

Configuration Manual

MSc Research Project
Data Analytics

Dhanshree Bauskar
Student ID: x19230460

School of Computing
National College of Ireland

Supervisor: Dr. Catherine Mulwa

National College of Ireland
Project Submission Sheet
School of Computing



Student Name:	Dhanshree Bauskar
Student ID:	x19230460
Programme:	Data Analytics
Year:	2021
Module:	MSc Research Project
Supervisor:	Dr. Catherine Mulwa
Submission Due Date:	16/12/2021
Project Title:	Dance Video Classification into Relevant Street Dancing Styles using Deep Learning Techniques
Word Count:	1941
Page Count:	25

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	16th December 2021

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Dhanshree Bauskar
x19230460

1 Introduction

A Configuration Manual provides information about the software and Hardware requirements to re-implement this research. Also, it gives step by step instructions about the installation of required softwares or tools.

2 Specifications

The pre-requisites for re-implementation of this project.

2.1 Hardware Configuration

The laptop used for this research has the configurations shown in figure 1. It is advised to have high-end configuration laptop, preferably 16 GB RAM to avoid storage issues while compiling the code.

Device specifications	
HP Pavilion Laptop 15-cs2xxx	
Device name	LAPTOP-32C8CMDC
Processor	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
Installed RAM	8.00 GB (7.89 GB usable)
Device ID	7694FFDC-B43A-4FF9-BB67-9A04A4E63A98
Product ID	00327-35846-63681-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	Touch support with 10 touch points

Figure 1: Device Specification

2.2 Software Configuration

The windows operating system of 64-bits was used. For programming Anaconda environment was used as it has library installed with Python as scripting language version 3.7. You can check the version of python as shown in figure 2.

```
(base) C:\Users\Dhanshree>python --version
Python 3.8.8
```

Figure 2: Device Specification

Anaconda Environment can be installed from the site <https://www.anaconda.com/products/individual>. According to the specification of your system, select the suitable version and install. After completion of the installation Anaconda navigator window will be opened as shown below figure 3

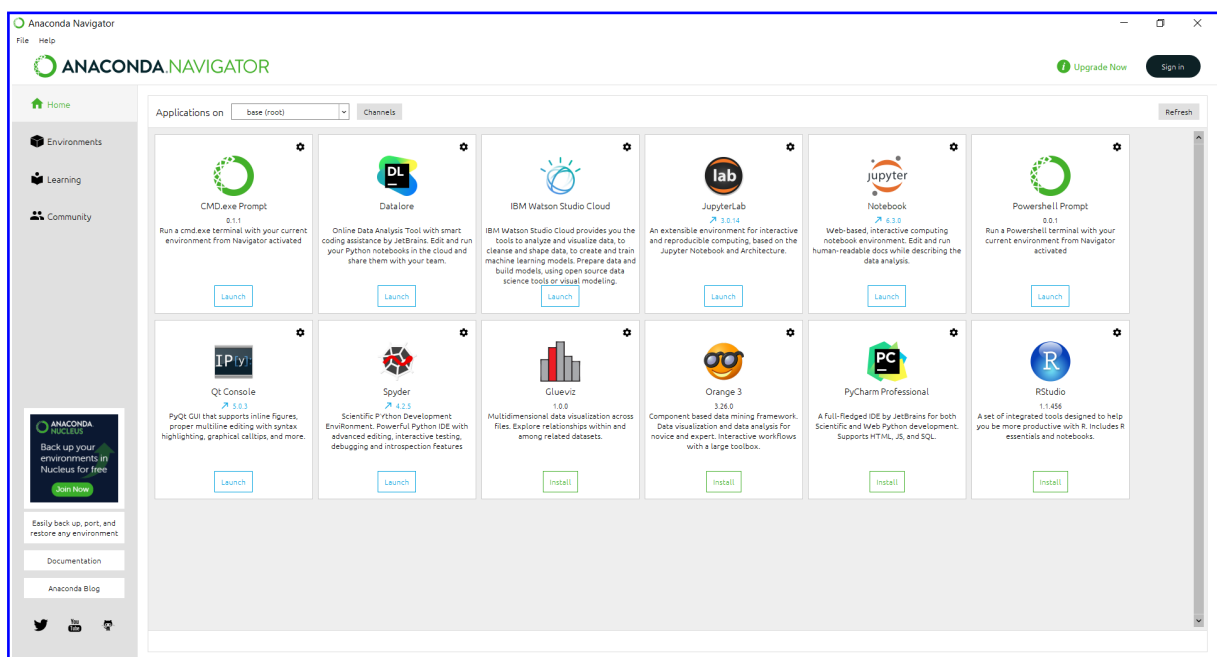


Figure 3: Device Specification

To the left side after clicking to Environment, you can see the installed packages and also you can search and install the packages you want. It can be easily understood by the figure 4 The packages can also be installed by giving command in Anaconda Command Prompt.

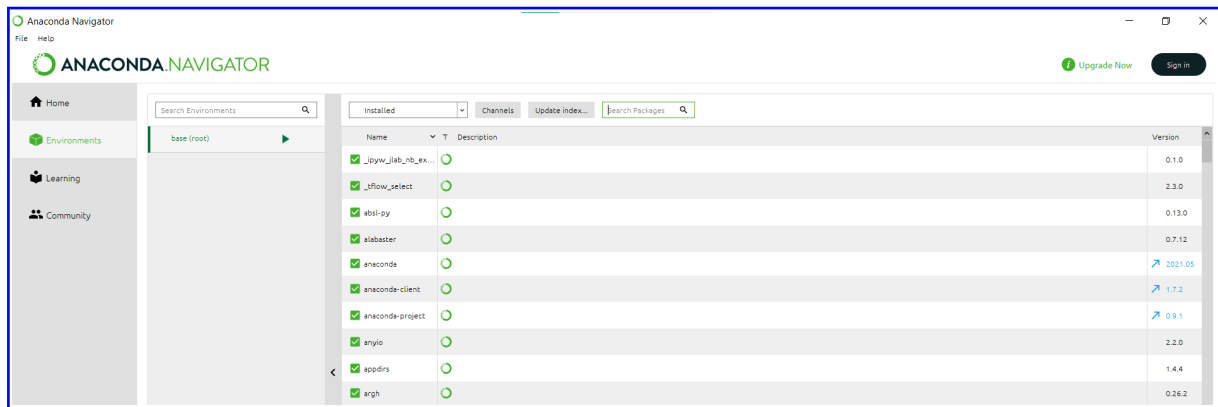


Figure 4: Device Specification

2.2.1 Libraries and packages Used

Variety of libraries were used in the implementation and with Anaconda it becomes easy to install. Libraries that need to be installed separately are mentioned below:

Tensorflow : Tensorflow is a package which will be used for classification. It can be installed by giving command in conda prompt.

```
(base) C:\Users\Dhanshree>conda install -c conda-forge tensorflow
```

Figure 5: Tensorflow installation command

Keras : It is the main package for using deep learning modules.

```
(base) C:\Users\Dhanshree>conda install -c conda-forge keras
```

Figure 6: Keras installation command

Some of the libraries which can be directly imported in the code are mentioned in figure 7.

Jupyter launch: When the jupyter is launched via Anaconda Navigator, it looks as shown in figure 8.

cv2 - Open CV
NumPy – Numerical Python
Glob – returns files that matches specific pattern
Pandas – To make a dataframe
SciPy – To calculate statistics
Sklearn – Contains efficient tools for Statistical Modelling
Matplotlib, Plotly – for plotting graphs
Os – used to create and remove folders

Figure 7: Device Specification

3 Implementation

When using Google Colab, Google all the required files should be stored in google drive and this drive must be connected/mounted to Colab, figure 9 shows the code for connecting drive with the Colaboratory.

