : "talking on the phone - right",
            'c3' : "texting - left",
            'c4' : "talking on the phone - left",
            'c5' : "operating the radio",
            'c6' : "drinking",
            'c7' : "reaching behind",
            'c8' : "hair and makeup",
            'c9' : "talking to passenger"}
```

```

d = {}
for c in classes:
    imgs = [img for img in os.listdir(os.path.join(train_dir,c))]
    count = 0
    for img in imgs:
        count+=1
    d[c]=count

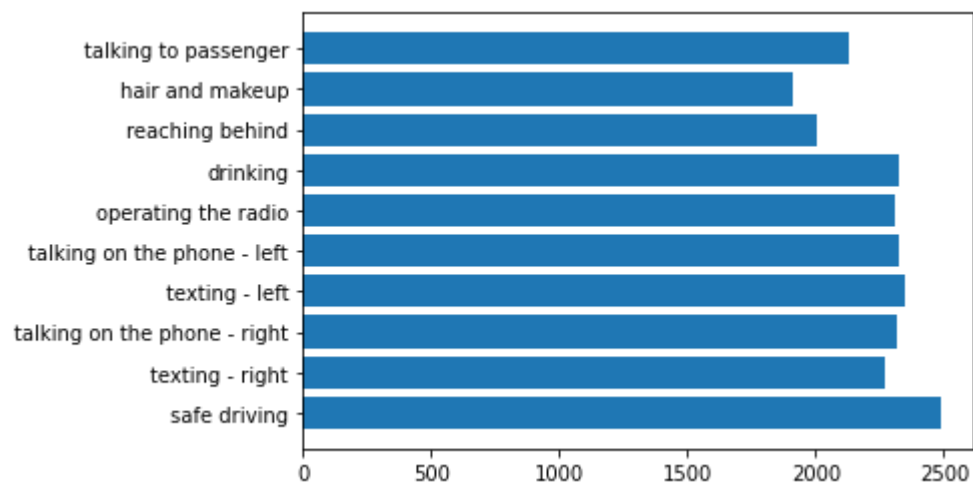
print(d)

k = list(d.keys())
v = list(d.values())
k = [i if i not in k else attribute[i] for i in k]

plt.barh(k,v)
plt.show()

```

From here, we get the following information about our training data



The next step involved loading our dataset into training and validation sets. For this we used ImageDataGenerator. Using this function we were able to standardise the input, samplewise centre it and perform the validation split itself by passing the conditions as parameters.

To load images we used flow_from_directory command which allows us to define batch size, target image size, mode of input, and the subset. Furthermore we used the shuffle command to ensure that the images were randomised and continuous input of one time did not cause the model to overfit

```

train_datagen = ImageDataGenerator(rescale = 1.0/255,validation_split=0.25)
train = train_datagen.flow_from_directory(
    train_dir,
    target_size = (224, 224),
    class_mode = 'categorical',
    batch_size = 16,
    shuffle = True,
    subset = 'training'
)

validate = train_datagen.flow_from_directory(
    train_dir,
    target_size = (224, 224),
    class_mode = 'categorical',
    batch_size = 16,
    shuffle = True,
    subset = 'validation'
)

```

Next, to ensure that the image was loaded accurately, we plotted a sample image to ensure that dimensions were correct and resizing did not distort the image

```

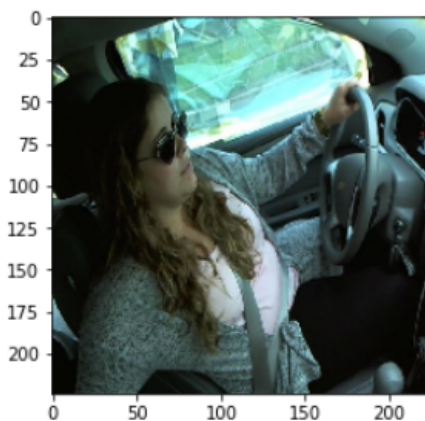
import matplotlib.pyplot as plt
for img,labels in train:
    plt.imshow(img[0])
    print(img.shape)
    print(labels.shape)
    break

```

```

(16, 224, 224, 3)
(16, 10)

```



In the following steps we defined the model that was used, from a base model, to AlexNet, VGG-16, these models further tweaked by using activation functions like LeakyRelu and so on. This was different for each different model.

Here is the command for AlexNet model

```
def convolutional_model():
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(227,227,3)),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
        tf.keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
        tf.keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(4096, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(4096, activation='relu'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation='softmax')])

    return model
```

Next we wrote the command for the callbacks function. Callbacks ensures that the model stops training more epochs after a certain condition is met to prevent overfitting. We performed callback testing on Validation Accuracy.

