VIDEO ACTIVITY RECOGNITION WITH TRANSFER LEARNING AND RECURRENT NEURAL NETWORKS USING TIME DISTRIBUTED LAYER IN KERAS

BY

B17 BATCH 1

MANGALAPALLI KRISHNA SAMPATH, 121710317026

PABBA PAVAN KUMAR GOUD, 121710317036

VANAMA BHARGAV SAI, 121710317062

AETUKURI SRIRAM, 121710317001

Under the Guidance of Dr. S. Praveen Kumar

TABLE OF CONTENTS

1. ABSTRACT

With the advancements in Deep Learning, many feats are achievable today that can be considered technologically appealing and also very useful in real life. Our project aims to recognize human activity in videos using Deep Learning and compare Gated Recurrent Unit (GRU) efficiency with Long Short Term Memory (LSTM) and various transfer learning models to determine an optimal and simplistic action recognition mode. Taking video as input, we convert it into sequences of frames, process each frame through a Convolutional Neural Network (CNN), and connect the entire sequence, using a time distributed layer, to LSTM or a GRU. To achieve this, we train our model on UCF101 and HMDB51, two large video activity datasets. UCF101 has 101 action classes that are widespread across various activities, and HMDB51 has 51 action classes that are more focused on humans' movement. Activity recognition is beneficial as it can help us refine many parts of society, especially post-pandemic. This project is our attempt at activity recognition and analysis on a couple of methods that can hopefully be helpful