The first step for implementation should be importing all the libraries and packages as shown in figure 10

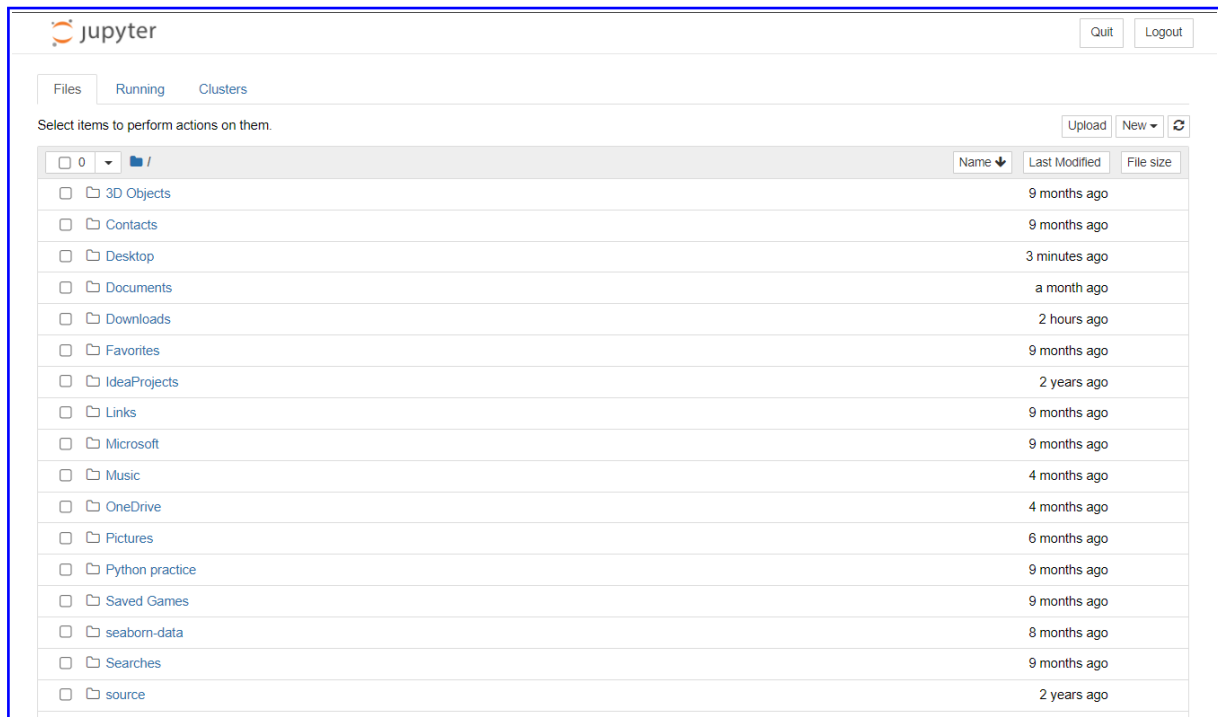


Figure 8: Snapshot of Jupyter launch

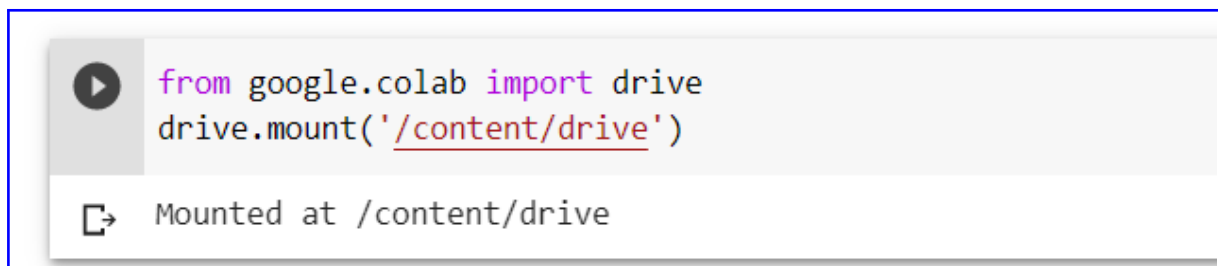


Figure 9: Snapshot of Mounting the Drive in Colab

3.1 Data Preprocessing

All the files must be stored in a folder, so that it will be easy for python command to process. Files should be stored as shown in figure 11

```

import keras
from keras.models import Sequential
from keras.applications.vgg16 import VGG16
from keras.layers import Dense, InputLayer, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, GlobalMaxPooling2D
from keras.preprocessing import image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.model_selection import train_test_split
import os
from glob import glob
import math
import cv2
from scipy import stats as s
import plotly as pt
import plotly.express as px

```

Figure 10: Snapshot of all the required imports

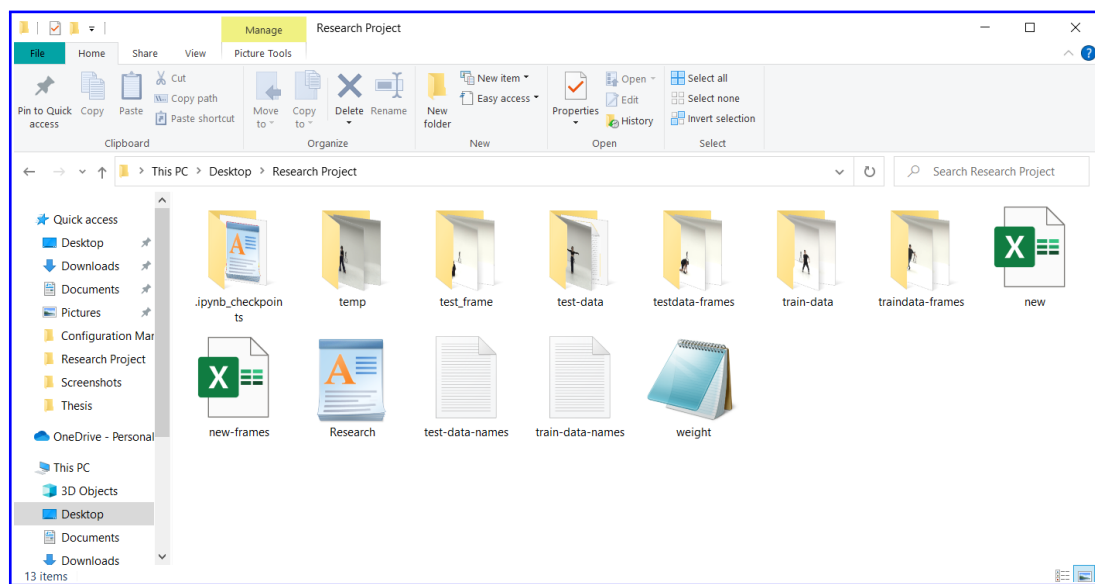


Figure 11: Snapshot of all files stored in a folder

For pre-processing the raw data, firstly, video names stored in txt file should be matched with the name of actual videos and the code for the same is shown in figure 12


```

# Store name of Training videos in a Dataframe
f = open("/content/drive/MyDrive/Dhanshree-dataset/train-data-names.txt", "r")
temp = f.read() # reading the names of training set
videos = temp.split('\n')

# creating a dataframe having video names
train = pd.DataFrame()
train['video_name'] = videos
train = train[:-1]
train_name = train['video_name']
train.head()

# Store name of TEST videos in a Dataframe
f = open("/content/drive/MyDrive/Dhanshree-dataset/test-data-names.txt", "r")
temp = f.read() # reading the names of testing set
videos = temp.split('\n')

# creating a dataframe having video names
test = pd.DataFrame()
test['video_name'] = videos
test = test[:-1]
test_name = test['video_name']
test.head()

