Name: Khuram Shahzad (p21 8742)

Subject: Recent Advances in Deep Learning

Submitted To: Dr. Waqas Ali

Human Action Recognition in Videos using Keras (CNN + LSTM)

Project Outline

- `Step 1:` **Download and Visualize the Data with its Labels**
- `Step 2:` **Preprocess the Dataset**
- `Step 3:` **Split the Data into Train and Test Set**
- `Step 4:` **Implement the ConvLSTM Approach**
  - `Step 4.1:` **Construct the Model**
  - `Step 4.2:` **Compile & Train the Model**

    - **Step 4.3: Plot Model's Loss & Accuracy Curves**
- **Step 5: implement the LRCN Approach**
    - **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 videos**

Import the Libraries

We will start by installing and importing the required libraries.

In [37]:
```
# %%capture
# # Discard the output of this cell
# # Install the required libraries.
# !pip install youtube-dl moviepy
# !pip install wget
# !pip install rarfile
# # !pip install git+https://github.com/TahaAnwar/pafy.git#egg=pafy
```

In [1]:
```
# Import the required libraries.
import os
import cv2
# import pafy
import wget
# import unrar
import math
import random
import numpy as np
import datetime as dt
import tensorflow as tf
from collections import deque
import matplotlib.pyplot as plt

from moviepy.editor import *
%matplotlib inline

from sklearn.model_selection import train_test_split

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
```

And will set `Numpy`, `Python`, and `Tensorflow` seeds to get consistent results on every execution.

In [39]:
```
seed_constant = 27
np.random.seed(seed_constant)
random.seed(seed_constant)
tf.random.set_seed(seed_constant)
```

Step 1: Download and Visualize the Data with its Labels ¶

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 - Action Recognition Dataset (https://www.crcv.ucf.edu/data/UCF50.php), consisting of realistic videos taken from youtube which differentiates this data set from most of the other available action recognition data sets as they are not realistic and are staged by actors. The Dataset contains:

- **50** Action Categories
- **25** Groups of Videos per Action Category
- **133** Average Videos per Action Category
- **199** Average Number of Frames per Video
- **320** Average Frames Width per Video
- **240** Average Frames Height per Video
- **26** Average Frames Per Seconds per Video

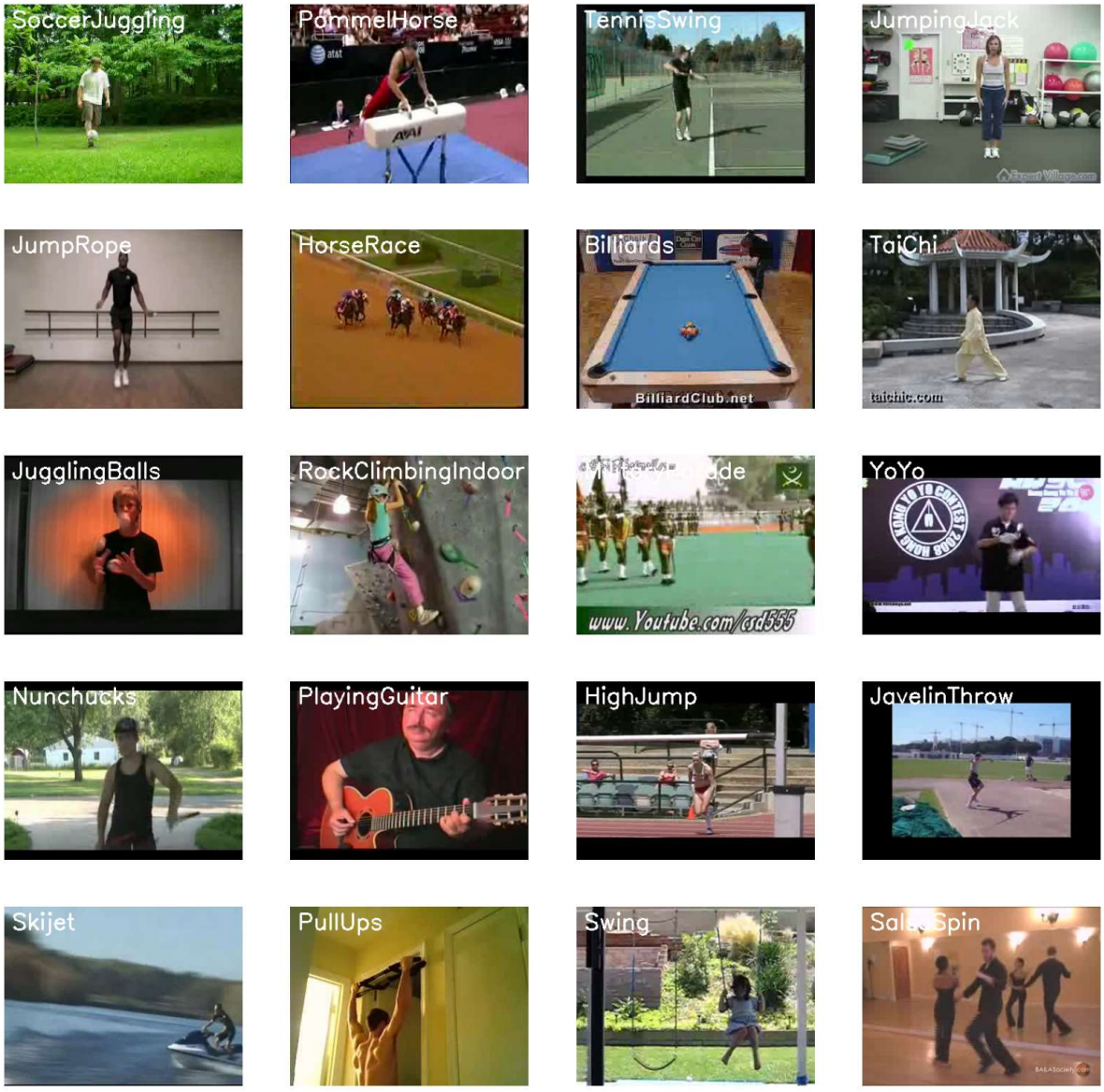
Download and extract the dataset.

In [40]:
```
# # Discard the output of this cell.
# # %%capture

# # Downlaod the UCF50 Dataset
# !python -m wget -o --no-check-certificate https://www.crcv.ucf.edu/data/UCF5
# # !wget --no-check-certificate https://www.crcv.ucf.edu/data/UCF50.rar
```

In [41]:
```
#Extract the Dataset
# !unrar x UCF50.rar
```

For visualization, we will pick `20` random categories from the dataset and a random video from each selected category and will visualize the first frame of the selected videos with their associated labels written. This way we'll be able to visualize a subset (`20` random videos) of the dataset.