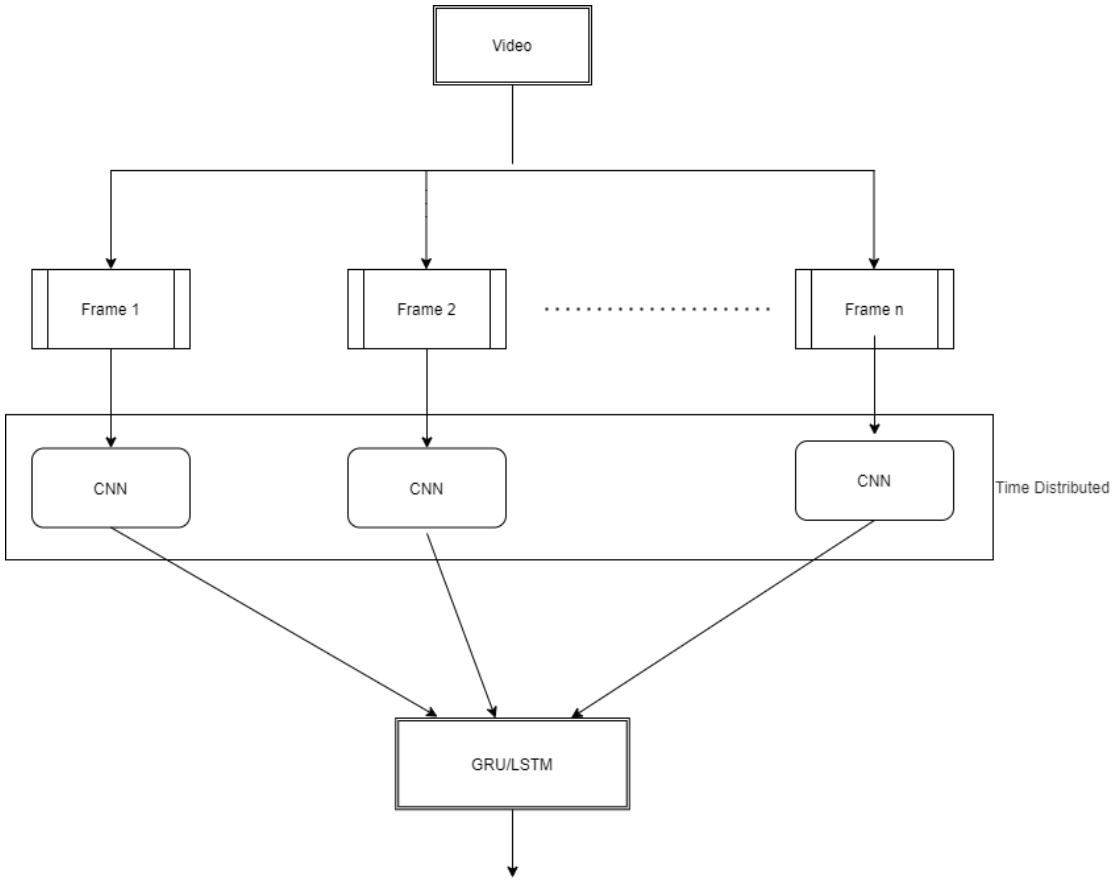
2. PROBLEM STATEMENT

In this project, we aim to build a deep learning model that predicts the activity in a given video . To achieve this we aim to use two datasets UCF101 and HMDB51 which have 101 and 51 action classes with 7GB and 2GB worth data of videos respectively. Our task is to build a classification model that has a CNN + LSTM / GRU by utilising multiple features in Tensorflow and Keras including the time distributed layer. We also compare multiple CNN algorithms like building a CNN from scratch and using transfer learning and find out what is the best model for the use case. We also aim to compare LSTM and GRU and how they perform in video classification problems by analysing the results from this project

3. OBJECTIVES

The primary objectives of this project include the following:

- Applying Deep Learning algorithms to build models to predict the activity in a given video.
- Applying CNN algorithms to process the frames (images) in the video and comparing the performance of the different CNN algorithms applied .
- Applying LSTM (and GRU) to process the sequence frames (images) and classify what action is performed in the video and comparing the performance and efficiency of LSTM and GRU
- Ensure high-accuracy performance of the developed models.
- Prepare a website using flask where uploading a video would provide the prediction.
- Using opencv and the developed models to make predictions and display them via command prompt.



4. DESIGN

- The first part of the model is converting video input to a sequence of frames. We do this with the help of the video generators module in Keras.
- Once we generate the sequence, we process each image from the series through a CNN. We do this by using the time-distributed layer in Keras.
- The processed outputs then connect to LSTM/GRU. While this is a simple architecture, we can optimize the performance and get high accuracies on our datasets.