```

Figure 12: Snapshot of the code for getting Video Names

3.2 labelling

Videos are labelled according to their dance categories, and the code for same is shown in the figure 13. Two lists were created for train and test videos where tags were to be stored. A 'for' loop runs over the shape of train/test data with starting index of 0, and the list created will be appended by the names stored in the data-frame of train/test. With the help of 'split' function the first letters of the name 'gBR' are separated and stored as a tag to corresponding video name. 0

```

# LABELLING

train_video_tag=[]
test_video_tag=[]

for i in range(train.shape[0]):
    train_video_tag.append(train['video_name'][i].split('_')[0].split(' ')[0])
    train['tag'] = train_video_tag

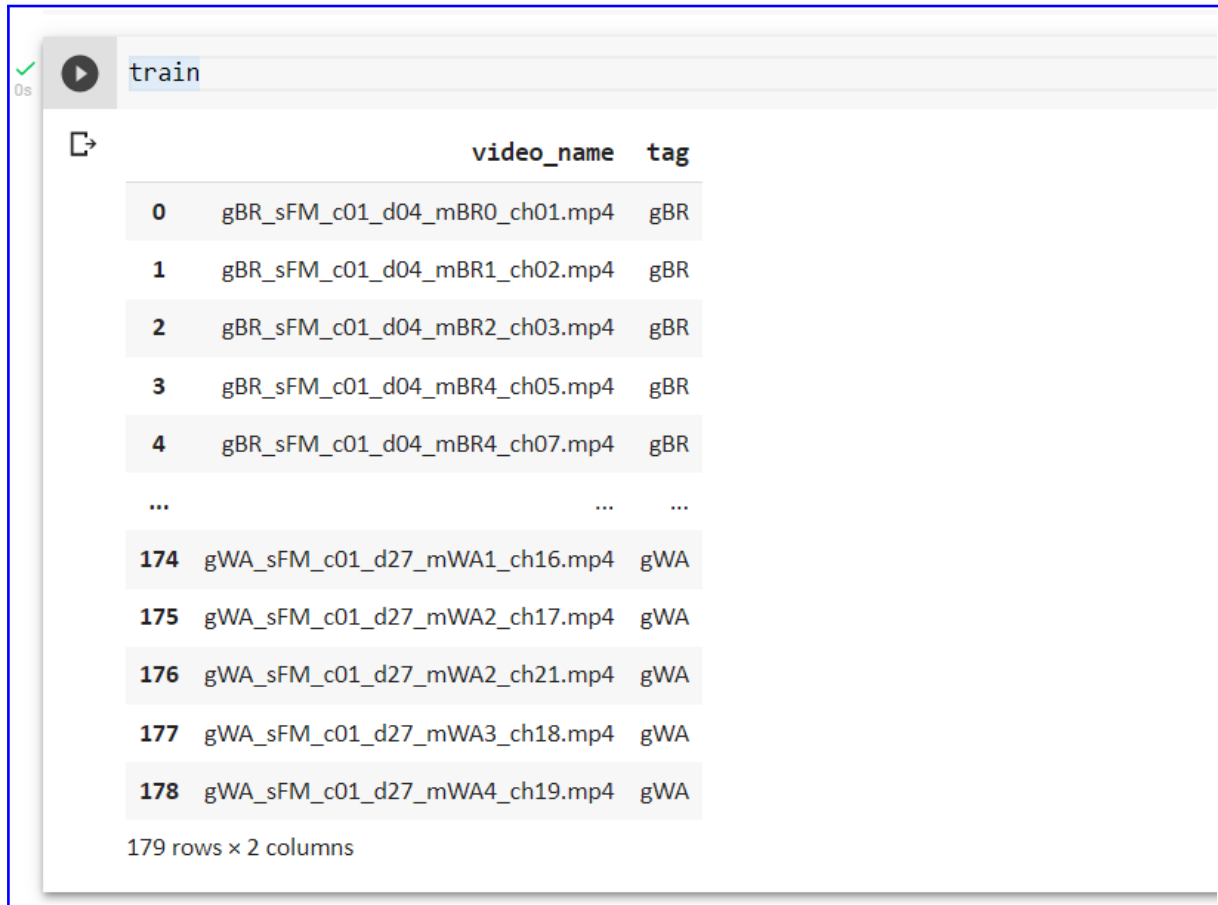
for i in range(test.shape[0]):
    test_video_tag.append(test['video_name'][i].split('_')[0])
    test['tag'] = test_video_tag

```

Figure 13: Snapshot of the code for labelling

For example, it can be seen from the figure 14, that a data-frame is formed with two

columns named 'video_name' and 'tag'.



	video_name	tag
0	gBR_sFM_c01_d04_mBR0_ch01.mp4	gBR
1	gBR_sFM_c01_d04_mBR1_ch02.mp4	gBR
2	gBR_sFM_c01_d04_mBR2_ch03.mp4	gBR
3	gBR_sFM_c01_d04_mBR4_ch05.mp4	gBR
4	gBR_sFM_c01_d04_mBR4_ch07.mp4	gBR
...
174	gWA_sFM_c01_d27_mWA1_ch16.mp4	gWA
175	gWA_sFM_c01_d27_mWA2_ch17.mp4	gWA
176	gWA_sFM_c01_d27_mWA2_ch21.mp4	gWA
177	gWA_sFM_c01_d27_mWA3_ch18.mp4	gWA
178	gWA_sFM_c01_d27_mWA4_ch19.mp4	gWA

179 rows x 2 columns

Figure 14: Snapshot of labels assigned

Frames are extracted from the test video data as well as train video data, can be referred from figure 15 and figure 16. A for loop is created which will run till the shape of the list 'train_name', a counter 'count' is set to be 0 and as the loop goes on each name in train_name will be stored in the variable 'videoFile'. The function 'cv2.VideoCapture' is used to capture the video traversed in the variable 'videoFile' and stored in 'cap' and the frame rate is set to be 5. Another while loop is applied inside the for loop to get every frame and store in a folder named 'traindata-frames' with the extension of frame ID, so that it will be easy to depict the frame number of a particular video.

Similar process is done for capturing frames from test video set and stored in the folder 'testdata-frames'. It can be understood with the figure 16

```
In [6]: # storing the frames from train videos
for i in tqdm(range(train_name.shape[0])):
    count = 0
    videoFile = train_name[i]
    cap = cv2.VideoCapture('train-data/'+videoFile) # capturing the video from the given path
    frameRate = cap.get(5) #frame rate
    x=1
    while(cap.isOpened()):
        frameId = cap.get(1) #current frame number
        ret, frame = cap.read()
        if (ret != True):
            break
        if (frameId % math.floor(frameRate) == 0):
            # storing the frames in a new folder named train_1
            filename = 'traindata-frames/' + videoFile + "_frame%d.jpg" % count;count+=1
            cv2.imwrite(filename, frame)
    cap.release()
```

Figure 15: Snapshot of code extracting frames from video

```
# storing the frames from test videos
for i in tqdm(range(test_name.shape[0])):
    count = 0
    videoFile = test_name[i]
    cap = cv2.VideoCapture('test-data/'+videoFile) # capturing the video from the given path
    frameRate = cap.get(5) #frame rate
    x=1
    while(cap.isOpened()):
        frameId = cap.get(1) #current frame number
        ret, frame = cap.read()
        if (ret != True):
            break
        if (frameId % math.floor(frameRate) == 0):
            # storing the frames in a new folder named train_1
            filename = 'testdata-frame/' + videoFile + "_frame%d.jpg" % count;count+=1
            cv2.imwrite(filename, frame)
    cap.release()
```

100% | 179/179 [53:09<00:00, 17.82s/it]

Figure 16: Snapshot of code extracting frames from video

To read the frames from the folder 'glob' function is used. Two empty lists are created with the name 'train_image' and 'train_class', using for loop the names of the frames captured from the folder will be splitted and stored in a data-frame called train_data. This data-frame will have two columns 'image_name' and 'class', each frame will be assigned with the corresponding classes and stored in an excel file. The code can be seen in the figure 17.


```
Creating Validation Set

In [10]: # separating the target
y = train_data['class']

# creating the training and validation set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=0.2, stratify = y)

In [11]: # creating dummies of target variable for train and validation set
y_train = pd.get_dummies(y_train)
y_test = pd.get_dummies(y_test)
```