In [42]:
```
# Create a Matplotlib figure and specify the size of the figure.
plt.figure(figsize = (20, 20))

# Get the names of all classes/categories in UCF50.
all_classes_names = os.listdir('UCF50')

# Generate a list of 20 random values. The values will be between 0-50,
# where 50 is the total number of class in the dataset.
random_range = random.sample(range(len(all_classes_names)), 20)

# Iterating through all the generated random values.
for counter, random_index in enumerate(random_range, 1):

    # Retrieve a Class Name using the Random Index.
    selected_class_Name = all_classes_names[random_index]

    # Retrieve the list of all the video files present in the randomly selecte
    video_files_names_list = os.listdir(f'UCF50/{selected_class_Name}')

    # Randomly select a video file from the list retrieved from the randomly s
    selected_video_file_name = random.choice(video_files_names_list)

    # Initialize a VideoCapture object to read from the video File.
    video_reader = cv2.VideoCapture(f'UCF50/{selected_class_Name}/{selected_vi

    # Read the first frame of the video file.
    _, bgr_frame = video_reader.read()

    # Release the VideoCapture object.
    video_reader.release()

    # Convert the frame from BGR into RGB format.
    rgb_frame = cv2.cvtColor(bgr_frame, cv2.COLOR_BGR2RGB)

    # Write the class name on the video frame.
    cv2.putText(rgb_frame, selected_class_Name, (10, 30), cv2.FONT_HERSHEY_SIM

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



Step 2: Preprocess the Dataset

Next, we will perform some preprocessing on the dataset. First, we will read the video files from the dataset and resize the frames of the videos to a fixed width and height, to reduce the computations and normalized the data to range `[0-1]` by dividing the pixel values with `255`, which makes convergence faster while training the network.

But first, let's initialize some constants.

In [43]:
```
# Specify the height and width to which each video frame will be resized in ou
IMAGE_HEIGHT , IMAGE_WIDTH = 64, 64

# Specify the number of frames of a video that will be fed to the model as one
SEQUENCE_LENGTH = 20

# Specify the directory containing the UCF50 dataset.
DATASET_DIR = "UCF50"

# Specify the list containing the names of the classes used for training. Feel
CLASSES_LIST = ["WalkingWithDog", "TaiChi", "Swing", "HorseRace", "MilitaryPar
```

Note: *The* ***`IMAGE_HEIGHT`*** *,* ***`IMAGE_WIDTH`*** *and* ***`SEQUENCE_LENGTH`*** *constants can be increased for better results, although increasing the sequence length is only effective to a certain point, and increasing the values will result in the process being more computationally expensive.*

Create a Function to Extract, Resize & Normalize Frames

We will create a function **`frames_extraction()`** that will create a list containing the resized and normalized frames of a video whose path is passed to it as an argument. The function will read the video file frame by frame, although not all frames are added to the list as we will only need an evenly distributed sequence length of frames.

In [44]:
```
def frames_extraction(video_path):
    '''
    This function will extract the required frames from a video after resizing
    Args:
        video_path: The path of the video in the disk, whose frames are to be
    Returns:
        frames_list: A list containing the resized and normalized frames of th
    '''

    # Declare a list to store video frames.
    frames_list = []

    # Read the Video File using the VideoCapture object.
    video_reader = cv2.VideoCapture(video_path)

    # Get the total number of frames in the video.
    video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate the the interval after which frames will be added to the list.
    skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH), 1)

    # Iterate through the Video Frames.
    for frame_counter in range(SEQUENCE_LENGTH):

        # Set the current frame position of the video.
        video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_

        # Reading the frame from the video.
        success, frame = video_reader.read()

        # Check if Video frame is not successfully read then break the loop
        if not success:
            break

        # Resize the Frame to fixed height and width.
        resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))

        # Normalize the resized frame by dividing it with 255 so that each pix
        normalized_frame = resized_frame / 255

        # Append the normalized frame into the frames list
        frames_list.append(normalized_frame)

    # Release the VideoCapture object.
    video_reader.release()

    # Return the frames list.
    return frames_list
```

Create a Function for Dataset Creation

Now we will create a function **`create_dataset()`** that will iterate through all the classes specified in the **`CLASSES_LIST`** constant and will call the function **`frame_extraction()`** on every video file of the selected classes and return the frames (**`features`**), class index (

labels), and video file path (**video_files_paths**).

In [45]:
```
def create_dataset():
    '''
    This function will extract the data of the selected classes and create the
    Returns:
        features:          A list containing the extracted frames of the video
        labels:            A list containing the indexes of the classes associ
        video_files_paths: A list containing the paths of the videos in the di
    '''

    # Declared Empty Lists to store the features, labels and video file path v
    features = []
    labels = []
    video_files_paths = []

    # Iterating through all the classes mentioned in the classes list
    for class_index, class_name in enumerate(CLASSES_LIST):

        # Display the name of the class whose data is being extracted.
        print(f'Extracting Data of Class: {class_name}')

        # Get the list of video files present in the specific class name direc
        files_list = os.listdir(os.path.join(DATASET_DIR, class_name))

        # Iterate through all the files present in the files list.
        for file_name in files_list:

            # Get the complete video path.
            video_file_path = os.path.join(DATASET_DIR, class_name, file_name)

            # Extract the frames of the video file.
            frames = frames_extraction(video_file_path)

            # Check if the extracted frames are equal to the SEQUENCE_LENGTH s
            # So ignore the vides having frames less than the SEQUENCE_LENGTH.
            if len(frames) == SEQUENCE_LENGTH:

                # Append the data to their repective lists.
                features.append(frames)
                labels.append(class_index)
                video_files_paths.append(video_file_path)

    # Converting the list to numpy arrays
    features = np.asarray(features)
    labels = np.array(labels)

    # Return the frames, class index, and video file path.
    return features, labels, video_files_paths
```

Now we will utilize the function **create_dataset()** created above to extract the data of the selected classes and create the required dataset.

In [46]:
```
# Create the dataset.
features, labels, video_files_paths = create_dataset()
```

