ASANSOL ENGINEERING COLLEGE

TOPIC → HUMAN ACTIVITY RECOGNITION

Batch: 3

Group Number: 9

Group Member:

Name	Roll No.	Registration No.
Ved Prakash	10800120129	201080100110057
Vishal Kumar Prasad	10800120148	201080100110038
Tamal Pal	10800120150	201080100110036
Priya Kumari	10800120179	201080100110007

<u>Algorithms Used:</u>

- **convLSTM:** ConvLSTM is a type of neural network architecture that combines the properties of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks.
  CNNs are commonly used for image processing tasks, where they learn to extract spatial features from the input data. On the other hand, LSTM networks are designed for sequential data processing, where they learn to capture temporal dependencies in the input sequence.

- **<u>LRCN:</u>** LRCN stands for Long-term Recurrent Convolutional Networks. It is a type of neural network architecture that combines the properties of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).
  The LRCN architecture was proposed for video analysis tasks, such as action recognition and video captioning. It consists of a CNN for feature extraction from video frames, followed by an RNN for modeling the temporal dependencies between the video frames. The output of the RNN is then fed into a fully connected layer for final classification.

Steps Used In The Project:

- Step 1: Download and visualize the data with lables
- Step 2: Preprocess the Dataset
- Step 3: Split the data into Train and Test Set
- Step 4: Implement the ConvLSTM Approch
    - Step 4.1: Construct the model
    - Step 5.1: Compile & Train the Model
    - Step 4.3: Plot Model's Loss & Accuracy Curves
- Step 5: Implement the LRCN Approch
    - Step 5.1: Construct the Model
    - Step 5.2: Compile & Train the Model
    - Step 5.3: Plot Model's Loss & Accuracy Curves
- Step 6: Test the best performing Model on YouTube Video

Data Set Used:

```
LINK OF DATASET:
https://www.crcv.ucf.edu/data/UCF50.rar
```

The Dataset contains:

- 50 Action Categories
- 25 Groups of Videos per Action Category
- 133 avg. videos per action category
- 199 avg. number of frames per video
- 320 avg. frames width per video
- 240 avg. frames height per video
- 26 avg. frames per seconds per video

```
!wget --no-check-
certificate https://www.crcv.ucf.edu/data/UCF50.
rar
```

This is used to Download the UCF50 Dataset.

```
!unrar x UCF50
```

This is used to extract the dataset.

Step 1: Download and visualize the dataset:

In the first step we will download and visualize the data along with labels to get an idea about what we will be dealing with. We will be using the UCF50 Dataset.

Command used are explained below:

```
plt.figure(figsize= (20, 20))
```

Create a Matplotlib figure and specify the size of the figure.

```
all_classes_names = os.listdir('UCF50')
```

Get the name of all classes/categories in UCF50

```
random_range = random.sample(range(len(all_class
es_names)), 20)
```

Generate a list of 20 random values. The values will be between 0-50, where 50 is the random total number of class in the dataset.

```
for counter, random_index in enumerate(random_range, 1):

  selected_class_Name = all_classes_names[random_index]

  video_file_names_list = os.listdir(f'UCF50/{selected_class_Name}')

  selected_video_file_name = random.choice(video_file_names_list)

  video_reader = cv2.VideoCapture(f'UCF50/{selected_class_Name}/{selected_
video_file_name}')

  _, brg_frame = video_reader.read()

  video_reader.release()

  rgb_frame = cv2.cvtColor(brg_frame, cv2.COLOR_BGR2RGB)

  cv2.putText(rgb_frame, selected_class_Name, (10, 30), cv2.FONT_HERSHEY_S
IMPLEX, 1, (255, 255, 255), 2)


  plt.subplot(5, 4, counter);plt.imshow(rgb_frame);plt.axis('off')
```

Iterating through all the generated random values and retrieve a Class Name using the random Index.

After that we will Retrieve the list of all the video files present in the randomly selected Class Directory then randomly select a video file from the list retrieved from the randomly selected Class Directory.

After that we will Initialize a Video Capture object to read from the video File.

Read the first frame of the video file.