Figure 19: Snapshot of creating Validation set

An example of X_test can be seen in figure 20.

```
In [12]: X_test

Out[12]: array([[[[0.7647059 , 0.7647059 , 0.73333335],
 [0.7490196 , 0.74509805, 0.7254902 ],
 [0.74509805, 0.74509805, 0.7137255 ],
 ...,
 [0.77254903, 0.77254903, 0.7411765 ],
 [0.7882353 , 0.7882353 , 0.75686276],
 [0.79607844, 0.79607844, 0.7647059 ]],

 [[0.77254903, 0.77254903, 0.7411765 ],
 [0.7647059 , 0.7607843 , 0.7411765 ],
 [0.7529412 , 0.7529412 , 0.72156864],
 ...,
 [0.78431374, 0.78431374, 0.7529412 ],
 [0.79607844, 0.79607844, 0.7647059 ],
 [0.8 , 0.8 , 0.76862746]],

 [[0.78039217, 0.78039217, 0.7490196 ],
 [0.77254903, 0.76862746, 0.7490196 ],
 [0.7647059 , 0.7647059 , 0.73333335],
```

Figure 20: Snapshot of an example of 'X_test'

A pre-trained VGG-16 model is applied as a base model as seen in figure 21.

```
In [9]: # creating the base model of pre-trained VGG16 model
base_model = VGG16(weights='imagenet', include_top=False)
```

Figure 21: Snapshot of applying Base-Model

Using VGG-16, a pre-trained model features of the frames are extracted for both X_train and X_test. It can be seen from figure 22 that the shape generated is (1328,7,7,512).

```
In [10]: ▶ # extracting features for training frames
X_train = base_model.predict(X_train)
X_train.shape

# extracting features for validation frames
X_test = base_model.predict(X_test)
X_test.shape

Out[10]: (1328, 7, 7, 512)
```

Figure 22: Snapshot of feature Extraction

Next process is Fine-Tuning the model, in this process the frame size are reshaped into 7*7*512 and the pixel values are normalised by dividing each frame by maximum pixel value present in the frame as can be seen in figure 23.

```
Fine-tuning the model

In [11]: ▶ X_train = X_train.reshape(5310, 7*7*512)
X_test = X_test.reshape(1328, 7*7*512)

# normalizing the pixel values
max = X_train.max()
X_train = X_train/max
X_test = X_test/max
```

Figure 23: Snapshot of fine Tuning the model

A model architecture defined for this experiment is shown in the figure 24. A variable named model is assigned with the function sequential that means the model created will be sequential in nature. Five layers of the model are defined with 'relu' as an activation function and required input shape is given. Every layer of the model has different densities, the first layer is 1024 dense followed by second layer having density of 512 then comes third layer with density of 256 subsequently fourth layer with 128 density value and the last layer has the density equals to the number of classes which are to be categorised, in our case the value is 10, as we are classifying 10 dance classes.

```
In [12]: #defining the model architecture
model = Sequential()
model.add(Dense(1024, activation='relu', input_shape=(25088,)))
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Figure 24: Snapshot of Model Architecture

once the architecture is defined, model is trained on the dataset. Here, a function is defined to save the weights of the frames and the model is compiled with loss function as 'Categorical Crossentropy' and optimiser as 'Adam' with 'Accuracy' as metrics.

After compiling the model, it needs to be trained on validation set with batch size given 20 and 15 epochs, which can be seen in figure 25.

Training the model

```
In [14]: # defining a function to save the weights of best model
from keras.callbacks import ModelCheckpoint
mcp_save = ModelCheckpoint('weight.hdf5', save_best_only=True, monitor='val_loss', mode='min')

In [15]: # compiling the model
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])

In [16]: # training the model
model.fit(X_train, y_train, epochs=15, validation_data=(X_test, y_test), callbacks=[mcp_save], batch_size=20)
```

Figure 25: Snapshot of Training the Model

The 15 epochs run while training the model with validation data, from the figure 26, it can be observed that as the accuracy increases with the epochs, the loss function also increases.

The model is then evaluated with weight as 'imagenet', and the weights are loaded in the model as seen in figure 27.

```

In [16]: # training the model
model.fit(X_train, y_train, epochs=15, validation_data=(X_test, y_test), callbacks=[mcp_save], batch_size=20)

Epoch 1/15
266/266 [=====] - 49s 183ms/step - loss: 2.3127 - accuracy: 0.1303 - val_loss: 1.8935 - val_accurac
y: 0.2139
Epoch 2/15
266/266 [=====] - 45s 167ms/step - loss: 1.8477 - accuracy: 0.2452 - val_loss: 1.5053 - val_accurac
y: 0.3667
Epoch 3/15
266/266 [=====] - 42s 159ms/step - loss: 1.5604 - accuracy: 0.3151 - val_loss: 1.1815 - val_accurac
y: 0.5218
Epoch 4/15
266/266 [=====] - 79s 298ms/step - loss: 1.4504 - accuracy: 0.3712 - val_loss: 1.1195 - val_accurac
y: 0.5791
Epoch 5/15
266/266 [=====] - 38s 144ms/step - loss: 1.3924 - accuracy: 0.4026 - val_loss: 1.2263 - val_accurac
y: 0.4631
Epoch 6/15
266/266 [=====] - 41s 153ms/step - loss: 1.2870 - accuracy: 0.4424 - val_loss: 1.3143 - val_accurac
y: 0.4413
Epoch 7/15
266/266 [=====] - 39s 148ms/step - loss: 1.3268 - accuracy: 0.4343 - val_loss: 1.2900 - val_accurac
y: 0.5399
Epoch 8/15
266/266 [=====] - 40s 149ms/step - loss: 1.3670 - accuracy: 0.4322 - val_loss: 1.5461 - val_accurac
y: 0.3208
Epoch 9/15
266/266 [=====] - 37s 139ms/step - loss: 1.3623 - accuracy: 0.4420 - val_loss: 1.1783 - val_accurac
y: 0.5678
Epoch 10/15
266/266 [=====] - 40s 151ms/step - loss: 1.2485 - accuracy: 0.5017 - val_loss: 1.2711 - val_accurac
y: 0.5316
Epoch 11/15
266/266 [=====] - 39s 146ms/step - loss: 1.2150 - accuracy: 0.5136 - val_loss: 1.1422 - val_accurac
y: 0.5489
Epoch 12/15
266/266 [=====] - 38s 142ms/step - loss: 1.3138 - accuracy: 0.4582 - val_loss: 1.2107 - val_accurac
y: 0.4390
Epoch 13/15
266/266 [=====] - 37s 139ms/step - loss: 1.3111 - accuracy: 0.4505 - val_loss: 1.3353 - val_accurac
y: 0.4209
Epoch 14/15
266/266 [=====] - 37s 141ms/step - loss: 1.1332 - accuracy: 0.5399 - val_loss: 1.2199 - val_accurac
y: 0.5858
Epoch 15/15
266/266 [=====] - 36s 137ms/step - loss: 1.0049 - accuracy: 0.6015 - val_loss: 1.4293 - val_accurac
y: 0.5399

Out[16]: <tensorflow.python.keras.callbacks.History at 0x20ae03d4c40>

```

Figure 26: Snapshot of Epochs

Evaluating model

```

In [17]: base_model = VGG16(weights='imagenet', include_top=False)

In [18]: #defining the model architecture
model = Sequential()
model.add(Dense(1024, activation='relu', input_shape=(25088,)))
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# loading the trained weights
model.load_weights('weight.hdf5')

# compiling the model
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])

```

Figure 27: Snapshot of Evaluation

After Evaluation, the model is run on the test data created. As observed from figure 28, two empty lists are created named actual and predict and a variable path. A for

loop is used to extract frames from each test video, as the loop goes on each video gets stored into the variable `videoFile`, three variables `count`, `counter` and `j` are initiated with value 0 followed by a code which clears all the files from the folder named 'temp'. A while loop is used to capture the frames of the video stored in the folder temp with the given frame rate. After capturing the frames, these are stored in the variable `images` using the function 'glob', again an empty list named `predicted.images` is created and a for loop is used to convert the frames into a targeted size and are appended in the list 'predicted.images'. Every time the first for loop runs only one video is captured and frames of one videos are processed and stored in the `predicted.images` list.