```
Extracting Data of Class: WalkingWithDog
Extracting Data of Class: TaiChi
Extracting Data of Class: Swing
Extracting Data of Class: HorseRace
Extracting Data of Class: MilitaryParade
```

Now we will convert `labels` (class indexes) into one-hot encoded vectors.

In [47]:
```
# Using Keras's to_categorical method to convert labels into one-hot-encoded v
one_hot_encoded_labels = to_categorical(labels)
```

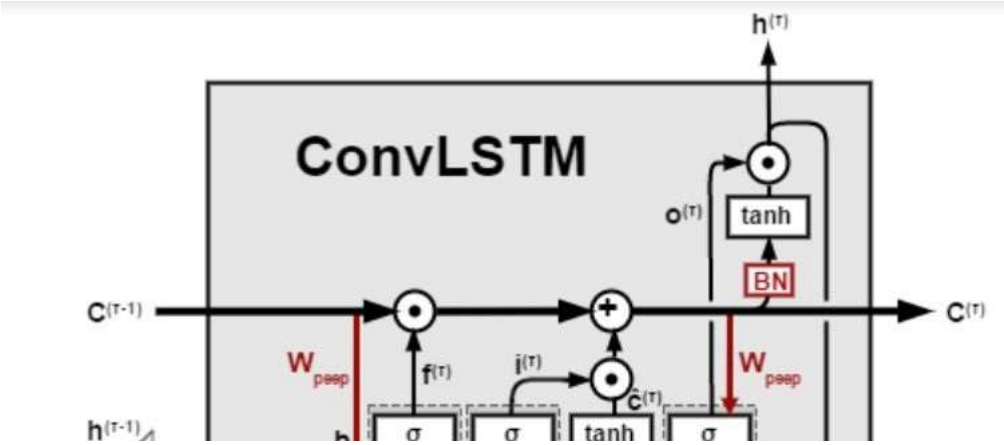
Step 3: Split the Data into Train and Test Set

As of now, we have the required **`features`** (a NumPy array containing all the extracted frames of the videos) and **`one_hot_encoded_labels`** (also a Numpy array containing all class labels in one hot encoded format). So now, we will split our data to create training and testing sets. We will also shuffle the dataset before the split to avoid any bias and get splits representing the overall distribution of the data.

In [48]:
```
# Split the Data into Train ( 75% ) and Test Set ( 25% ).
features_train, features_test, labels_train, labels_test = train_test_split(fe
                                                                             te
                                                                             ra
```

Step 4: Implement the ConvLSTM Approach

In this step, we will implement the first approach by using a combination of ConvLSTM cells. A ConvLSTM cell is a variant of an LSTM network that contains convolutions operations in the network. it is an LSTM with convolution embedded in the architecture, which makes it capable of identifying spatial features of the data while keeping into account the temporal relation.



Step 4.1: Construct the Model

To construct the model, we will use Keras ConvLSTM2D recurrent layers. The `ConvLSTM2D` layer also takes in the number of filters and kernel size required for applying the convolutional operations. The output of the layers is flattened in the end and is fed to the `Dense` layer with softmax activation which outputs the probability of each action category.

We will also use `MaxPooling3D` layers to reduce the dimensions of the frames and avoid unnecessary computations and `Dropout` layers to prevent overfitting the model on the data. The architecture is a simple one and has a small number of trainable parameters. This is because we are only dealing with a small subset of the dataset which does not require a large-scale model.

In [49]:
```
def create_convlstm_model():
    '''
    This function will construct the required convlstm model.
    Returns:
        model: It is the required constructed convlstm model.
    '''

    # We will use a Sequential model for model construction
    model = Sequential()

    # Define the Model Architecture.
    ########################################################################

    model.add(ConvLSTM2D(filters = 4, kernel_size = (3, 3), activation = 'tanh
                         recurrent_dropout=0.2, return_sequences=True, input_s


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

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

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

    model.add(ConvLSTM2D(filters = 14, kernel_size = (3, 3), activation = 'tan
                         recurrent_dropout=0.2, return_sequences=True))

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

    model.add(ConvLSTM2D(filters = 16, kernel_size = (3, 3), activation = 'tan
                         recurrent_dropout=0.2, return_sequences=True))

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

    model.add(Flatten())

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

    ########################################################################

    # Display the models summary.
    model.summary()

    # Return the constructed convlstm model.
    return model
```

Now we will utilize the function **`create_convlstm_model()`** created above, to construct the required `convlstm` model.

In [50]:
```
# Construct the required convlstm model.
convlstm_model = create_convlstm_model()

# Display the success message.
print("Model Created Successfully!")
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_lstm2d_4 (ConvLSTM2D)  (None, 20, 62, 62, 4)     1024

 max_pooling3d_4 (MaxPooling  (None, 20, 31, 31, 4)    0
 3D)

 time_distributed_15 (TimeDi  (None, 20, 31, 31, 4)    0
 stributed)

 conv_lstm2d_5 (ConvLSTM2D)  (None, 20, 29, 29, 8)     3488

 max_pooling3d_5 (MaxPooling  (None, 20, 15, 15, 8)    0
 3D)

 time_distributed_16 (TimeDi  (None, 20, 15, 15, 8)    0
 stributed)

 conv_lstm2d_6 (ConvLSTM2D)  (None, 20, 13, 13, 14)    11144

 max_pooling3d_6 (MaxPooling  (None, 20, 7, 7, 14)     0
 3D)

 time_distributed_17 (TimeDi  (None, 20, 7, 7, 14)     0
 stributed)

 conv_lstm2d_7 (ConvLSTM2D)  (None, 20, 5, 5, 16)      17344

 max_pooling3d_7 (MaxPooling  (None, 20, 3, 3, 16)     0
 3D)

 flatten_2 (Flatten)         (None, 2880)              0

 dense_2 (Dense)             (None, 5)                 14405

=================================================================
Total params: 47,405
Trainable params: 47,405
Non-trainable params: 0
_________________________________________________________________
Model Created Successfully!
```

Check Model's Structure:

In [51]:
```
!pip install pydot
```

```
Requirement already satisfied: pydot in c:\programdata\anaconda3\lib\site-pac
kages (1.4.2)
Requirement already satisfied: pyparsing>=2.1.4 in c:\programdata\anaconda3\l
ib\site-packages (from pydot) (3.0.9)
```

In [52]:
```
# Plot the structure of the contructed model.
plot_model(convlstm_model, to_file = 'convlstm_model_structure_plot.png', show
```

```
You must install pydot (`pip install pydot`) and install graphviz (see instru
ctions at https://graphviz.gitlab.io/download/) (https://graphviz.gitlab.io/d
ownload/)) for plot_model to work.
```

Step 4.2: Compile & Train the Model

Next, we will add an early stopping callback to prevent overfitting (https://en.wikipedia.org/wiki/Overfitting) and start the training after compiling the model.

In [53]:
```
# Create an Instance of Early Stopping Callback
early_stopping_callback = EarlyStopping(monitor = 'val_loss', patience = 10, m
callbacks1=[tf.keras.callbacks.ModelCheckpoint(filepath='my_model', save_best_
# Compile the model and specify loss function, optimizer and metrics values to
convlstm_model.compile(loss = 'categorical_crossentropy', optimizer = 'Adam',

# Start training the model.
convlstm_model_training_history = convlstm_model.fit(x = features_train, y = l
                                                     shuffle = True, validatic
                                                     callbacks = callbacks1)
```

```
Epoch 1/200
92/92 [==============================] - ETA: 0s - loss: 1.5991 - accurac
y: 0.2745

WARNING:absl:Found untraced functions such as _update_step_xla while savin
g (showing 1 of 1). These functions will not be directly callable after lo
ading.