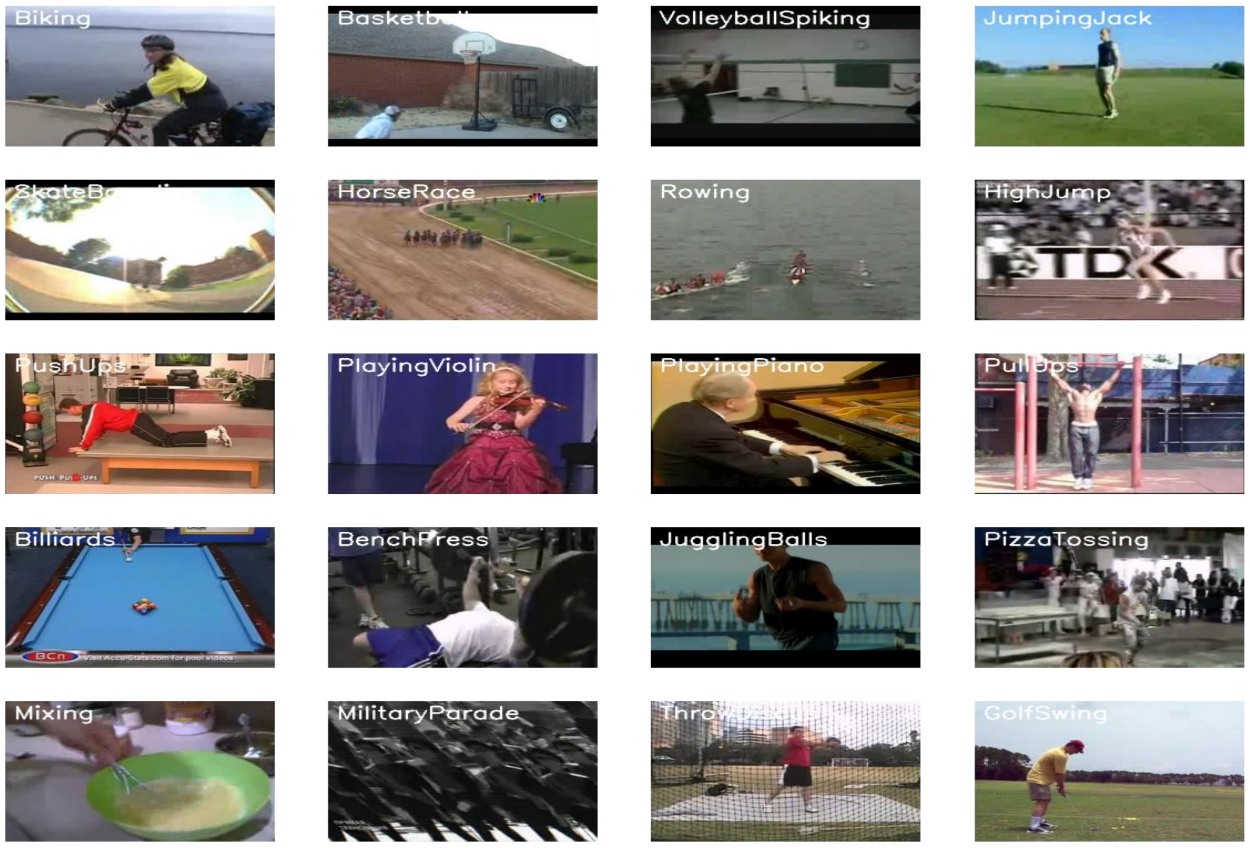
Release the Video Capture object.

Convert the frame from BGR into RGB format.

Write the class name on the video frame

Display the frame.

<u>OUTPUT:</u>



Step 2: Process the dataset:

In this part we are processing the data set i.e. we will read the video files from the dataset and resize the frames of the videos to a fixed width and height.

Command used are explained below:

```
IMAGE_HEIGHT, IMAGE_WIDTH = 64, 64

SEQUENCE_LENGTH = 20

DATASET_DIR = "UCF50"

CLASSES_LIST = ["WalkingWithDog", "TaiChi", "Swing",
"HorseRace"]
```

IMAGE_HEIGHT , IMAGE_WIDTH = 64 , 64 → This will resize the image into 64 by 64 because we already 320 X 20 but we don't need that big size image , also it increase the computation.