```
In [21]: # creating two lists to store predicted and actual tags
predict = []
actual = []
path = 'C:/Users/Dhanshree/Desktop/Research Project/test-data/'
# for loop to extract frames from each test video
for i in tqdm(range(test_videos.shape[0])):
    videoFile = test_videos[i]
    cap = cv2.VideoCapture('test-data/'+videoFile)
    count = 0
    counter = 0
    j = 0
    # removing all other files from the temp folder
    files = glob('temp/*')
    for f in files:
        os.remove(f)
    while cap.isOpened():
        ret, frame = cap.read()
        if ret:
            # filename = 'frames/' + str(j) + '_frame{:d}.jpg'.format(count)
            filename = 'temp/' + "_frame{:d}.jpg" % count; count+=1
            cv2.imwrite(filename, frame)
            counter += 60 # i.e. at 30 fps, this advances one second
            j+= 1
            cap.set(cv2.CAP_PROP_POS_FRAMES, counter)
        else:
            cap.release()
            break
    # reading all the frames from temp folder
    images = glob("temp/*.jpg")

    prediction_images = []
    for i in range(len(images)):
        img = image.load_img(images[i], target_size=(224,224,3))
        img = image.img_to_array(img)
        img = img/255
        prediction_images.append(img)

    # converting all the frames for a test video into numpy array
    prediction_images = np.array(prediction_images)
    # extracting features using pre-trained model
    prediction_images = base_model.predict(prediction_images)
    # converting features in one dimensional array
    prediction_images = prediction_images.reshape(prediction_images.shape[0], 7*7*512)
    # predicting tags for each array
    prediction = model.predict_classes(prediction_images)
    # appending the mode of predictions in predict list to assign the tag to the video
    predict.append(y.columns.values[s.mode(prediction)[0][0]])
    # appending the actual tag of the video
    actual.append(videoFile.split('_')[0])

0%| 0/29 [00:00<?, ?it/s]

WARNING:tensorflow:From <ipython-input-21-74afc5d5b27a>:45: Sequential.predict_classes (from tensorflow.python.keras.engine.sequential) is deprecated and will be removed after 2021-01-01.
Instructions for updating:
Please use instead: * `np.argmax(model.predict(x), axis=-1)`, if your model does multi-class classification (e.g. if it uses a 'softmax' last-layer activation). * `(model.predict(x) > 0.5).astype("int32")`, if your model does binary classification (e.g. if it uses a 'sigmoid' last-layer activation).

100%| 29/29 [09:49<00:00, 20.34s/it]
```

Figure 28: Snapshot of Testing on the Model

Once all the frames of all test videos are stored as list of frames, these are converted to numpy array. Using the base model features of all the frames are extracted and frames are reshaped into the size 7*7*512. With the help of base model tags for the frames are predicted and stored in the empty list 'prediction' and actual tags are appended with the help of tags stored in `videoFile`.

Using sklearn library accuracy score is calculated, as a result training VGG-16 model with the batch size of 20 with 15 epochs gives the accuracy of 37.9% as seen in figure 29

```
In [22]: ▶ # checking the accuracy of the predicted tags
          from sklearn.metrics import accuracy_score
          accuracy_score(predict, actual)*100

Out[22]: 37.93103448275862
```

Figure 29: Snapshot of accuracy

As accuracy is very low, the plot seen in the figure 35 was the part of experiments which failed to give suitable accuracy.

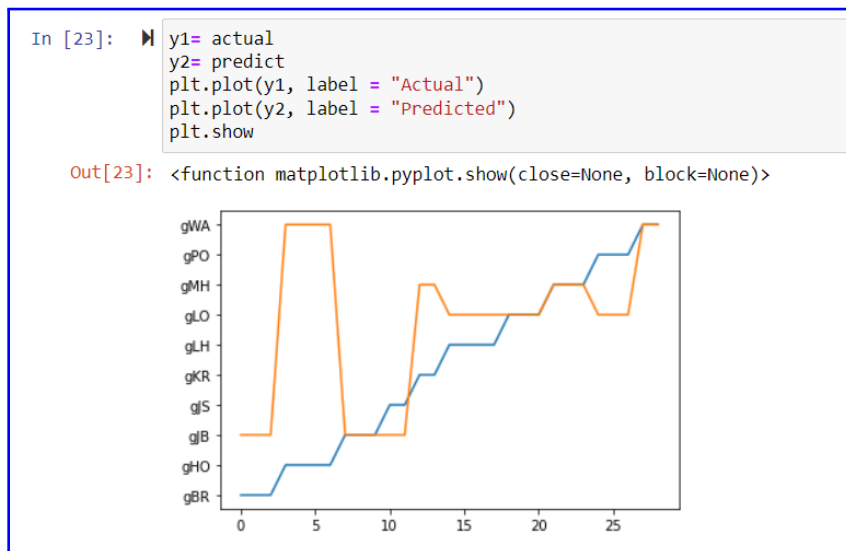


Figure 30: Snapshot of actual and predicted values

When lot of hyper-parameter tuning is done on the dataset, the final accuracy plot achieved from VGG-16 is shown below in figure 31

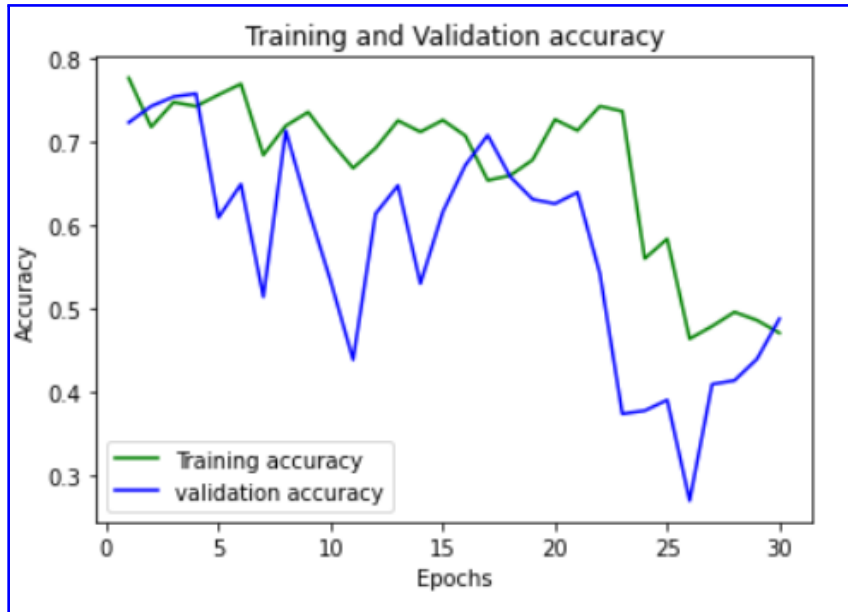


Figure 31: Snapshot of actual and predicted accuracy

```
In [150]: # checking the accuracy of the predicted tags
from sklearn.metrics import accuracy_score
print('The Accuracy for VGG-16 model is :', accuracy_score(predict, actual)*100)

Out[150]: 75.86206896551724
```

Figure 32: Snapshot of final accuracy score achieved

```
In [151]: from sklearn.metrics import f1_score
f1_score(actual, predict, average=None)

Out[151]: array([[1.         , 0.         , 0.5        , 0.         , 1.         ,
                  0.88888889, 1.         , 1.         , 0.8        , 1.         ]])
```

Figure 33: Snapshot of F1-score for VGG-16

```
In [152]: from sklearn.metrics import precision_score
precision_score(actual, predict, average=None)

D:\Softwares\anaconda\lib\site-packages\sklearn\metrics\_classification.py:12
fined and being set to 0.0 in labels with no predicted samples. Use `zero_div
_warn_prf(average, modifier, msg_start, len(result))