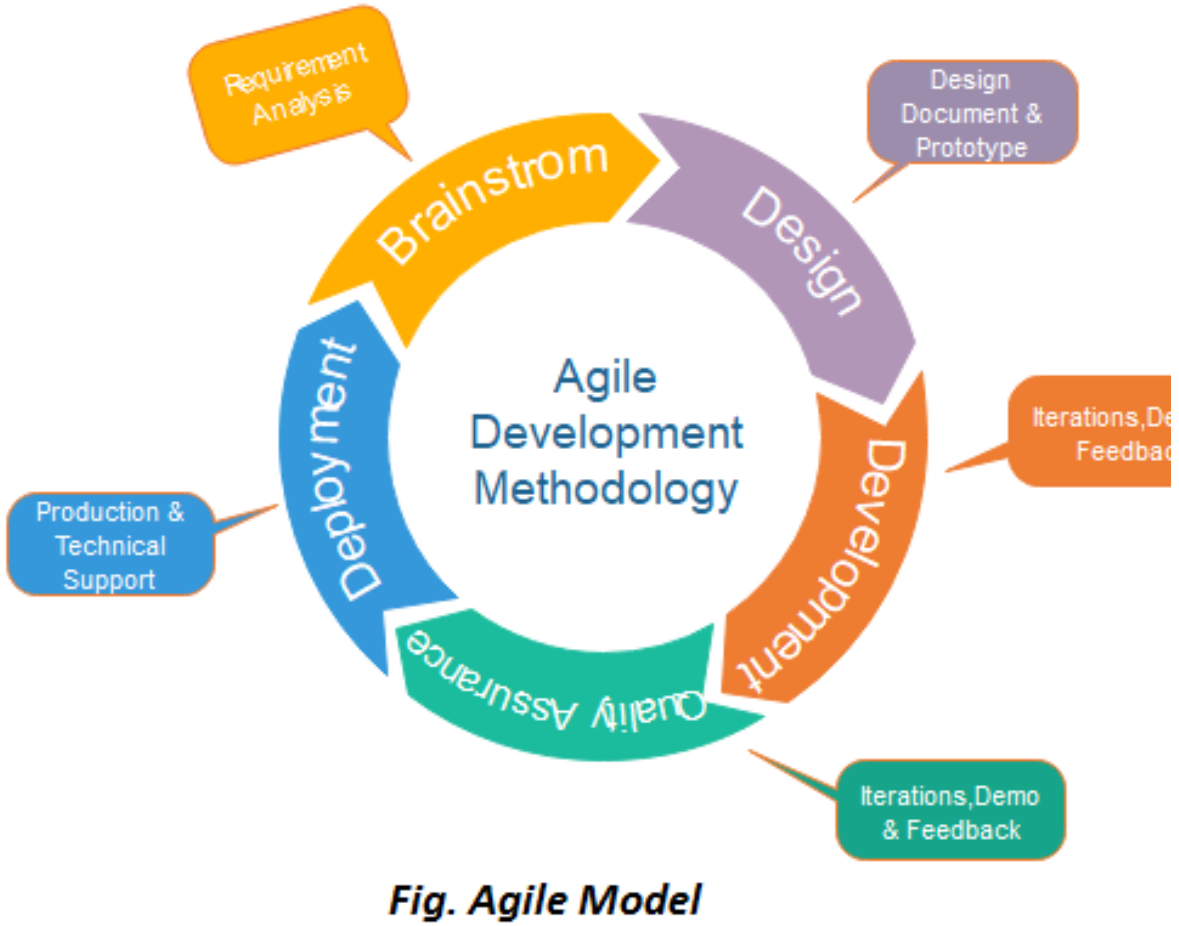


Fig. Agile Model

5. AGILE MODEL

- Requirement Analysis: Gathering video. This is done with the help of UCF101 and HMDB51 datasets
- Design: Splitting video into frames, processing each frame(image through) a CNN model and them combine the sequence over Time Distributed layer and then pass the sequence to LSTM/GRU
- Development: Implementing the design and working on code
- Quality Assurance: Testing results and assessing accuracy of the model
- Deployment: Deploying the design
- Then we use the results and knowledge to make better decisions in the next cycle to increase accuracy, like using Transfer Learning instead of using CNN from scratch

6. DATASETS

- We use a massive video activity dataset called 'UCF101 Dataset'. This dataset contains 101 action classes with more than 13,000 videos that, combined, exceed 27 hours.
- The 101 action classes belong to various categories like human-human interactions, sports, playing musical instruments, body motion activities, and human-object interaction.
- Another dataset HMDB51, a dataset on human motion is also used. It has51 classes, and has over 7000 videos with a size of nearly 2GB.
- The contrast between the two datasets is that UCF101 is a significantly larger dataset and its classes are broader in range and category in comparison.

7. SYSTEM REQUIREMENTS

- Sofware Requirements: Python Version 3.6 or newer. Any modern operating system like Windows 10/7 or Linux, 64 bit is preferrable but not mandatory. Internet connection is required for downloading datasets
  and packages. Tensorflow and Keras must be installed. Open CV and Flask are also required.
- Hardware Requirements: Large storage space is required to store the large dataset, SSD storage can be useful in saving time. More RAM is required, minimum 8 GB is advisable. Recent processors and GPU are essential. We used an 8 core AMD R7 3700x processor and Nvidia RTX 2070 Super for training the models.

8. IMPLEMENTATION

8.1 PROCESSING INPUT VIDEO INTO A SEQUENCE OF FRAMES

- The first part of implementation includes converting input videos to a sequence of frames. We do this by processing each video in the datasets through the VideoFrameGenerator module in Keras.
- This VideoFrameGenerator converts a given video into a sequence of images and also prepares the training and validation sets for the model.
- The global pattern of videos is defined and each video that satisfies the pattern is processed through VideoFrameGenerator and added to the training or validation set.
- Features like number of frames, size of the frame, number of color channels and batch size are also fixed. In this project, we fix the frame size as 224*224 with 3 color channels (representing RGB).

```
glob_pattern='vid/{classname}/*.avi'

data_aug = keras.preprocessing.image.ImageDat
    zoom_range=.1,
    horizontal_flip=True,
    rotation_range=8,
    width_shift_range=.2,
    height_shift_range=.2)

train = VideoFrameGenerator(
    classes=classes,
    glob_pattern=glob_pattern,
    nb_frames=NBFRAME,
    split=.33,
    shuffle=True,
    batch_size=BS,
    target_shape=SIZE,
    nb_channel=CHANNELS,
    transformation=data_aug,
    use_frame_cache=True)
```

8.2 CONVOLUTIONAL NEURAL NETWORK

Once a sequence of frames is generated, we move to the next step. Each frame in the sequence must be individually processed through a CNN model. The following are the methods we used to do this.