64 X 64 is enough to identify what's happening inside the video.

SEQUENCE_LENGTH = 20 → This will sample whole video in 20 frames.

This is done because larger the sequence time greater will be its computation time and it will also take time in inference. So we need to limit the sequence length.

DATASET_DIR = "UCF50" → This is the dataset on which we are working

CLASSES_LIST = ["WalkingWithDog", "TaiChi", "Swing", "HorseRace"] → This is the classes on which the training is to be done. To control the training time we are only working with 4 although we can work on all 50 datasets.

```
def frames_extraction(video_path):

  frames_list = []

  video_reader = cv2.VideoCapture(video_path)

  video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

  skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH), 1)

  for frame_counter in range(SEQUENCE_LENGTH):

      video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window)

      success, frame = video_reader.read()

      if not success:
        break

      resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))

      normalized_frame = resized_frame / 255

      frames_list.append(normalized_frame)

  video_reader.release()

  return frames_list
```

video_reader = cv2.VideoCapture(video_path) → This will read the video by giving the video path.

video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT)) → This we get the frames count for each video.

skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH), 1) → This will the required frame out of any video

video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window) → This will get the required frame which is to be read.

Note:- Reading of the frame is not done by it, it only fetches the required frame.

success, frame = video_reader.read() → It will read the required frame.

resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH)) → It will resize the frame by 64 x 64.

normalized_frame = resized_frame / 255 → It helps in training and also speeds up the training process.

frames_list.append(normalized_frame) → The normalized frame is appended to the frame list.

video_reader.release() → After every thing is done video file is released.

```python
def create_dataset():

  features = []
  labels = []
  video_files_paths = []

  for class_index, class_name in enumerate(CLASSES_LIST):

      print(f'Extracting Data of class: {class_name}')

      files_list = os.listdir(os.path.join(DATASET_DIR, class_name))

      for file_name in files_list:

        video_file_path = os.path.join(DATASET_DIR, class_name, file_name)

        frames = frames_extraction(video_file_path)

        if len(frames) == SEQUENCE_LENGTH:

          features.append(frames)
          labels.append(class_index)
          video_files_paths.append(video_file_path)

  features = np.asarray(features)
  labels = np.array(labels)

  return features, labels, video_files_paths
```

features = [ ] → contains pre processed frames from different categories.

labels = [ ] → contains all class names , categories names or action name of the video.

In 1st for loop we are iterating over those 4 class list that is **["WalkingWithDog", "TaiChi", "Swing", "HorseRace"]**

files_list = os.listdir(os.path.join(DATASET_DIR, class_name)) → This will get all the files list , video list for each class list.

Then in 2nd for loop again we iterating on each file list

video_file_path = os.path.join(DATASET_DIR, class_name, file_name) → This will get complete location of the file.

frames = frames_extraction(video_file_path) → This will get the pre processed frames equal to the sequence length that is we will 20 pre processed frames , after passing the video file path in frames_extraction method.

We have to make sure that **if condition** is true.

features.append(frames) → This will append all the features in the features list.

labels.append(class_index) →This will append the class index in the label list

video_files_paths.append(video_file_path) → This will append all the video file path to the list.

features = np.asarray(features) → This will convert features list into numpy arrays.

labels = np.array(labels) → This will convert labels list into numpy arrays.

Finally we are returning features , labels and video_file_paths

```
features, labels, video_files_paths = create_dataset()
```

In this we are calling create_dataset( ) to get features , labels , and video_file_path

```
one_hot_encoded_labels = to_categorical(labels)
```

In this we are taking the label to_ categorical function and converting that into one_hot_encoded.

Step 3: Split the Data into Train and Test Set:

In this step we are just splitting the one_hot_encoded_labels and features into Train and Test Set.

```
features_train, features_test, labels_train, labels_test =
      train_test_split(features, one_hot_encoded_labels,
      test_size = 0.25, shuffle = True,
      random_state=seed_constant)
```

Split the data into Train → 75% and Test → 25% Set.