```
class CustomCallback(tf.keras.callbacks.Callback):
    def __init__(self, threshold):
        super(CustomCallback, self).__init__()
        self.threshold = threshold

    def on_epoch_end(self, epoch, logs=None):
        val_acc = logs["val_accuracy"]
        if val_acc >= self.threshold:
            self.model.stop_training = True
```

```
my_callback = CustomCallback(threshold=0.99)
```

Then we define the loss and optimizer for our model. The loss function we used was Categorical Cross Entropy and the optimizer used varied from base Adam to SGD with tweaked parameters. Here is the piece of code involving SGD function with momentum involved to improve learning rate.

```
loss = tf.keras.losses.CategoricalCrossentropy(from_logits=False,
                                                label_smoothing=0,
                                                axis=-1,
                                                reduction="auto",
                                                name="categorical_crossentropy",
                                                )

sgd = tf.keras.optimizers.SGD(
    learning_rate=0.01,
    momentum=0.9,
    nesterov=False,
    name="SGD"
)
```

Now that all parameters of the model are defined, we finally compile the model.

```

model = convolutional_model()

model.compile(optimizer=sgd,
              loss=loss,
              metrics=['accuracy'],
              )

```

We now obtained a summary of model along with a graph for the same using the following commands.

```
model.summary()
```

```
tf.keras.utils.plot_model(model, show_shapes=True)
```

We now train the model on our training data

```

history = model.fit(train,
                    epochs = 100,
                    verbose = 1,
                    validation_data = validate,
                    callbacks=[my_callback]
                    )

```

Since the model is now trained, we can now save the model using tensorflow built in commands

```

from tensorflow.keras.models import save_model
save_model(model, "myModel.h5")

```

We can also obtain a chart of of loss vs validation loss and accuracy vs validation accuracy to analyse the epochs during training

```

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 10))
ax1.plot(history.history['loss'], color='r', label="Training loss")
ax1.plot(history.history['val_loss'], color='g', label="validation loss")
ax1.set_xticks(np.arange(1, 100, 1))
ax1.set_yticks(np.arange(0, 2, 0.1))

ax2.plot(history.history['accuracy'], color='r', label="Training accuracy")
ax2.plot(history.history['val_accuracy'], color='g', label="Validation accuracy")
ax2.set_xticks(np.arange(1, 100, 1))

legend = plt.legend(loc='best', shadow=True)
plt.tight_layout()
plt.show()

```

Now that we have our model trained, we now need to test it against our testing data. For this we simply run a loop over the directory extracting images one by one, and performing the pre-processing operations then itself and storing the predictions in a list. In the end we keep on appending data to this list and finally convert this list to a dataframe which is then saved in the form of a CSV.

```

c = 0
folder = './imgs/test/'

data = []
for name in os.listdir(folder):
    path = folder+name
    image = tf.keras.preprocessing.image.load_img(path)
    image = image.resize((224, 224))
    # imgplot = plt.imshow(image)
    # plt.show()
    img = tf.keras.preprocessing.image.img_to_array(image)
    img = img/255.0
    img = img.reshape(-1,224, 224, 3)

    pred = model.predict(img)
    row = pred[0].tolist()
    row.insert(0,name)
    data.append(row)
    print(row)
    c+= 1
# if(c==50):
#     break
print(c)
data

```

The operations looks like this :