INFO:tensorflow:Assets written to: my_model\assets

INFO:tensorflow:Assets written to: my_model\assets

92/92 [==============================] - 188s 2s/step - loss: 1.5991 - acc
uracy: 0.2745 - val_loss: 1.5529 - val_accuracy: 0.2391
Epoch 2/200
92/92 [==============================] - ETA: 0s - loss: 1.3629 - accurac
y: 0.4266

WARNING:absl:Found untraced functions such as _update_step_xla while savin
g (showing 1 of 1). These functions will not be directly callable after lo
ading
```

Evaluate the Trained Model

After training, we will evaluate the model on the test set.

In [57]:
```
# Evaluate the trained model.
model_evaluation_history = convlstm_model.evaluate(features_test, la



                                                   bels_test)
```

```
5/5 [==============================] - 9s 2s/step - loss: 1.1872 - accuracy:
0.7273
```

Save the Model

Now we will save the model to avoid training it from scratch every time we need the model.

In [58]:
```
# Get the loss and accuracy from model_evaluation_history.
model_evaluation_loss, model_evaluation_accuracy = model_evaluation_history

# Define the string date format.
# Get the current Date and Time in a DateTime Object.
# Convert the DateTime object to string according to the style mentioned in da
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_tim

# Define a useful name for our model to make it easy for us while navigating t
model_file_name = f'convlstm_model___Date_Time_{current_date_time_string}___Lo

# Save your Model.
convlstm_model.save(model_file_name)
```

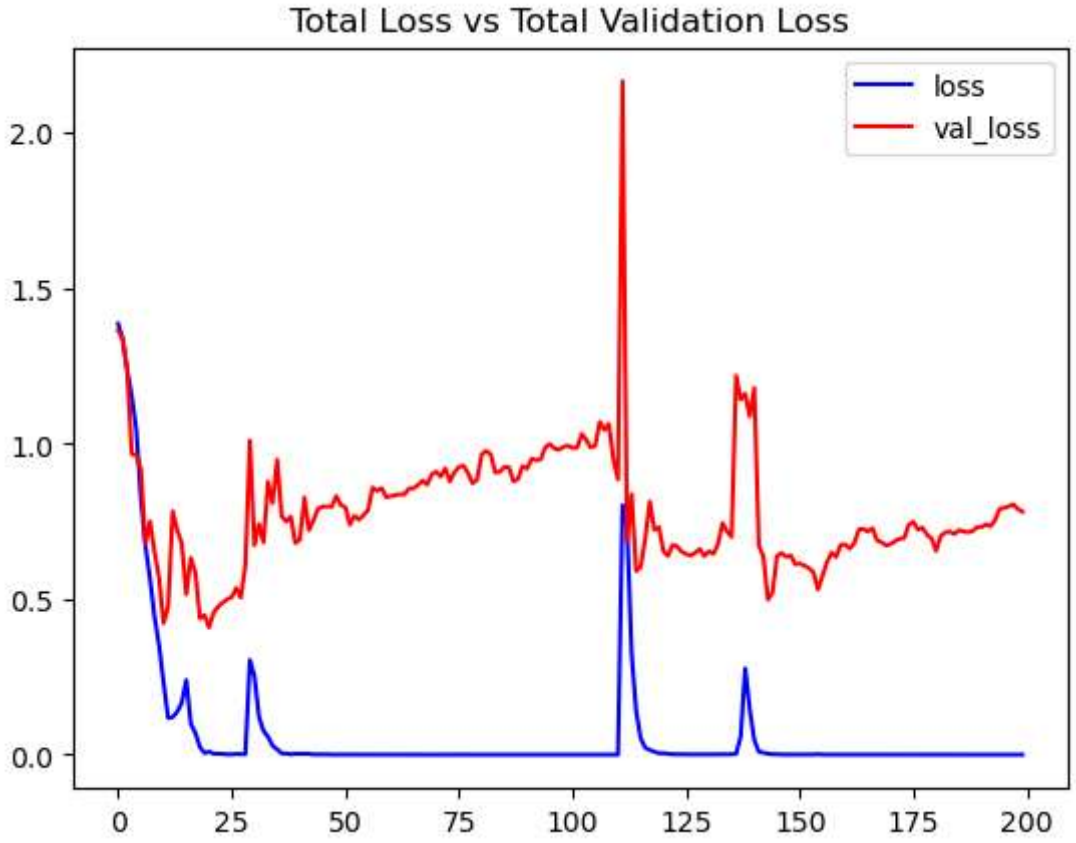
Step 4.3: Plot Model's Loss & Accuracy Curves

Now we will create a function **`plot_metric()`** to visualize the training and validation metrics. We already have separate metrics from our training and validation steps so now we just have to visualize them.

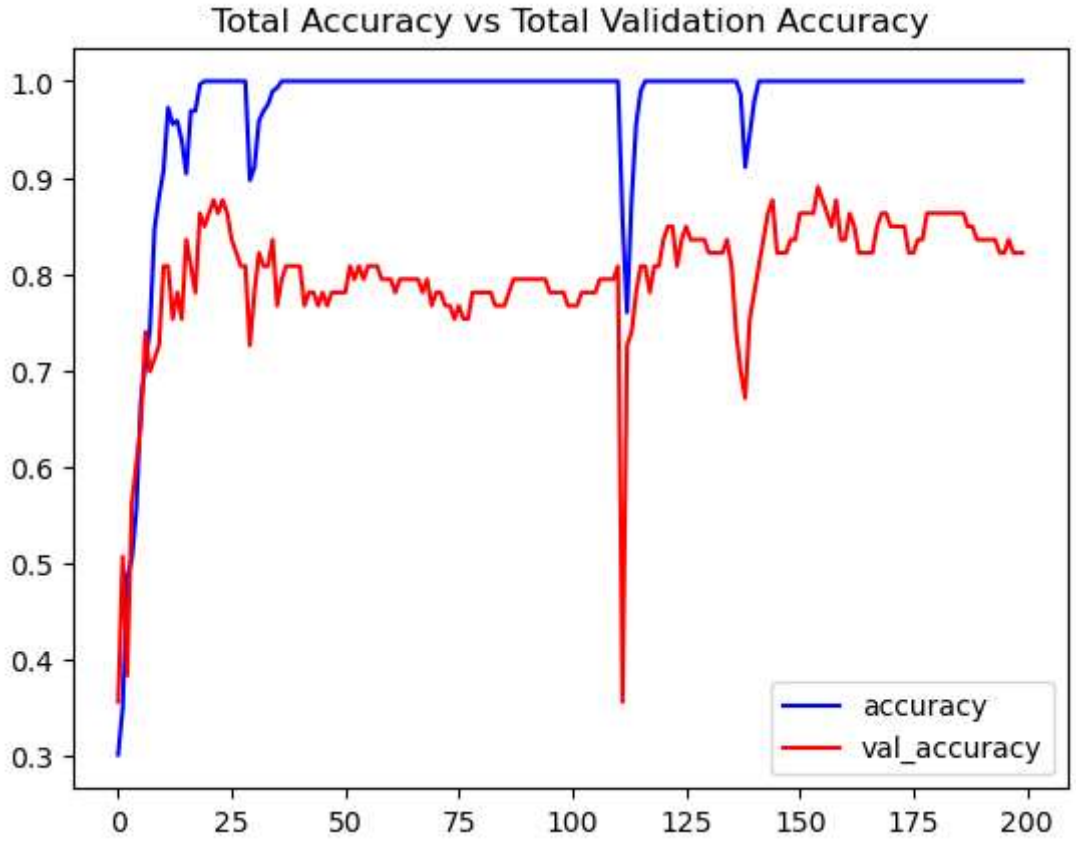
In [59]:
```
def plot_metric(model_training_history, metric_name_1, metric_name_2, plot_nam
    '''
    This function will plot the metrics passed to it in a graph.
    Args:
        model_training_history: A history object containing a record of traini
                                loss values and metrics values at successive e
        metric_name_1:          The name of the first metric that needs to be
        metric_name_2:          The name of the second metric that needs to be
        plot_name:              The title of the graph.
    '''