Out[152]: array([[1.         , 0.         , 0.33333333, 0.         , 1.         ,
                  0.8        , 1.         , 1.         , 1.         , 1.         ]])
```

Figure 34: Snapshot of precision score for VGG-16

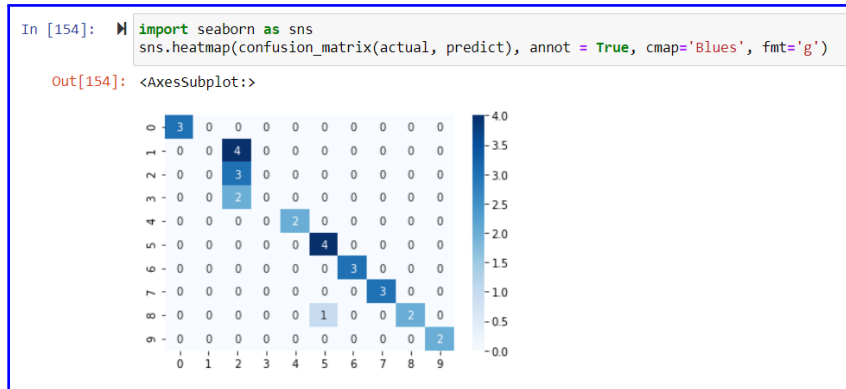


Figure 35: Snapshot of Confusion matrix for VGG-16

3.3 VGG-19 Model for Video Classification

3.3.1 Training of VGG-19

```
VGG-19

In [9]: # creating the base model of pre-trained VGG16 model
VGG19_model = VGG19(weights='imagenet', include_top=False)

In [10]: # extracting features for training frames
X_train_VGG19 = VGG19_model.predict(X_train)
X_train_VGG19.shape

# extracting features for validation frames
X_test_VGG19 = VGG19_model.predict(X_test)
X_test_VGG19.shape

Out[10]: (1328, 7, 7, 512)
```

Figure 36: Snapshot of training VGG-19 Model

```
# normalizing the pixel values
max = X_train_VGG19.max()
X_train_VGG19 = X_train_VGG19/max
X_test_VGG19 = X_test_VGG19/max
```

Figure 37: Snapshot of normalising VGG-19

```

#defining the model architecture
model_VGG19 = Sequential()
model_VGG19.add(Dense(1024, activation='relu', input_shape=(25088,)))
model_VGG19.add(Dropout(0.5))
model_VGG19.add(Dense(512, activation='relu'))
model_VGG19.add(Dropout(0.5))
model_VGG19.add(Dense(256, activation='relu'))
model_VGG19.add(Dropout(0.5))
model_VGG19.add(Dense(128, activation='relu'))
model_VGG19.add(Dropout(0.5))
model_VGG19.add(Dense(10, activation='softmax'))

```

```
model_VGG19.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 1024)	25691136
dropout_4 (Dropout)	(None, 1024)	0
dense_6 (Dense)	(None, 512)	524800
dropout_5 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 256)	131328
dropout_6 (Dropout)	(None, 256)	0
dense_8 (Dense)	(None, 128)	32896
dropout_7 (Dropout)	(None, 128)	0
dense_9 (Dense)	(None, 10)	1290
Total params: 26,381,450		
Trainable params: 26,381,450		
Non-trainable params: 0		

Figure 38: Snapshot of architecture of VGG-19

```

# defining a function to save the weights of best model

from keras.callbacks import ModelCheckpoint
mcp_save_VGG19 = ModelCheckpoint('weight_VGG19.hdf5', save_best_only=True, monitor='val_loss', mode='min')

```

Figure 39: Snapshot of saving weights of VGG-19

```
# compiling the model
model_VGG19.compile(loss='categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])

history_VGG19 = model_VGG19.fit(X_train_VGG19, y_train, epochs=15, validation_data=(X_test_VGG19, y_test), callbacks=[mcp_sav

Epoch 1/15
177/177 [=====] - 29s 163ms/step - loss: 2.1158 - accuracy: 0.1768 - val_loss: 1.7501 - val_accuracy: 0.2146
Epoch 2/15
177/177 [=====] - 28s 158ms/step - loss: 1.5174 - accuracy: 0.3614 - val_loss: 0.9733 - val_accuracy: 0.6770
Epoch 3/15
177/177 [=====] - 27s 153ms/step - loss: 1.1953 - accuracy: 0.4780 - val_loss: 0.7167 - val_accuracy: 0.7530
Epoch 4/15
177/177 [=====] - 27s 153ms/step - loss: 1.0687 - accuracy: 0.5390 - val_loss: 0.6187 - val_accuracy: 0.8012
Epoch 5/15
177/177 [=====] - 26s 147ms/step - loss: 1.0624 - accuracy: 0.5582 - val_loss: 0.6505 - val_accuracy: 0.7824
Epoch 6/15
177/177 [=====] - 26s 148ms/step - loss: 0.9347 - accuracy: 0.6049 - val_loss: 0.9777 - val_accuracy: 0.5723
Epoch 7/15
177/177 [=====] - 26s 149ms/step - loss: 0.8921 - accuracy: 0.6220 - val_loss: 1.0635 - val_accuracy: 0.6137
Epoch 8/15
177/177 [=====] - 26s 149ms/step - loss: 0.8370 - accuracy: 0.6437 - val_loss: 0.7865 - val_accuracy: 0.7184
Epoch 9/15
177/177 [=====] - 27s 150ms/step - loss: 0.7555 - accuracy: 0.6785 - val_loss: 1.0003 - val_accuracy: 0.6717
Epoch 10/15
177/177 [=====] - 27s 154ms/step - loss: 0.6895 - accuracy: 0.7047 - val_loss: 0.5773 - val_accuracy: 0.7658
Epoch 11/15
177/177 [=====] - 26s 147ms/step - loss: 0.7624 - accuracy: 0.6838 - val_loss: 1.6144 - val_accuracy: 0.4270
Epoch 12/15
```