--> Building A CNN Model From Scratch

--> Building A MobileNet Model

--> Building A ResNet Model

```python
def build_convnet(shape=(224, 224, 3)):
    momentum = .9
    model = keras.Sequential()
    model.add(Conv2D(64, (3, 3), input_shape=shape,
                     padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(BatchNormalization(momentum=momentum))

    model.add(MaxPool2D())

    model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
    model.add(BatchNormalization(momentum=momentum))

    model.add(MaxPool2D())

    model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
    model.add(BatchNormalization(momentum=momentum))

    model.add(MaxPool2D())

    model.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
    model.add(BatchNormalization(momentum=momentum))


    model.add(GlobalMaxPool2D())
    return model
```

8.2.1 Building A CNN Model From Scratch

- First we tried to build our own CNN model. The input image is fixed for a size of 224*224 with 3 color channels as established earlier.
- This is a standard model with a few Conv2D layers with ReLu activation function, batch normalization and max pooling.
- This model gave decent results for a small part of the datasets but it is found to be inadequate for processing the entire dataset.

```python
def build_mobilenet(shape=(224, 224, 3), nbout=3):
    model = keras.applications.mobilenet.MobileNet(
        include_top=False,
        input_shape=shape,
        weights='imagenet')

    trainable = 9
    for layer in model.layers[:-trainable]:
        layer.trainable = False
    for layer in model.layers[-trainable:]:
        layer.trainable = True
    output = keras.layers.GlobalMaxPool2D()
    return keras.Sequential([model, output])
```

8.2.2 Building A MobileNet Model

- Since the CNN model we built from the scratch is not providing high accuracy, we move on to a more sophisticated method, Transfer Learning.
- Using Transfer Learning, we can use a pre-existing model like MobileNet.
- We convert a few layers (9 in this case) as trainable to customize the outputs of the model to match our use case, as we intend to work this for different datasets.

```
def build_resnet(shape=(224, 224, 3), nbout=3):
    model = tf.keras.applications.ResNet152V2(
    include_top=False,
    weights="imagenet",
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classifier_activation="softmax",)

    trainable = 24

    for layer in model.layers[:-trainable]:
        layer.trainable = False
    for layer in model.layers[-trainable:]:
        layer.trainable = True
    output = keras.layers.GlobalMaxPool2D()
    return keras.Sequential([model, output])
```

8.2.3 Building A ResNet Model

- The MobileNet model gave good accuracy for a part of the dataset but it's performance on the entire dataset is not high enough.
- So, we used another sophisticated model called ResNet instead of MobileNet. ResNet 152 has 152 layers, making it very comprehensive for processing images.
- The weights of this model are already defined on a massive image dataset called imagenet and this is very helpful for processing the individual images in our case.
- We also use the final 24 layers out of the 152 layers by marking them as trainable, which makes us customize ResNet for our use case.

4.3 RECURRENT NEURAL NETWORK

- Once we generate the sequence of images, we have our train and validation sets ready. We have to process them through our action model.
- We define our action model by first passing each image in the sequence through a CNN defined in the previous stem and then connect it via the TimeDistributed Layer in Keras.
- So, after CNN processes each image in the sequence, the entire sequence, which is held together by the TimeDistributed layer, is provided as input to a Recurrent Neural Network.
- In this project we tried both LSTM and GRU for this. The main difference is in their internal architecture and design rather than the code we need to call them from, which is simplified by Tensorflow and Keras.
- After the LSTM/GRU layer, a few dense layers and dropout layers are added. Finally, the last layer is given a softmax activation layer to keep the value of prediction between 0 and 1.
- The field nbout refers to the number of classes in the dataset (101 for UCF101 and 51 for HMDB51) and the final layer provides a value of prediction for every single class.

```
def action_model(shape=(5, 224, 224, 3), nbout=101):

    convnet = build_resnet(shape[1:])

    model = keras.Sequential()
    model.add(TimeDistributed(convnet, input_shape=shape))

    model.add(LSTM(128))

    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(.5))

    model.add(Dense(512, activation='relu'))
    model.add(Dropout(.5))

    model.add(Dense(128, activation='relu'))
    model.add(Dropout(.5))

    model.add(Dense(64, activation='relu'))
    model.add(Dense(nbout, activation='softmax'))

    return model
```

```
app = Flask(__name__)

def model_predict(path):

        pattern = path
        pattern2 = pattern[74:]
        pattern3 = "uploads/"+pattern2

        print('**********************')
        print(pattern3)
        print('**********************')

        test = VideoFrameGenerator(
        glob_pattern=path,
        nb_frames=NBFRAME,
        batch_size=1,
        target_shape=SIZE,
        nb_channel=CHANNELS,
        transformation=data_aug,
        use_frame_cache=True)

        graph = tf.compat.v1.get_default_graph()

        model1 =  load_model('model_best.h5')

        print(model1.predict(test))
        preds = model1.predict(test)
        print(preds)
        print(np.argmax(preds))
        return np.argmax(preds)
        return None
```

4.4 FLASK APPLICATION

4.4.1 PREDICTION FUNCTION

-> The function defined makes predictions for the given video and returns the argument with the highest confidence score from the list generated by the predict function.