    # Get metric values using metric names as identifiers.
    metric_value_1 = model_training_history.history[metric_name_1]
    metric_value_2 = model_training_history.history[metric_name_2]

    # Construct a range object which will be used as x-axis (horizontal plane)
    epochs = range(len(metric_value_1))

    # Plot the Graph.
    plt.plot(epochs, metric_value_1, 'blue', label = metric_name_1)
    plt.plot(epochs, metric_value_2, 'red', label = metric_name_2)

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

    # Add legend to the plot.
    plt.legend()
```

Now we will utilize the function **`plot_metric()`** created above, to visualize and understand the metrics.

In [60]:
```
# Visualize the training and validation loss metrices.
plot_metric(convlstm_model_training_history, 'loss', 'val_loss', 'Total Loss v
```



In [61]:
```
# Visualize the training and validation accuracy metrices.
plot_metric(convlstm_model_training_history, 'accuracy', 'val_accuracy', 'Tota
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

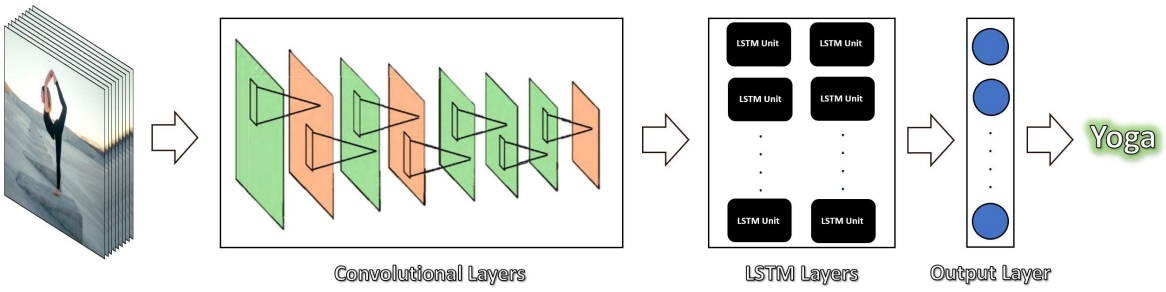
In [ ]:

In [ ]:

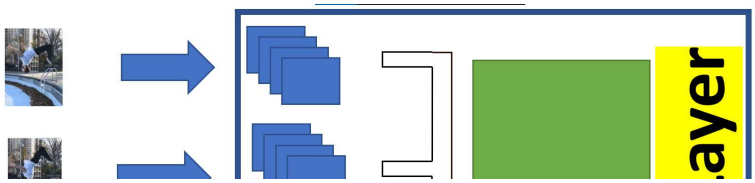
Step 5: Implement the LRCN Approach

In this step, we will implement the LRCN Approach by combining Convolution and LSTM layers in a single model. Another similar approach can be to use a CNN model and LSTM model trained separately. The CNN model can be used to extract spatial features from the frames in the video, and for this purpose, a pre-trained model can be used, that can be fine-tuned for the problem. And the LSTM model can then use the features extracted by CNN, to predict the action being performed in the video.

But here, we will implement another approach known as the Long-term Recurrent Convolutional Network (LRCN), which combines CNN and LSTM layers in a single model. The Convolutional layers are used for spatial feature extraction from the frames, and the extracted spatial features are fed to LSTM layer(s) at each time-steps for temporal sequence modeling. This way the network learns spatiotemporal features directly in an end-to-end training, resulting in a robust model.



We will also use TimeDistributed wrapper layer, which allows applying the same layer to every frame of the video independently. So it makes a layer (around which it is wrapped) capable of taking input of shape `(no_of_frames, width, height, num_of_channels)` if originally the layer's input shape was `(width, height, num_of_channels)` which is very beneficial as it allows to input the whole video into the model in a single shot.



Step 5.1: Construct the Model

To implement our LRCN architecture, we will use time-distributed **`Conv2D`** layers which will be followed by **`MaxPooling2D`** and **`Dropout`** layers. The feature extracted from the **`Conv2D`** layers will be then flattened using the **`Flatten`** layer and will be fed to a **`LSTM`** layer. The **`Dense`** layer with softmax activation will then use the output from the **`LSTM`** layer to predict the action being performed.

In [62]:
```
def create_LRCN_model():
    '''
    This function will construct the required LRCN model.
    Returns:
        model: It is the required constructed LRCN model.
    '''

    # We will use a Sequential model for model construction.
    model = Sequential()

    # Define the Model Architecture.
    ########################################################################

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

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

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

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

    model.add(TimeDistributed(Conv2D(64, (3, 3), padding='same',activation = '
    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'))

    ########################################################################

    # Display the models summary.
    model.summary()

    # Return the constructed LRCN model.
    return model
```

Now we will utilize the function **`create_LRCN_model()`** created above to construct the required `LRCN` model.

In [63]:
```
# Construct the required LRCN model.
LRCN_model = create_LRCN_model()

# Display the success message.
print("Model Created Successfully!")
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 time_distributed_18 (TimeDi  (None, 20, 64, 64, 16)   448
 stributed)

 time_distributed_19 (TimeDi  (None, 20, 16, 16, 16)   0
 stributed)

 time_distributed_20 (TimeDi  (None, 20, 16, 16, 16)   0
 stributed)

 time_distributed_21 (TimeDi  (None, 20, 16, 16, 32)   4640
 stributed)

 time_distributed_22 (TimeDi  (None, 20, 4, 4, 32)     0
 stributed)

 time_distributed_23 (TimeDi  (None, 20, 4, 4, 32)     0
 stributed)

 time_distributed_24 (TimeDi  (None, 20, 4, 4, 64)     18496
 stributed)

 time_distributed_25 (TimeDi  (None, 20, 2, 2, 64)     0
 stributed)

 time_distributed_26 (TimeDi  (None, 20, 2, 2, 64)     0
 stributed)

 time_distributed_27 (TimeDi  (None, 20, 2, 2, 64)     36928
 stributed)

 time_distributed_28 (TimeDi  (None, 20, 1, 1, 64)     0
 stributed)

 time_distributed_29 (TimeDi  (None, 20, 64)           0
 stributed)

 lstm_1 (LSTM)               (None, 32)                12416

 dense_3 (Dense)             (None, 5)                 165

=================================================================
Total params: 73,093
Trainable params: 73,093
Non-trainable params: 0
_________________________________________________________________
Model Created Successfully!
```

Check Model's Structure:

Now we will use the **`plot_model()`** function to check the structure of the constructed `LRCN` model. As we had checked for the previous model.