Figure 40: Snapshot of training of VGG-19 model

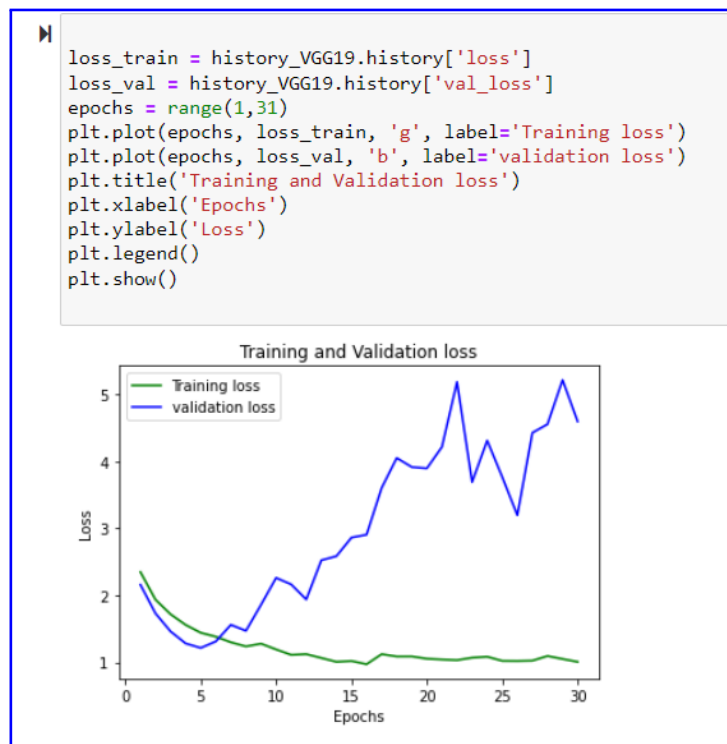


Figure 41: Snapshot of training loss of VGG-19 model

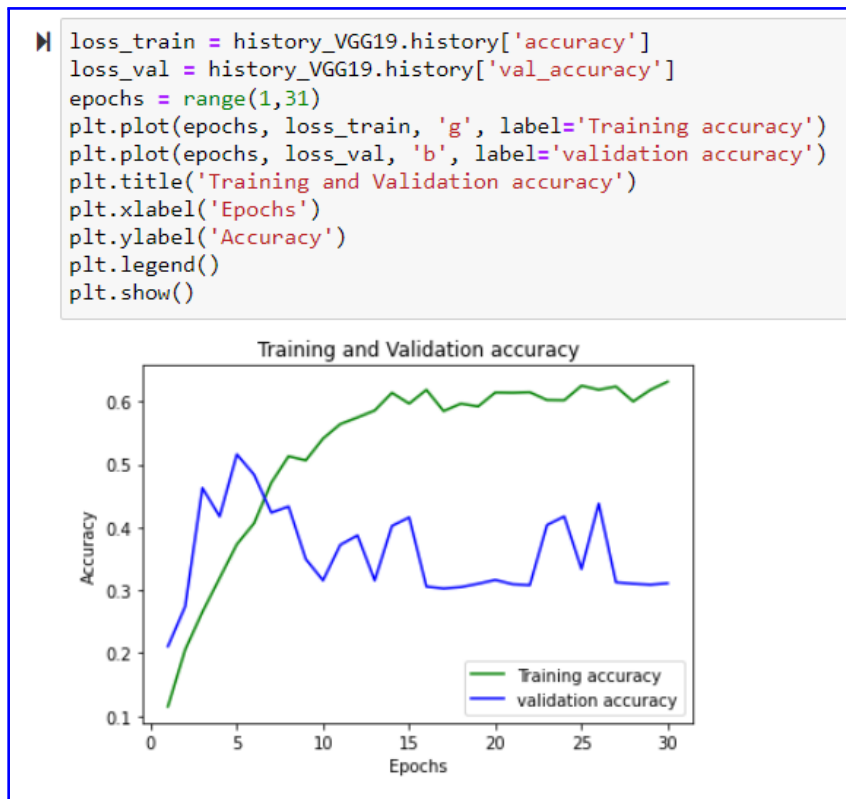


Figure 42: Snapshot of training accuracy of VGG-19 model

Evaluating VGG 19

```

In [37]: > # creating the base model of pre-trained VGG16 model
          VGG19_model = VGG19(weights='imagenet', include_top=False)

In [38]: > #defining the model architecture
          model_VGG19 = Sequential()
          model_VGG19.add(Dense(1024, activation='relu', input_shape=(25088,)))
          model_VGG19.add(Dropout(0.5))
          model_VGG19.add(Dense(512, activation='relu'))
          model_VGG19.add(Dropout(0.5))
          model_VGG19.add(Dense(256, activation='relu'))
          model_VGG19.add(Dropout(0.5))
          model_VGG19.add(Dense(128, activation='relu'))
          model_VGG19.add(Dropout(0.5))
          model_VGG19.add(Dense(10, activation='softmax'))

          # Loading the trained weights
          model_VGG19.load_weights('weight_VGG19.hdf5')

          # compiling the model
          model_VGG19.compile(loss='categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])

```

Figure 43: Snapshot of evaluating VGG-19 model

The final accuracy obtained by model VGG-19 can be seen in figure 44.

```
# checking the accuracy of the predicted tags
from sklearn.metrics import accuracy_score
print('The Accuracy for VGG-19 model is :',accuracy_score(predict_VGG19, actual_VGG19)*100)

The Accuracy for VGG-19 model is : 68.96551724137932
```

Figure 44: Snapshot of final accuracy for VGG-19 model

3.4 CNN Model for Video Classification

CNN model was implemented for this dataset but unfortunately, it did not perform well with the dataset. In figure 45 you can see all the imports necessary to implement CNN model has been imported from Python library.

CNN

```
In [64]: from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Dropout, Flatten
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import plot_model
```

Figure 45: Snapshot of imports needed for CNN

```
# Defining The Model Architecture
model_cnn.add(Conv2D(filters = 224, kernel_size = (3, 3), activation = 'relu', input_shape = (224, 224, 3)))
model_cnn.add(Conv2D(filters = 224, kernel_size = (3, 3), activation = 'relu'))
model_cnn.add(BatchNormalization())
model_cnn.add(MaxPooling2D(pool_size = (2, 2)))
model_cnn.add(GlobalAveragePooling2D())
model_cnn.add(Dense(256, activation = 'relu'))
model_cnn.add(BatchNormalization())
model_cnn.add(Dense(10, activation = 'softmax'))

# Printing the models summary
model_cnn.summary()

print("Model Created Successfully!")

Model: "sequential_7"

```

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 222, 222, 224)	6272
conv2d_10 (Conv2D)	(None, 220, 220, 224)	451808
batch_normalization_5 (Batch Normalization)	(None, 220, 220, 224)	896
max_pooling2d_3 (MaxPooling2D)	(None, 110, 110, 224)	0
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 224)	0
dense_8 (Dense)	(None, 256)	57600
batch_normalization_6 (Batch Normalization)	(None, 256)	1024
dense_9 (Dense)	(None, 10)	2570

```

Total params: 520,170
Trainable params: 519,210
Non-trainable params: 960

Model Created Successfully!

```

Figure 46: Snapshot of architecture of CNN Model

From figure 46 it can be referred that Convolutional Neural Network architecture have been implemented with activation keeping as 'Relu'. Various combinations of epochs and batch-size of input data have been experimented and out of which the result shown by the model in figure 47 was the good performance compared to other results.

```
In [67]: model_cnn.compile(optimizer='rmsprop',loss='sparse_categorical_crossentropy',metrics=['accuracy'])

In [194]: history_cnn = model_cnn.fit(X_train, y_train, epochs=18, validation_data=(X_test, y_test), callbacks=[mcp_save], batch_size=32)

Epoch 1/18
152/152 [=====] - 24s 157ms/step - loss: 1.7614 - accuracy: 0.2603 - val_loss: 1.3998 - val_accuracy: 0.4059
Epoch 2/18
152/152 [=====] - 24s 158ms/step - loss: 1.4142 - accuracy: 0.3889 - val_loss: 0.8660 - val_accuracy: 0.6017
Epoch 3/18
152/152 [=====] - 23s 153ms/step - loss: 1.2126 - accuracy: 0.4578 - val_loss: 0.9198 - val_accuracy: 0.6002
Epoch 4/18
152/152 [=====] - 24s 159ms/step - loss: 1.1861 - accuracy: 0.4736 - val_loss: 0.7453 - val_accuracy: 0.7831
Epoch 5/18
152/152 [=====] - 25s 161ms/step - loss: 1.1117 - accuracy: 0.5098 - val_loss: 0.6842 - val_accuracy: 0.7327
Epoch 6/18
152/152 [=====] - 23s 149ms/step - loss: 1.0585 - accuracy: 0.5330 - val_loss: 0.7127 - val_accuracy: 0.6679
Epoch 7/18
152/152 [=====] - 24s 156ms/step - loss: 0.9631 - accuracy: 0.5821 - val_loss: 0.6871 - val_accuracy: 0.6401
Epoch 8/18
152/152 [=====] - 24s 155ms/step - loss: 1.0233 - accuracy: 0.5667 - val_loss: 1.1115 - val_accuracy: 0.6160
Epoch 9/18
152/152 [=====] - 23s 152ms/step - loss: 0.9509 - accuracy: 0.5970 - val_loss: 1.3896 - val_accuracy: 0.5166
Epoch 10/18
152/152 [=====] - 25s 161ms/step - loss: 0.9440 - accuracy: 0.6124 - val_loss: 1.0335 - val_accuracy: 0.6461
Epoch 11/18
152/152 [=====] - 24s 156ms/step - loss: 1.0008 - accuracy: 0.6090 - val_loss: 1.5196 - val_accuracy: 0.5264
Epoch 12/18
```

Figure 47: Snapshot of epochs run for CNN model

As the training of CNN model was saved in the variable named history_cnn, a plot seen in figure 48 drawn between the training loss and validation loss shows that the training loss decreases as the number of epochs increases.