```

1/1 [=====] - 1s 737ms/step
['img_1.jpg', 0.1002325713634491, 0.09996291249990463, 0.10007422417402267,
0.09998470544815063, 0.09994187206029892, 0.099888876080513, 0.10021279007196
426, 0.09987707436084747, 0.10015776753425598, 0.09966722875833511]
1/1 [=====] - 0s 18ms/step
['img_10.jpg', 0.10021284222602844, 0.09995889663696289, 0.10008985549211502,
0.09999386221170425, 0.0999647006392479, 0.0998440608382225, 0.10023391246795
654, 0.09995604306459427, 0.10013137012720108, 0.09961442649364471]
1/1 [=====] - 0s 17ms/step
['img_100.jpg', 0.10022524744272232, 0.09995529055595398, 0.1000682264566421
5, 0.09998351335525513, 0.09992122650146484, 0.09987419843673706, 0.100198045
37296295, 0.09992078691720963, 0.10021940618753433, 0.09963413327932358]
1/1 [=====] - 0s 17ms/step
['img_1000.jpg', 0.10026417672634125, 0.0999116376042366, 0.1001137867569923
4, 0.09995031356811523, 0.09990755468606949, 0.09982139617204666, 0.100223958
49227905, 0.0999559536576271, 0.10022853314876556, 0.09962271898984909]
1/1 [=====] - 0s 18ms/step
['img_100000.jpg', 0.10025500506162643, 0.09996737539768219, 0.10009939968585
968, 0.09997837990522385, 0.09993379563093185, 0.09985221177339554, 0.1002224

```

```

df = pd.DataFrame(data,columns=['img', 'c0', 'c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9'])

```

```

import csv

df.to_csv('submission.csv',index = False)

```

The file is then submitted on the Kaggle page then determines the score of our model. The model is tested for its log loss score and we went on to improve the score from an initial score of 7 to 1.4

Summary :

When using AlexNet with SGD having Momentum with Label Smoothing, our Model returned a log loss of 2.25

Improving upon it, first by removing Label Smoothing to get more accurate prediction values from softmax function the log loss improved to 2.03

However using Leaky Relu as activation function, this score further improved to 1.82 keeping all other parameters the same.

The best performing model was obtained with base AlexNet and Adam loss function of 1.46 with Relu and 1.57 with Leaky Relu.

This process was then replicated with VGG16.

The base model returned a log loss of 2.85. This drastically improved to 1.83 by keeping most parameters the same and removing label smoothing and samplewise mean operation. This was obtained using ReLu as the activation function.

VGG16 however was taking up a lot of memory, hence the operations were performed with significantly lower batch sizes and could have possibly adversely affected the results. Hyperparameter tuning for VGG16 was not possible due to memory constraints.

Deployment

The h5 file of our model was 200Mb and exceeded the file size limit of Github (25 mb) by a huge margin so it was not possible to deploy the model on the web. But locally it was deployed using Streamlit. The code for the deployment is given below :

```
import io
from PIL import Image
import streamlit as st
import tensorflow as tf
from keras.models import load_model
import numpy as np
import pandas as pd

model = load_model("myModel.h5")

MODEL_PATH = "myModel.h5"
LABELS_PATH = "custom_model/model_classes.txt"
```



```
def load_image():
    uploaded_file = st.file_uploader(label="Pick an image to test")
    if uploaded_file is not None:
        image_data = uploaded_file.getvalue()
        st.image(image_data)
        return Image.open(io.BytesIO(image_data))
    else:
        return None
```

```
# def load_model(model_path):
#     model = torch.load(model_path, map_location='cpu')
#     model.eval()
#     return model
```

```
def load_labels(labels_file):
    with open(labels_file, "r") as f:
        categories = [s.strip() for s in f.readlines()]
    return categories
```

```
def predict(model, image):
    image = image.resize((227, 227))
    # imgplot = plt.imshow(image)
    # plt.show()
    img = tf.keras.preprocessing.image.img_to_array(image)
    img = img / 255.0
    img = img.reshape(-1, 227, 227, 3)
```

```
pred = model.predict(img)
st.write(pred)
attribute = [
    "safe driving",
    "texting - right",
    "talking on the phone - right",
    "texting - left",
    "talking on the phone - left",
    "operating the radio",
    "drinking",
    "reaching behind",
    "hair and makeup",
    "talking to passenger",
```

```


]
index = np.argmax(pred[0])
st.write(
    "The model when run on the given image returns the following classification : "
)
st.write(attribute[index])

def main():
    st.title("Custom model demo")
    # model = load_model(MODEL_PATH)
    # categories = load_labels(LABELS_PATH)
    image = load_image()
    result = st.button("Run on image")
    if result:
        st.write("Calculating results...")
        predict(model, image)

if __name__ == "__main__":
    main()

```

After deployment it showed results in the following manner
 Here an image of a man drinking coffee was uploaded



Run on image

Calculating results...

	0	1	2	3	4
0	3.56940660140026e-7	0.00003169722549500875	0.23082447052001953	6.223763193702325e-8	0.000004:

The model when run on the given image returns the following classification :

drinking