-> First the path of the video is given as an argument, so from this path we navigate to the video and provide it as a pattern. This pattern is processed through the VideoFrameGenerator and the video is converted to a sequence of frames that can now be given as input to the model.

-> Then we load the model that we saved using load_model. Then we perform model.predict which generates a list of confidence scores for each of the available classes. We then use the argmax function from numpy to identify the highest confidence score and return that index

```python
@app.route('/', methods=['GET'])
def index():
    # Main page
    return render_template('index.html')


@app.route('/predict', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # Get the file from post request
        f = request.files['file']

        # Save the file to ./uploads
        basepath = os.path.dirname(__file__)
        file_path = os.path.join(
            basepath, 'uploads', secure_filename(f.filename))
        f.save(file_path)

        # # Make prediction
        preds = model_predict(file_path)

        result = classes[preds]

        return result
    return None


if __name__ == '__main__':
    app.run(debug=True)
```

8.4.2 DEFINING THE INDEX AND UPLOAD FUNCTIONS

-> The index function uses the GET method to load the index page.

-> The upload function uses the GET and POST methods. Here with POST method, the prediction result is posted to the web page.

-> We call the model_predict method and it returns the class index of the highest score. With that we navigate to that index in the classes list to find out the predicted class. Then we return this class name as string to the web page.

9. RESULTS

9.1 Comparison of Resnet and Mobilenet

- The first comparison is between Mobilenet and ResNet. There is a massive difference between them in terms of accuracy and loss. The Mobilenet model gave an accuracy of 72% with a training loss of 0.76.
- While the ResNet scored an accuracy and loss of 96.06% and 0.0996, respectively.
- The Mobilenet model's best epoch is 192, while ResNet's is 110. The ResNet model trained significantly quicker than MobileNet.
- With these results, it is safe to assert the ResNet model is the better one for processing the frames, and we use this in the comparison between LSTM and GRU

Table 1: Comparison of Resnet and Mobilenet with LSTM on UCF101 dataset

Model	Best Epoch	Accuracy (in %)	Loss
Mobilenet	192	72.00	0.7616
Resnet	110	96.06	0.0996

9.2 LSTM and GRU

- For the UCF101 dataset, the accuracy and loss are pretty close for both LSTM and GRU. GRU has accuracy and loss of 97.03 and 0.0883, respectively, while LSTM fetched 96.06 and a loss of 0.0996.
- While they are almost identical, it is essential to note that GRU's best epoch is 102, and LSTM is 110.
- GRU is quicker while providing slightly better results when training on nearly 7GB worth of data.
- However, it is a different scenario for HMDB51. LSTM gave an accuracy of nearly 80% with a 0.554 training loss. In contrast, GRU scored an accuracy of only 70% and a loss of 0.835.
- The gap is narrow if you consider validation accuracy and loss, but LSTM still performs better. GRU proved to be quicker, with its best epoch being 73, while LSTM's best epoch is 87.
- Though quicker, GRU's results are not as good considering the superior performance of LSTM.