In [64]:
```
# Plot the structure of the contructed LRCN model.
plot_model(LRCN_model, to_file = 'LRCN_model_structure_plot.png', show_shapes
```

```
You must install pydot (`pip install pydot`) and install graphviz (see instru
ctions at https://graphviz.gitlab.io/download/) (https://graphviz.gitlab.io/d
ownload/)) for plot_model to work.
```

Step 5.2: Compile & Train the Model

After checking the structure, we will compile and start training the model.

In [74]:
```
# Create an Instance of Early Stopping Callback.
early_stopping_callback = EarlyStopping(monitor = 'val_loss', patience = 15, m

# Compile the model and specify loss function, optimizer and metrics to the mo
LRCN_model.compile(loss = 'categorical_crossentropy', optimizer = 'Adam', metr

# Start training the model.
LRCN_model_training_history = LRCN_model.fit(x = features_train, y = labels_tr
                                             shuffle = True, validation_split
```

```
Epoch 1/70
92/92 [==============================] - 13s 121ms/step - loss: 0.2258 - accu
racy: 0.9158 - val_loss: 0.5244 - val_accuracy: 0.8261
Epoch 2/70
92/92 [==============================] - 14s 147ms/step - loss: 0.1892 - accu
racy: 0.9348 - val_loss: 0.6325 - val_accuracy: 0.7500
Epoch 3/70
92/92 [==============================] - 14s 149ms/step - loss: 0.1916 - accu
racy: 0.9375 - val_loss: 0.5726 - val_accuracy: 0.8261
Epoch 4/70
92/92 [==============================] - 14s 152ms/step - loss: 0.2074 - accu
racy: 0.9266 - val_loss: 0.6893 - val_accuracy: 0.7391
Epoch 5/70
92/92 [==============================] - 14s 149ms/step - loss: 0.2366 - accu
racy: 0.9185 - val_loss: 0.4593 - val_accuracy: 0.8804
Epoch 6/70
92/92 [==============================] - 14s 150ms/step - loss: 0.1674 - accu
racy: 0.9457 - val_loss: 0.4328 - val_accuracy: 0.8370
Epoch 7/70
92/92 [==============================] - 14s 149ms/step - loss: 0.0947 - accu
racy: 0.9701 - val_loss: 0.4229 - val_accuracy: 0.8478
Epoch 8/70
92/92 [==============================] - 14s 149ms/step - loss: 0.0889 - accu
racy: 0.9674 - val_loss: 0.4569 - val_accuracy: 0.8696
Epoch 9/70
92/92 [==============================] - 14s 150ms/step - loss: 0.0525 - accu
racy: 0.9783 - val_loss: 0.3657 - val_accuracy: 0.8913
Epoch 10/70
92/92 [==============================] - 14s 150ms/step - loss: 0.2000 - accu
racy: 0.9293 - val_loss: 0.5723 - val_accuracy: 0.7717
Epoch 11/70
92/92 [==============================] - 14s 150ms/step - loss: 0.0773 - accu
racy: 0.9728 - val_loss: 0.4652 - val_accuracy: 0.8587
Epoch 12/70
92/92 [==============================] - 14s 151ms/step - loss: 0.0950 - accu
racy: 0.9647 - val_loss: 0.4386 - val_accuracy: 0.8696
Epoch 13/70
92/92 [==============================] - 14s 153ms/step - loss: 0.0936 - accu
racy: 0.9783 - val_loss: 0.6341 - val_accuracy: 0.8261
Epoch 14/70
92/92 [==============================] - 14s 151ms/step - loss: 0.0210 - accu
racy: 1.0000 - val_loss: 0.4164 - val_accuracy: 0.8913
Epoch 15/70
92/92 [==============================] - 14s 151ms/step - loss: 0.1156 - accu
racy: 0.9674 - val_loss: 1.1800 - val_accuracy: 0.7500
Epoch 16/70
92/92 [==============================] - 14s 151ms/step - loss: 0.1348 - accu
racy: 0.9511 - val_loss: 0.4398 - val_accuracy: 0.8587
Epoch 17/70
92/92 [==============================] - 14s 151ms/step - loss: 0.1317 - accu
racy: 0.9429 - val_loss: 0.8156 - val_accuracy: 0.7826
Epoch 18/70
92/92 [==============================] - 14s 151ms/step - loss: 0.1377 - accu
racy: 0.9511 - val_loss: 0.2186 - val_accuracy: 0.9130
Epoch 19/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0484 - accu
racy: 0.9891 - val_loss: 0.5043 - val_accuracy: 0.8587
```

```
Epoch 20/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0257 - accu
racy: 0.9973 - val_loss: 0.3406 - val_accuracy: 0.8913
Epoch 21/70
92/92 [==============================] - 14s 156ms/step - loss: 0.0087 - accu
racy: 1.0000 - val_loss: 0.4238 - val_accuracy: 0.8696
Epoch 22/70
92/92 [==============================] - 14s 151ms/step - loss: 0.0073 - accu
racy: 1.0000 - val_loss: 0.3617 - val_accuracy: 0.9022
Epoch 23/70
92/92 [==============================] - 14s 151ms/step - loss: 0.0070 - accu
racy: 1.0000 - val_loss: 0.5202 - val_accuracy: 0.8696
Epoch 24/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0048 - accu
racy: 1.0000 - val_loss: 0.4952 - val_accuracy: 0.8696
Epoch 25/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0042 - accu
racy: 1.0000 - val_loss: 0.4750 - val_accuracy: 0.8696
Epoch 26/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0115 - accu
racy: 0.9973 - val_loss: 0.4101 - val_accuracy: 0.9022
Epoch 27/70
92/92 [==============================] - 14s 152ms/step - loss: 0.5416 - accu
racy: 0.8478 - val_loss: 0.7871 - val_accuracy: 0.7174
Epoch 28/70
92/92 [==============================] - 14s 152ms/step - loss: 0.1842 - accu
racy: 0.9375 - val_loss: 0.5739 - val_accuracy: 0.8587
Epoch 29/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0554 - accu
racy: 0.9918 - val_loss: 0.5371 - val_accuracy: 0.8587
Epoch 30/70
92/92 [==============================] - 14s 154ms/step - loss: 0.0338 - accu
racy: 0.9891 - val_loss: 0.6562 - val_accuracy: 0.8478
Epoch 31/70
92/92 [==============================] - 14s 151ms/step - loss: 0.0429 - accu
racy: 0.9891 - val_loss: 0.5371 - val_accuracy: 0.8696
Epoch 32/70
92/92 [==============================] - 14s 152ms/step - loss: 0.0255 - accu
racy: 0.9946 - val_loss: 0.4208 - val_accuracy: 0.8804
Epoch 33/70
92/92 [==============================] - 14s 153ms/step - loss: 0.0096 - accu
racy: 1.0000 - val_loss: 0.4498 - val_accuracy: 0.8696
```

Evaluating the trained Model

As done for the previous one, we will evaluate the `LRCN` model on the test set.