Step 4: Split the Data into Train and Test Set:

This function will construct the required convlstlm model.

```
def create_convlstm_model():

  model = Sequential()

  model.add(ConvLSTM2D(filters = 4, kernel_size = (3,3), activation = 'tan
h', data_format="channels_last",
                       recurrent_dropout=0.2, return_sequences=True, input
_shape = (SEQUENCE_LENGTH,

           IMAGE_HEIGHT, IMAGE_WIDTH, 3)))

  model.add(MaxPooling3D(pool_size=(1,2,2), padding='same', data_format='c
hannels_last'))
  model.add(TimeDistributed(Dropout(0.2)))

  model.add(ConvLSTM2D(filters = 8, kernel_size = (3,3), activation='tanh'
, data_format="channels_last",
                       recurrent_dropout=0.2, return_sequences=True))

  model.add(MaxPooling3D(pool_size=(1,2,2), padding='same', data_format='c
hannels_last'))
  model.add(TimeDistributed(Dropout(0.2)))

  model.add(ConvLSTM2D(filters = 14, kernel_size = (3,3), activation='tanh
', data_format="channels_last",
                       recurrent_dropout=0.2, return_sequences=True))

  model.add(MaxPooling3D(pool_size=(1,2,2), padding='same', data_format='c
hannels_last'))
  model.add(TimeDistributed(Dropout(0.2)))

  model.add(ConvLSTM2D(filters = 16, kernel_size = (3,3), activation='tanh
', data_format="channels_last",
                       recurrent_dropout=0.2, return_sequences=True))

  model.add(MaxPooling3D(pool_size=(1,2,2), padding='same', data_format='c
hannels_last'))
  # model.add(TimeDistributed(Dropout(0.2)))

  model.add(Flatten())
  model.add(Dense(len(CLASSES_LIST), activation = "softmax"))

  model.summary()

  return model
```

model = Sequential() → Sequential class used to construt the model

model.add(ConvLSTM2D(filters = 4, kernel_size = (3,3), activation = 'tanh', data_format="channels_last",

recurrent_dropout=0.2, return_sequences=True, input_shape = (SEQUENCE_LENGTH,

IMAGE_HEIGHT, IMAGE_WIDTH, 3))) →

This is a code snippet for adding a Convolutional LSTM layer in a deep learning model using Keras library. Let me explain the different parameters used in this layer:

- filters: This parameter specifies the number of filters in the convolutional layer. Here, we have used 4 filters.
- kernel_size: This parameter specifies the size of the kernel (filter) to be used in the convolution operation. Here, we have used a 3x3 kernel.
- activation: This parameter specifies the activation function to be used in the layer. Here, we have used the hyperbolic tangent function (tanh).
- data_format: This parameter specifies the ordering of the dimensions in the input data. Here, we have used the "channels_last" format where the input shape is (batch, sequence_length, image_height, image_width, channels).
- recurrent_dropout: This parameter specifies the dropout rate for the recurrent connections of the LSTM. Here, we have used a rate of 0.2.
- return_sequences: This parameter specifies whether to return the output sequence or just the last output in the sequence. Here, we have set it to True, so the layer will return the output sequence.
- input_shape: This parameter specifies the shape of the input data. Here, we have specified the input shape as (SEQUENCE_LENGTH, IMAGE_HEIGHT, IMAGE_WIDTH, 3), where SEQUENCE_LENGTH is the length of the input sequence, and IMAGE_HEIGHT and IMAGE_WIDTH are the height and width of the input image, respectively. The last dimension (3) represents the number of color channels in the input image (RGB).

Overall, this layer applies a 2D convolution operation to the input sequence of images using 4 filters and a 3x3 kernel, followed by a 2D LSTM operation with recurrent dropout of 0.2. The output sequence is returned as the layer's output**.**

model.add(MaxPooling3D(pool_size=(1,2,2), padding='same', data_format='channels_last')) →

This is a code snippet for adding a 3D max pooling layer in a deep learning model using Keras library. Let me explain the different parameters used in this layer:

- pool_size: This parameter specifies the size of the pooling window. Here, we have used a pool size of (1, 2, 2), which means the pooling operation will be applied to each 2x2 spatial region along the time dimension.
- padding: This parameter specifies the padding to be applied to the input tensor. Here, we have used the "same" padding, which means the input tensor will be padded with zeros so that the output has the same spatial dimensions as the input.
- data_format: This parameter specifies the ordering of the dimensions in the input data. Here, we have used the "channels_last" format where the input shape is (batch, time, height, width, channels).