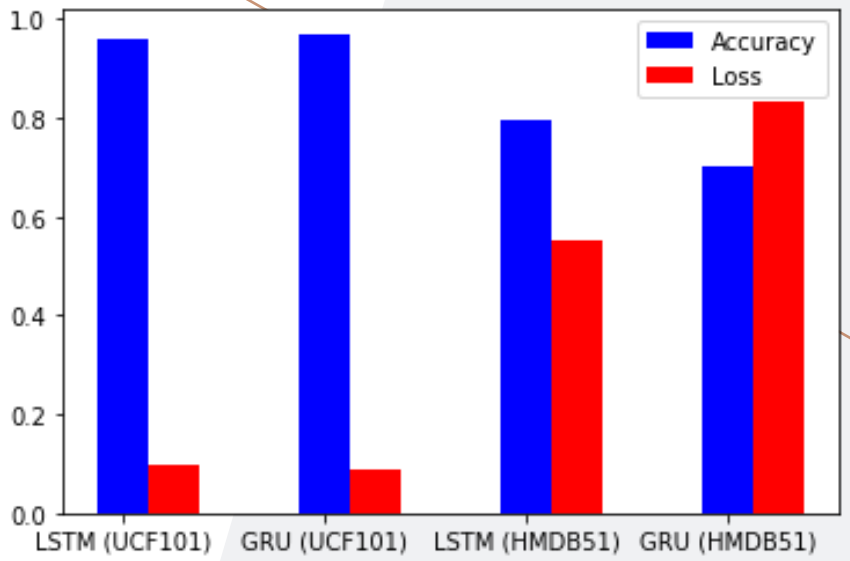
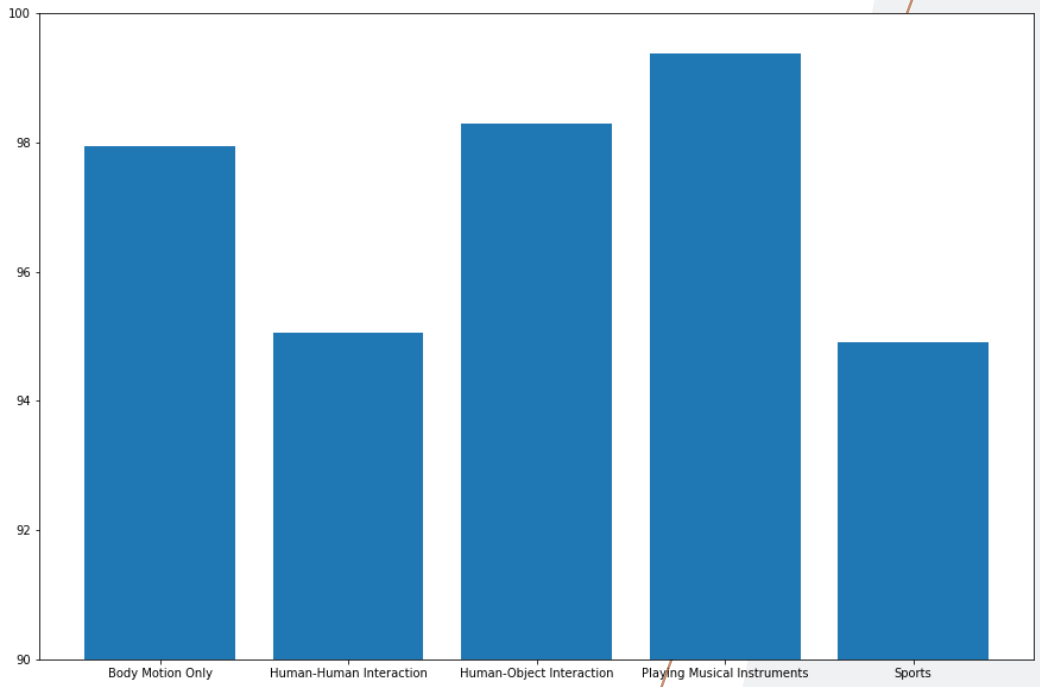


Table 2: Comparison of LSTM and GRU with Resnet on UCF101 and HMDB51 datasets

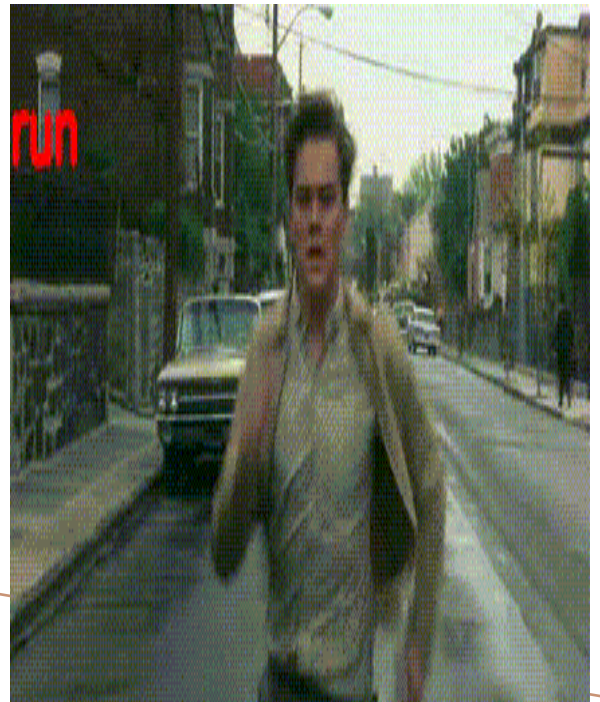
Model	Best Epoch	Accuracy (in %)	Loss	Validation Accuracy (in%)	Validation Loss
LSTM (UCF101)	110	96.06	0.0996	93.93	0.3750
GRU (UCF101)	102	97.03	0.0883	95.09	0.3364
LSTM (HMDB51)	87	79.65	0.5540	58.24	2.36
GRU (HMDB51)	73	70.03	0.8350	54.02	2.25