In [75]:
```
# Evaluate the trained model.
model_evaluation_history = LRCN_model.evaluate(features_test, labels_test)
```

```
5/5 [==============================] - 1s 143ms/step - loss: 0.3205 - accurac
y: 0.9026
```

Save the Model

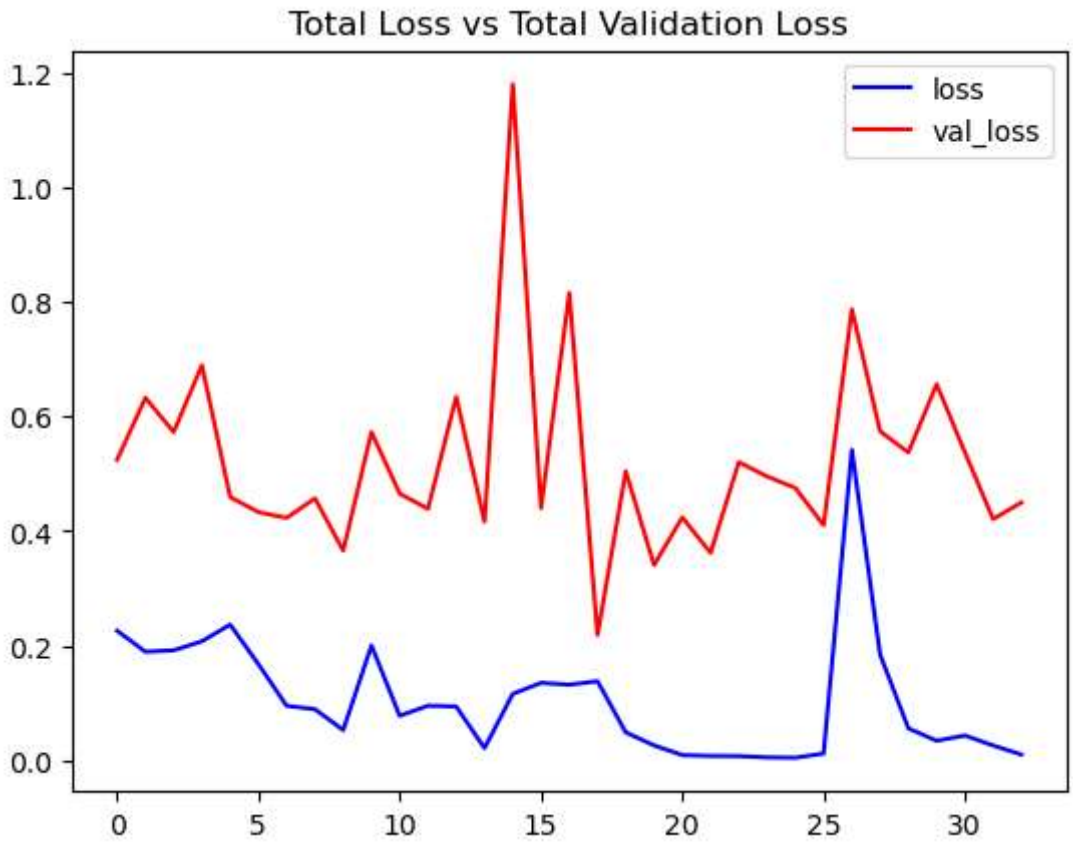
After that, we will save the model for future uses using the same technique we had used for the previous model.

In [76]:
```
# Get the loss and accuracy from model_evaluation_history.
model_evaluation_loss, model_evaluation_accuracy = model_evaluation_history

# Define the string date format.
# Get the current Date and Time in a DateTime Object.
# Convert the DateTime object to string according to the style mentioned in da
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_tim

# Define a useful name for our model to make it easy for us while navigating t
model_file_name = f'LRCN_model___Date_Time_{current_date_time_string}___Loss_{

# Save the Model.
LRCN_model.save(model_file_name)
```

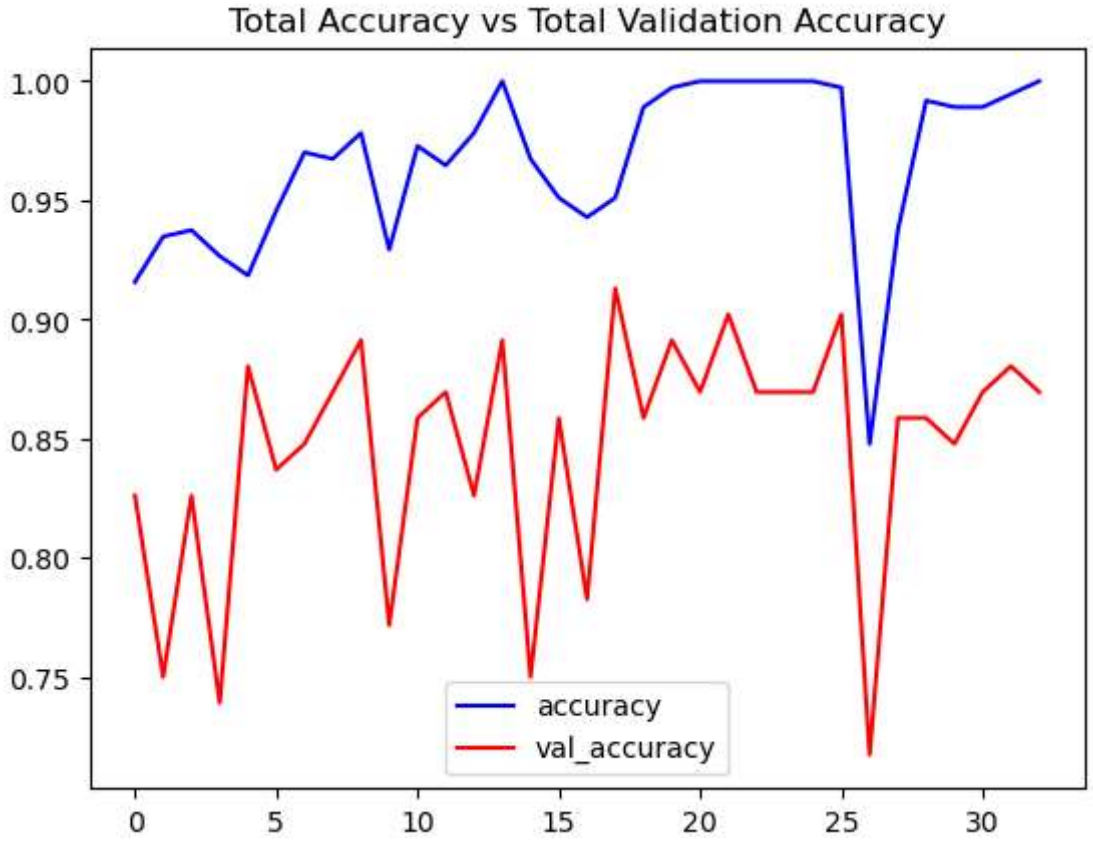
Step 5.3: Plot Model's Loss & Accuracy Curves

Now we will utilize the function **`plot_metric()`** we had created above to visualize the training and validation metrics of this model.