Overall, this layer performs max pooling along the time dimension of the input tensor, with a pool size of (1, 2, 2), and with "same" padding.

model.add(TimeDistributed(Dropout(0.2)))

This is a code snippet for adding a time-distributed dropout layer in a deep learning model using Keras library. Let me explain the different parameters used in this layer:

- Dropout: This is a regularization technique used to prevent overfitting in deep learning models. It randomly drops out (sets to zero) a fraction of the input units during training time, which forces the network to learn more robust features that are useful for making predictions on unseen data. Here, we have used a dropout rate of 0.2, which means 20% of the input units will be randomly set to zero during training.
- TimeDistributed: This wrapper allows a layer to be applied to each timestep of a 3D input tensor independently. Here, we have applied the dropout layer to each timestep of the input tensor.

Overall, this layer applies dropout regularization to each timestep of the input tensor with a rate of 0.2

model.add(Flatten()) →

- Flatten: This layer is used to convert the output of the previous layer (which is usually a multi-dimensional tensor) into a 1D tensor that can be used as input to a fully connected (dense) layer. This is necessary because dense layers require a 1D input vector.

Overall, this layer flattens the output of the previous layer into a 1D vector that can be fed into a fully connected layer.

model.add(Dense(len(CLASSES_LIST), activation = "softmax"))

This is a code snippet for adding a dense layer with softmax activation in a deep learning model using Keras library. Let me explain the different parameters used in this layer:

- Dense: This is a fully connected layer where each neuron is connected to every neuron in the previous layer. Here, we are adding a dense layer with a number of neurons equal to the number of classes in the problem (len(CLASSES_LIST)).
- len(CLASSES_LIST): This parameter specifies the number of output neurons in the dense layer, which should match the number of classes in the problem.
- activation: This parameter specifies the activation function to be used in the layer. Here, we have used the softmax activation function, which is commonly used in multi-class classification problems. The softmax function outputs a probability distribution over the classes, where the sum of the probabilities of all classes is equal to one.

Overall, this layer is the output layer of the network and performs the final classification step.

model.summary()

- Display the models summary.

convlstm_model = create_convlstm_model()

- Contruct the required convlstm model.

plot_model(convlstm_model, to_file = 'convlstm_model_structure_plot.png', show_shapes = True, show_layer_names = True) →

This code snippet is using the plot_model function in Keras library to generate a visualization of the architecture of a deep learning model and save it to a file. Here is an explanation of the different parameters used in this function:

- convlstm_model: This is the Keras model object that we want to visualize.

to_file: This parameter specifies the name of the file where the visualization will be saved. Here, we have used the filename "convlstm_model_structure_plot.png".

- show_shapes: This parameter specifies whether to display the shape (number of neurons) of each layer in the visualization. Here, we have set it to True, so the shape of each layer will be displayed.
- show_layer_names: This parameter specifies whether to display the name of each layer in the visualization. Here, we have set it to True, so the name of each layer will be displayed.

Overall, this code snippet generates a visualization of the architecture of the convlstm_model and saves it to a file called "convlstm_model_structure_plot.png".

early_stopping_callback = EarlyStopping(monitor = 'val_loss', patience = 10, mode = 'min', restore_best_weights = True) →
Create an Instance of the Early Stopping Callback

convlstm_model.compile(loss = 'categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])

->
Compile the model and specify loss function, optimizer and metrics values to the model

convlstm_model_training_history = convlstm_model.fit(x = features_train, y = labels_train, epochs=2, batch_size=4, shuffle = True, validation_split = 0.2, callbacks = [early_stopping_callback])

→ Start training the model.

model_evaluation_history = convlstm_model.evaluate(features_test, labels_test) → Evaluate the trained model.

model_evaluation_loss, model_evaluation_accuracy = model_evaluation_history → Get the loss and accuracy from model_evaluation_history.