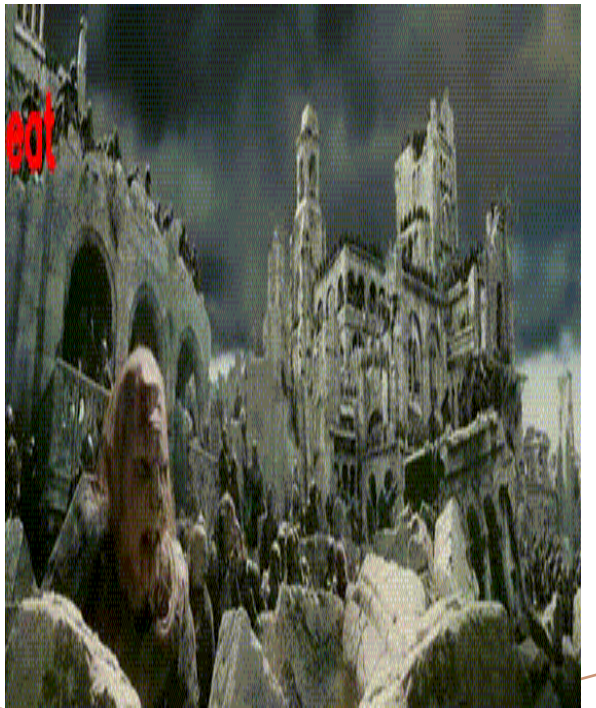


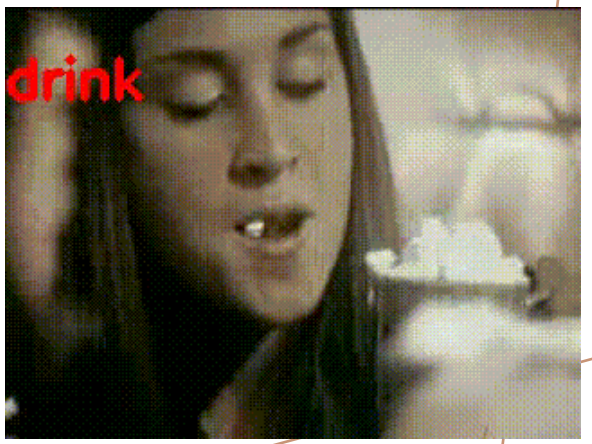
9.3 Accuracy Achieved On The Different Groups Of Activities In UCF101 Dataset

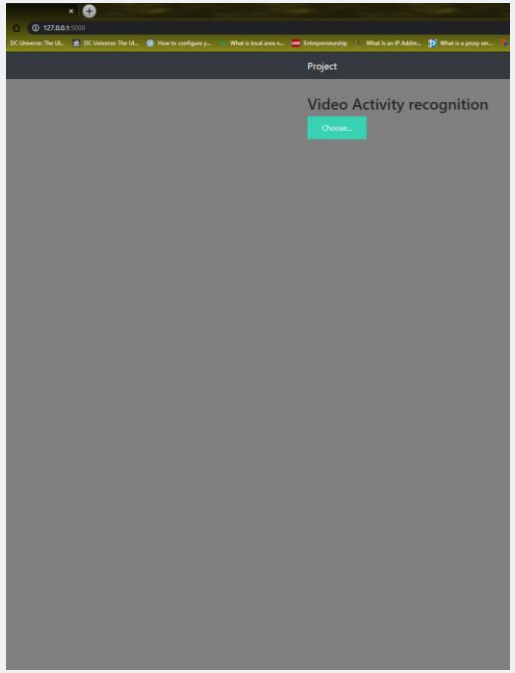
The figure shows the accuracy of each of the five kinds of activities in the USF101 dataset: body motion activities, human-human interactions, sports, playing musical instruments, and human-object interaction. All the different breadth of activities achieved high accuracy. The lowest accuracy achieved is 94.91% for sports, and the highest achieved is 99.37% for playing musical instruments.

