

Human Activity Recognition using TensorFlow (CNN + LSTM)

CONVLSTM / RCNN

By Taha Anwar, Rizwan Naeem and Momin Anjum On September 24, 2021

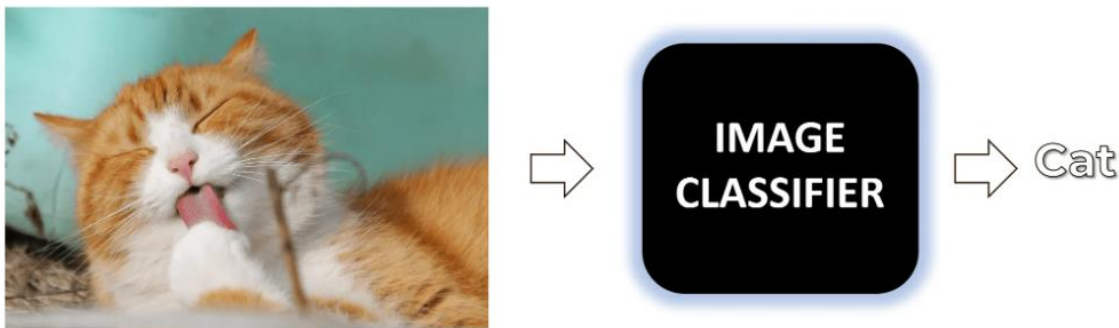
Convolutional Neural Networks (CNN) are great for image data and Long-Short Term Memory (LSTM) networks are great when working with sequence data but when you combine both of them, you get the best of both worlds and you solve difficult computer vision problems like video classification.

In this tutorial, we'll learn to implement human action recognition on videos using a Convolutional Neural Network combined with a Long-Short Term Memory Network. We'll actually be using two different architectures and approaches in TensorFlow to do this. In the end, we'll take the best-performing model and perform predictions with it on YouTube videos.

Before I start with the code, let me cover some theories on video classification and different approaches that are available for it.

Image Classification

You may already be familiar with an image classification problem, where, you simply pass an image to the classifier (either a trained Deep Neural Network (CNN or an MLP) or a classical classifier) and get the class predictions out of it.



But what if you have a video? What will happen then?



Before we talk about how to go about dealing with videos, let's just discuss what videos are exactly.

But First What Exactly Videos are?

Well, so it's no secret that a video is just a sequence of multiple still images (aka. frames) that are updated really fast creating the appearance of a motion. Consider the

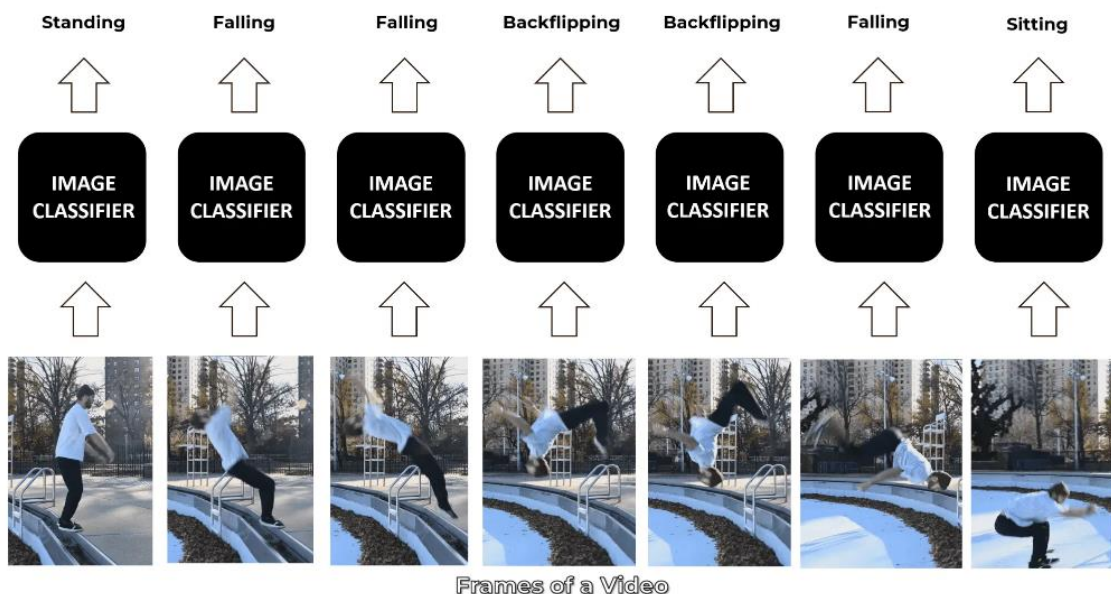
video (converted into .gif format) below of a cat jumping on a bookshelf, it is just a combination of 15 different still images that are being updated one after the other.



Now that we understand what videos are, let's take a look at a number of approaches that we can use to do video classification.