date_time_format = '%Y_%m_%d__%H_%M_%S' → Define the string date format.

current_date_time_dt = dt.datetime.now()

→ Get the current Date and Time in a DateTime Object.

current_date_time_string = dt.datetime.strftime(current_date_time_dt, date_time_format) → Convert the DateTime object to String according to the style mentioned in date_time_format string.

model_file_name = f'convlstm_model__Date_Time_{current_date_time_string}__Loss_{model_evaluation_loss}__Accuracy_{model_evaluation_accuracy}.h5'

→Define a useful name for our model to make it easy for us while navigating through multiple saved models.

convlstm_model.save(model_file_name) → Save your Model.

metric_value_1 = model_training_history.history[metric_name_1] → Get metric values using metric names as identifiers.

epochs = range(len(metric_value_1)) → Contruct a range object which will be used as x-axis (horizontal plane) of the graph.

plt.plot(epochs, metric_value_1, 'blue', label = metric_name_1) → Plot the Graph.

plt.title(str(plot_name)) → Add title to the plot.

plt.legend() → Add legend to the plot

plot_metric(convlstm_model_training_history, 'loss', 'val_loss', 'Total Loss vs Total Validation Loss') → Visualize the training and validation loss metrices.

Step 5: Implement the LRCN Approach:

This function will construct the required LRCN model:

```
def create_LRCN_model():

  model = Sequential()

  model.add(TimeDistributed(Conv2D(16, (3, 3), padding='same', activation=
'relu'), input_shape=(SEQUENCE_LENGTH, IMAGE_HEIGHT, IMAGE_WIDTH, 3)))

  model.add(TimeDistributed(MaxPooling2D((4, 4))))
  model.add(TimeDistributed(Dropout(0.25)))

  model.add(TimeDistributed(Conv2D(32, (3, 3), padding='same', activation=
'relu')))
  model.add(TimeDistributed(MaxPooling2D((4, 4))))
  model.add(TimeDistributed(Dropout(0.25)))

  model.add(TimeDistributed(Conv2D(64, (3, 3), padding='same', activation=
'relu')))
  model.add(TimeDistributed(MaxPooling2D((2, 2))))
  model.add(TimeDistributed(Dropout(0.25)))

  model.add(TimeDistributed(Conv2D(64, (3, 3), padding='same', activation=
'relu')))
  model.add(TimeDistributed(MaxPooling2D((2, 2))))
  # model.add(TimeDistributed(Dropout(0.25)))

  model.add(TimeDistributed(Flatten()))

  model.add(LSTM(32))

  model.add(Dense(len(CLASSES_LIST), activation='softmax'))

  model.summary()

  return model
```

At the very first we are initializing the model and then the architecture is defined using several layers. (line 9 [model = Sequence ( )])

Conv2D:- It will create a shell in which features of the shell is given.

MaxPooling2D:- It reduces the size image. It also reduces computation. We using MaxPooling2D because we are working with image.

DropOut:-_we are randomly ignoring 20% weight, we don't depend on some key , neuron output, we require whole output.

TimeDistributed:-_ It is used to work with time series data or video frames. It allows to use a layer for each input. That means that instead of having several input "models", we can use "one model" applied to each input.

<u>Note 1 :-</u>

Each layer is in wrapped up in Time Distributed layer as because we are working on a sequence of image frames and every thing is applied to all sequence.

<u>Note 2:-</u>

Multiple times same function are written because we need to create layer after layer to get the architecture.

Flatten( ) :-_ It combines the feature map to output layer.