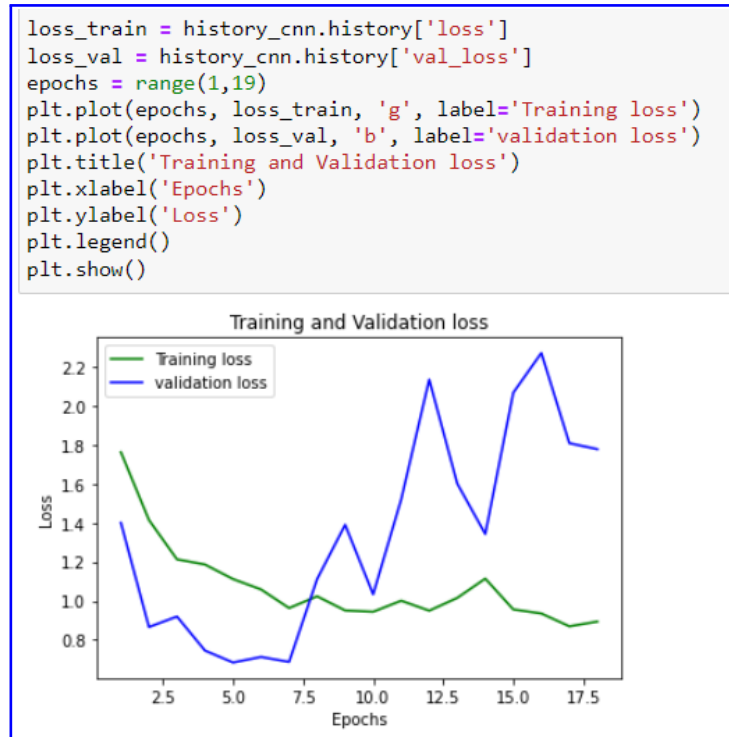


Figure 48: Snapshot of Training and Validation loss of CNN model

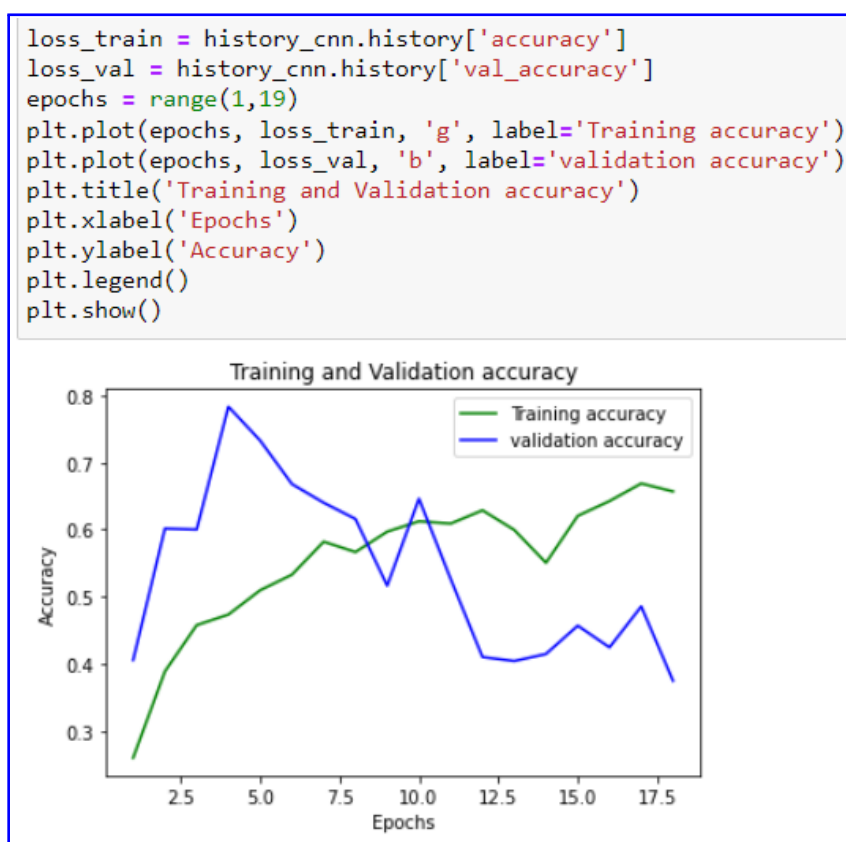


Figure 49: Snapshot of training and validation accuracy of CNN model

The final accuracy result obtained from CNN model on the dataset can be seen in the figure 50 which is 27.58 and the confusion matrix formed by the model is also seen in the figure 50.

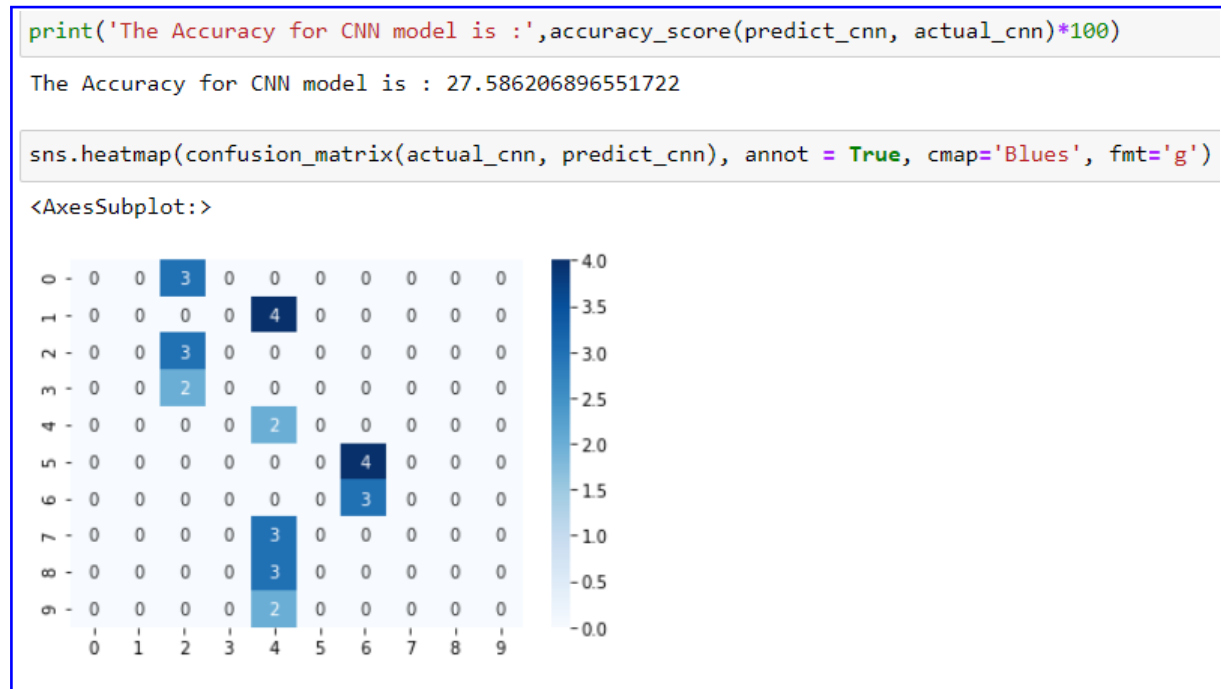


Figure 50: Snapshot of actual and predicted values

4 Issues Faced

While implementing this project various issues were detected and solved according to the need of this research.

Time Consuming: Initially after converting videos into the list of frames, we had 12916 images. Compiling large number of images the device was not compatible and so, the program was shifted to Google Colaboratory. As Google Colab is a shared platform, running high level code is always a task. If the internet is fluctuating, then there are high chances of crashing the code. It almost takes 3-4 hours to compile one piece of code which means it is time consuming regardless of whether it will run properly or not.

Memory Consumption: Processing large video dataset consumes memory, as while implementing allocation of memory to different variables with different size of data.