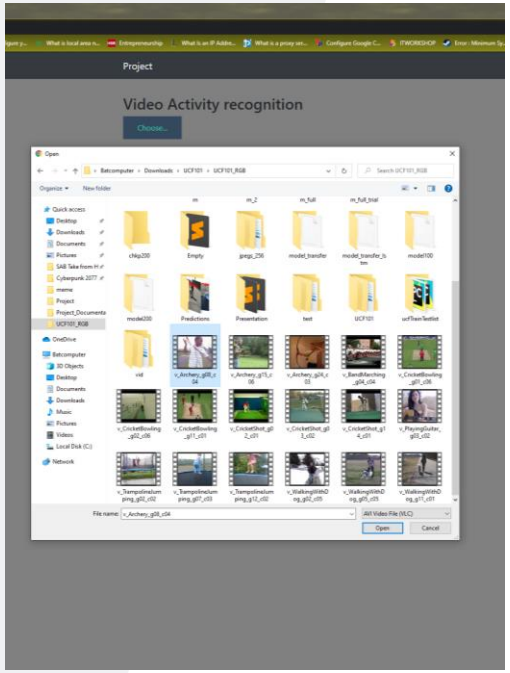
9.4 The Model's Output For A Few Given Input Videos

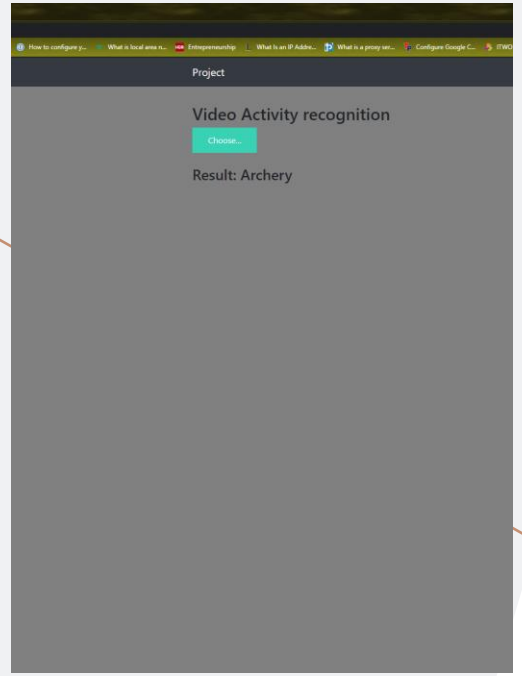


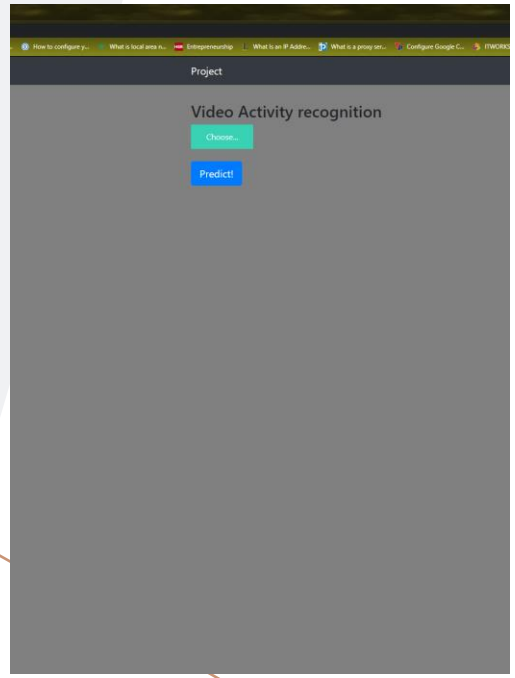












9.4 Screenshots of The Flask Application

We can see the screenshots of the webpage of the flask application here.

10. CONCLUSION

- When comparing LSTM and GRU in activity recognition, the results are nothing short of surprising.
- While the preconceived consensus is that GRU is preferable for smaller datasets and LSTM for the larger ones, this project's findings contrast that assumption. GRU performed slightly better and faster than LSTM when training on UCF101, a large dataset that is nearly 7GB in size.
- However, LSTM is significantly superior when dealing with HMDB51, a 2GB dataset.
- Another essential factor we should consider is the differences between the two datasets. UCF101 has a wide range of classes focusing on various activities, while HMDB51 focuses more on human movement.
- It is not unusual that GRU didn't perform better on the HMDB51 dataset, as it has a simpler architecture, and LSTM has the internal structure to deal with the complexity of the similarity of the actions.

11. FUTURE SCOPE

- GRU's performance on the UCF101 dataset proves that it can handle large amounts of data. However, that depends on the data itself.
- HMDB51, though a smaller dataset, has classes like run and walk, which, when stripped down to frames, look similar.
- There is massive scope for improvement for GRU in distinguishing such complex data. A customized and capable GRU model can potentially provide equivalent or better results than LSTMs while being faster.

THANK YOU