<u>Note :-</u> In place of flatten( ) , global average pooling can be used which replaces fully connected layers in CNNs.

model.add(LSTM(32)) :-

Dense( ):-_ It the final layer in which number of nodes is equal to the number of class we have.

activation = "softmax" :- It distributes probabilities among the nodes.

model.summary( ) :- It will give summary of the model.

```
LRCN_model = create_LRCN_model()

print("Model Created Successfully!")
```

LRCN_model = create_LRCN_model( ) → This function is called so that we can get the model of the summary. In simple words its just a function call.

```
plot_model(LRCN_model, to_file = 'LRCN_model_structure_plot.png', show_sha
pes=True, show_layer_names=True)
```

plot_model( ) → It checks structure of the constructed model.

```
early_stopping_callback = EarlyStopping(monitor = 'val_loss', patience=15,
 mode='min', restore_best_weights=True)

# Compile the model and specify loss function, optimizer and metrics to th
e model.
```

```
LRCN_model.compile(loss = 'categorical_crossentropy', optimizer='Adam', me
trics=["accuracy"])

LRCN_model_training_history = LRCN_model.fit(x=features_train, y=labels_tr
ain, epochs=30, batch_size=4, shuffle=True, validation_split=0.2, callback
s = [early_stopping_callback])
```

early_stopping_callback :- It stops the network early in the training when it meets the necessary criteria and the criteria is defined in within parenthesis.

Note:- This early_stopping_callback ( ) function is optional.

LRCN_model.compile( ) :- It compiles the model.

LRCN_model.fit ( ) :- In this function training of the model is done using features , labels , batch size , shuffle , etc.,.

```
model_evaluation_history = LRCN_model.evaluate(features_test, labels_test)
```

LRCN_model.evaluate ( ) :- It evaluates the performance of the already trained model.

```
model_evaluation_loss, model_evaluation_accuracy = model_evaluation_histor
y

date_time_format = '%Y_%m_%d__%H_%M_%S'
current_date_time_dt = dt.datetime.now()
current_date_time_string = dt.datetime.strftime(current_date_time_dt, date
_time_format)

model_file_name = f'LRCN_model__Date_Time_{current_date_time_string}__Loss
_{model_evaluation_loss}__Accuracy_{model_evaluation_accuracy}.h5'

LRCN_model.save(model_file_name)
```

In this part we are just saving the model using current date and time.

```
plot_metric(LRCN_model_training_history, 'loss', 'val_loss', 'Total Loss v
s Total Validation Loss')
```

In this part we are simply plotting graph of Total Loss vs Total Validation Loss.

```
plot_metric(LRCN_model_training_history, 'accuracy', 'val_accuracy', 'Tota
l Accuracy vs Total Validation Accuracy')
```

In this part we are simply plotting graph of Total Accuracy vs Total Validation Accuracy.

Step 6: Test the best performing model:

This function downloads the youtube video whose URL is passes to it as an an argument.

```python
def download_youtube_videos(youtube_video_url, output_directory):

  d_video = YouTube(youtube_video_url)

  title = d_video.title

  output_file_path = f'{output_directory}/{title}.3gpp'

  stream = d_video.streams.first()

  stream.download()

  return title
```

d_video = YouTube(youtube_video_url) → Create a video object which contains useful information about the video.

title = d_video.title → Retrieve the title of the video.

output_file_path = f'{output_directory}/{title}.3gpp' → Construct the output file path.

stream = d_video.streams.first() Download the youtube video at the best available quality and store it to the constructed path.

return title → Return the video title.

```python
test_videos_directory = 'test_videos'
os.makedirs(test_videos_directory, exist_ok = True)

video_title = download_youtube_videos('https://www.youtube.com/watch?v=8u0qjmHIOcE', test_videos_directory)

input_video_file_path = f'{test_videos_directory}/{video_title}.3gpp'
```

test_videos_directory = 'test_videos'

os.makedirs(test_videos_directory, exist_ok = True) → Make the output directory if it does not exist.

video_title = download_youtube_videos('https://www.youtube.com/watch?v=8u0qjmHIOcE', test_videos_directory) 🡒 Download a YouTube Video.