In [77]:
```
# Visualize the training and validation loss metrices.
plot_metric(LRCN_model_training_history, 'loss', 'val_loss', 'Total Loss vs To
```



In [78]:
```
# Visualize the training and validation accuracy metrices.
plot_metric(LRCN_model_training_history, 'accuracy', 'val_accuracy', 'Total Ac
```



Step 6: Test the Best Performing Model on YouTube videos

From the results, it seems that the LRCN model performed significantly well for a small number of classes. so in this step, we will put the `LRCN` model to test on some youtube videos.

Create a Function to Download YouTube Videos:

We will create a function **`download_youtube_videos()`** to download the YouTube videos first using **`pafy`** library. The library only requires a URL to a video to download it along with its associated metadata like the title of the video.

In [79]:
```
def download_youtube_videos(youtube_video_url, output_directory):
    '''
    This function downloads the youtube video whose URL is passed to it as an
    Args:
        youtube_video_url: URL of the video that is required to be downloaded.
        output_directory:  The directory path to which the video needs to be s
    Returns:
        title: The title of the downloaded youtube video.
    '''

    # Create a video object which contains useful information about the video
    video = pafy.new(youtube_video_url)

    # Retrieve the title of the video.
    title = video.title

    # Get the best available quality object for the video.
    video_best = video.getbest()

    # Construct the output file path.
    output_file_path = f'{output_directory}/{title}.mp4'

    # Download the youtube video at the best available quality and store it t
    video_best.download(filepath = output_file_path, quiet = True)

    # Return the video title.
    return title
```

In [ ]:
```
```

Create a Function To Perform Action Recognition on Videos

Next, we will create a function **`predict_on_video()`** that will simply read a video frame by frame from the path passed in as an argument and will perform action recognition on video and save the results.

In [80]:
```
def predict_on_video(video_file_path, output_file_path, SEQUENCE_LENGTH):
    '''
    This function will perform action recognition on a video using the LRCN mo
    Args:
    video_file_path:  The path of the video stored in the disk on which the ac
    output_file_path: The path where the ouput video with the predicted action
    SEQUENCE_LENGTH:  The fixed number of frames of a video that can be passed
    '''

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

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

    # Initialize the VideoWriter Object to store the output video in the disk.
    video_writer = cv2.VideoWriter(output_file_path, cv2.VideoWriter_fourcc('M
                                   video_reader.get(cv2.CAP_PROP_FPS), (origin

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

    # Initialize a variable to store the predicted action being performed in t
    predicted_class_name = ''

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

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

        # Check if frame is not read properly then break the loop.
        if not ok:
            break

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

        # Normalize the resized frame by dividing it with 255 so that each pix
        normalized_frame = resized_frame / 255

        # Appending the pre-processed frame into the frames list.
        frames_queue.append(normalized_frame)

        # Check if the number of frames in the queue are equal to the fixed se
        if len(frames_queue) == SEQUENCE_LENGTH:

            # Pass the normalized frames to the model and get the predicted pr
            # convlstm_model
            predicted_labels_probabilities = LRCN_model.predict(np.expand_dims

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

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

```
        # Write predicted class name on top of the frame.
        cv2.putText(frame, predicted_class_name, (10, 30), cv2.FONT_HERSHEY_SI

        # Write The frame into the disk using the VideoWriter Object.
        video_writer.write(frame)

    # Release the VideoCapture and VideoWriter objects.
    video_reader.release()
    video_writer.release()
```

Perform Action Recognition on the Test Video

Now we will utilize the function **`predict_on_video()`** created above to perform action recognition on the test video we had downloaded using the function **`download_youtube_videos()`** and display the output video with the predicted action overlayed on it.

In [81]:
```
# Construct the output video path.

input_video_file_path = f'ArmyParade.mp4'
output_video_file_path = f'ArmyParade1_Output.mp4'

# Perform Action Recognition on the Test Video.
predict_on_video(input_video_file_path, output_video_file_path, SEQUENCE_LENGT

# Display the output video.
VideoFileClip(output_video_file_path, audio=False, target_resolution=(300,None
```

```
1/1 [==============================] - 0s 381ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 32ms/step
```

In [ ]:

In [ ]:
```
# Construct the output video path.
output_video_file_path = f'TestVideo_Output.mp4'

# Perform Action Recognition on the Test Video.
predict_on_video(input_video_file_path, output_video_file_path, SEQUENCE_LENGT

# Display the output video.
VideoFileClip(output_video_file_path, audio=False, target_resolution=(300,None
```

Create a Function To Perform a Single Prediction on Videos

Now let's create a function that will perform a single prediction for the complete videos. We will extract evenly distributed **N `(SEQUENCE_LENGTH)`** frames from the entire video and pass them to the `LRCN` model. This approach is really useful when you are working with videos containing only one activity as it saves unnecessary computations and time in that scenario.

In [ ]:
```
def predict_single_action(video_file_path, SEQUENCE_LENGTH):
    '''
    This function will perform single action recognition prediction on a video
    Args:
    video_file_path:  The path of the video stored in the disk on which the ac
    SEQUENCE_LENGTH:  The fixed number of frames of a video that can be passed
    '''

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

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

    # Declare a list to store video frames we will extract.
    frames_list = []

    # Initialize a variable to store the predicted action being performed in t
    predicted_class_name = ''

    # Get the number of frames in the video.
    video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate the interval after which frames will be added to the list.
    skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH),1)

    # Iterating the number of times equal to the fixed length of sequence.
    for frame_counter in range(SEQUENCE_LENGTH):

        # Set the current frame position of the video.
        video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_

        # Read a frame.
        success, frame = video_reader.read()

        # Check if frame is not read properly then break the loop.
        if not success:
            break

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

        # Normalize the resized frame by dividing it with 255 so that each pix
        normalized_frame = resized_frame / 255

        # Appending the pre-processed frame into the frames list
        frames_list.append(normalized_frame)

    # Passing the  pre-processed frames to the model and get the predicted pro
    predicted_labels_probabilities = LRCN_model.predict(np.expand_dims(frames_

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

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

```
    # Display the predicted action along with the prediction confidence.
    print(f'Action Predicted: {predicted_class_name}\nConfidence: {predicted_l

    # Release the VideoCapture object.
    video_reader.release()
```

Perform Single Prediction on a Test Video

Now we will utilize the function **`predict_single_action()`** created above to perform a single prediction on a complete youtube test video that we will download using the function **`download_youtube_videos()`** , we had created above.

In [ ]:
```
# Download the youtube video.
video_title = download_youtube_videos('https://youtu.be/fc3w827kwyA', test_vid

# Construct tihe nput youtube video path
input_video_file_path = f'{test_videos_directory}/{video_title}.mp4'

# Perform Single Prediction on the Test Video.
predict_single_action(input_video_file_path, SEQUENCE_LENGTH)

# Display the input video.
VideoFileClip(input_video_file_path, audio=False, target_resolution=(300,None)
```

In [ ]:
```

```

In [ ]:
```

```