input_video_file_path = f'{test_videos_directory}/{video_title}.3gpp' 🡒 Get the YouTube Video's path we just downloaded.

```
def predict_on_video(video_file_path, output_file_path, SEQUENCE_LENGTH):

  video_reader = cv2.VideoCapture(video_file_path)

  original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))
  original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))

  video_writer = cv2.VideoWriter(output_file_path, cv2.VideoWriter_fourcc(
'M', 'P', '4', 'V'), video_reader.get(cv2.CAP_PROP_FPS), (original_video_w
idth, original_video_height))

  frames_queue = deque(maxlen = SEQUENCE_LENGTH)

  predicted_class_name = ''

  while video_reader.isOpened():

    ok, frame = video_reader.read()

    if not ok:
      break

    resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))

    normalized_frame = resized_frame / 255

    frames_queue.append(normalized_frame)

    if len(frames_queue) == SEQUENCE_LENGTH:

        predicted_labels_probabilities = LRCN_model.predict(np.expand_dims
(frames_queue, axis = 0))[0]
        print(predicted_labels_probabilities)

        predicted_label = np.argmax(predicted_labels_probabilities)

        predicted_class_name = CLASSES_LIST[predicted_label]
    cv2.putText(frame, predicted_class_name, (10, 30), cv2.FONT_HERSHEY_SI
MPLEX, 1, (0, 255, 0), 2)

    video_writer.write(frame)

  video_reader.release()
  video_writer.release()
```

Note:

video_file_path → The path of the video stored in the disk on which the action recognition is to be performed.

output_file_path → The path where the output video with the predicted action being performed overlayed will be stored.

SEQUENCE_LENGTH → The fixed number of frames of a video that can be passes to the model as one sequence.

video_reader = cv2.VideoCapture(video_file_path) → Initialize the VideoCapture object to read from the video file.

original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))

original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT)) → Get the width and height of the video.

video_writer = cv2.VideoWriter(output_file_path, cv2.VideoWriter_fourcc('M', 'P', '4', 'V'), video_reader.get(cv2.CAP_PROP_FPS), (original_video_width, original_video_height)) → Initialize the VideoWriter Object to store the output video in the disk.

frames_queue = deque(maxlen = SEQUENCE_LENGTH) → Declare a queue to store video frames.

predicted_class_name = '' → Initialize a variable to store the predicted action being performed in the video.

while video_reader.isOpened(): → Iterate until the video is accessed successfully.

ok, frame = video_reader.read() → Read the frame.

if not ok:

  break → Check if frame is not read properly then break the loop.

resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH)) → Resize the Frame to fixed Dimensions.

normalized_frame = resized_frame / 255 → Normalize he resized frame by dividing it with 255 so that each pixel value then lies between 0 and 1.

frames_queue.append(normalized_frame) → Appending the pre-processed frmae into the frames list.

if len(frames_queue) == SEQUENCE_LENGTH: → Check if the number of frames in the queue are equal to the fixed sequence length

predicted_labels_probabilities = LRCN_model.predict(np.expand_dims(frames_queue, axis = 0))[0]

print(predicted_labels_probabilities) → Pass the normalized frames to the model and get the predicted probabilities.

predicted_label = np.argmax(predicted_labels_probabilities) → Get the index of class with highest probability.

predicted_class_name = CLASSES_LIST[predicted_label] → Get the class name using the retrieved index.

cv2.putText(frame, predicted_class_name, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2) → Write predicted class name on top of the frame.

video_writer.write(frame) → Write the frame into the disk using the VideoWriter object.

video_reader.release()

video_writer.release() → Release the VideoCapture and VideoWriter

```
output_video_file_path = f'{test_videos_directory}/{video_title}-Output-
SeqLen{SEQUENCE_LENGTH}.mp4'

input_video_file_path = f'/content/{video_title}.3gpp'

predict_on_video(input_video_file_path, output_video_file_path, SEQUENCE_L
ENGTH)

VideoFileClip(output_video_file_path, audio=False, target_resolution=(300,
 None)).ipython_display()
```

output_video_file_path = f'{test_videos_directory}/{video_title}-Output-SeqLen{SEQUENCE_LENGTH}.mp4' → Construct the output video path.

predict_on_video(input_video_file_path, output_video_file_path, SEQUENCE_LENGTH) → Perform Action Recognition on the Test Video.

VideoFileClip(output_video_file_path, audio=False, target_resolution=(300, None)).ipython_display() → Display the output video.

Conclusion

We successfully developed a human activity recognition system using Python and machine learning techniques. Our results demonstrate the potential of using sensor data and machine learning to accurately recognize human activities. This can have important applications in healthcare, sports, and security.

THANK YOU