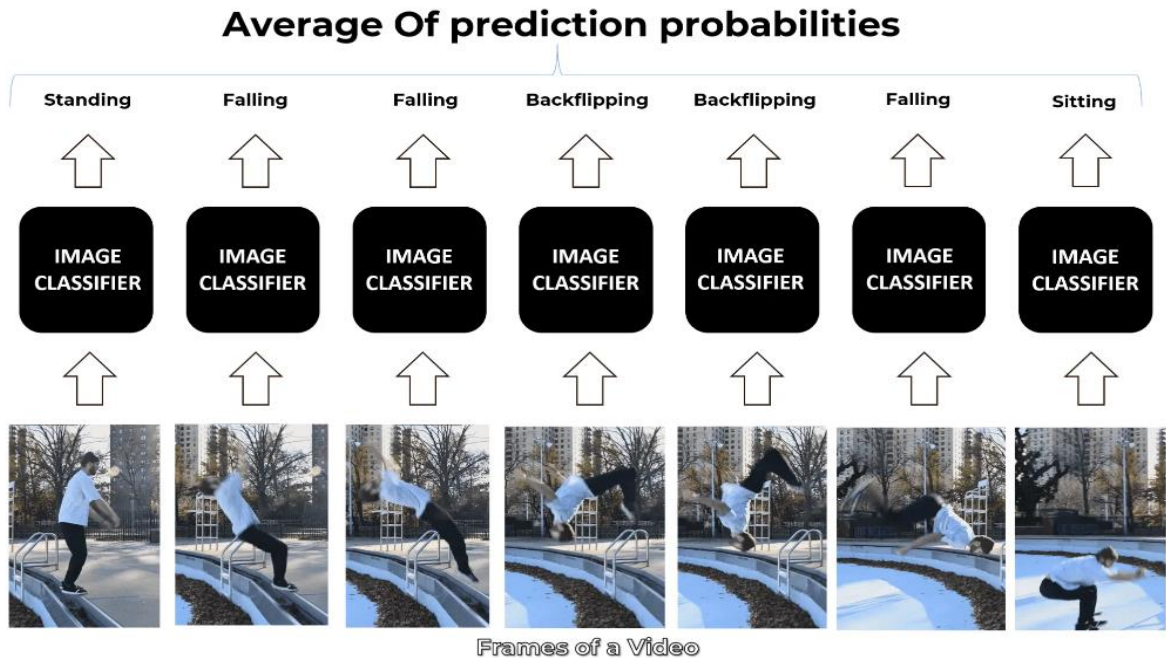
Approach 1: Single-Frame Classification

The simplest and most basic way of classifying actions in a video can be using an image classifier on each frame of the video and classify action in each frame independently. So, if we implement this approach for a video of a person doing a backflip, we will get the following results.



The classifier predicts Falling in some frames instead of Backflipping because this approach ignores the temporal relation of the frames sequence. And even if a person looks at those frames independently, he may think that the person is Falling. Now a simple

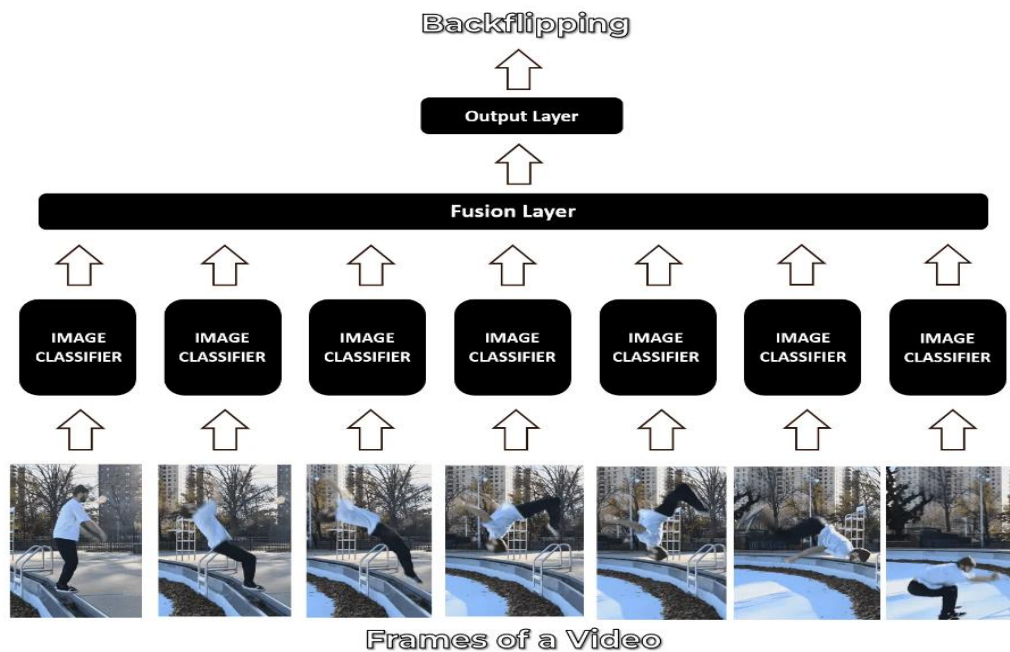
way to get a final prediction for the video is to consider the most frequent one which can work in simple scenarios but is **Falling** in our case and is not correct. So, another way to go about this is to take an average of the probabilities of predictions and get a more robust final prediction.



You should also check another **Video Classification and Human Activity Recognition** tutorial I had published a while back, in which I had discussed a number of other approaches too and implemented this one using a single-frame CNN with moving averages and it had worked fine for a relatively simpler problem. But as mentioned before, this approach is not effective, because it does not take into account the temporal aspect of the data.

Approach 2: Late Fusion

Another slightly different approach is late fusion, in which after performing predictions on each frame independently, the classification results are passed to a fusion layer that merges all the information and makes the prediction. This approach also leverages the temporal information of the data.

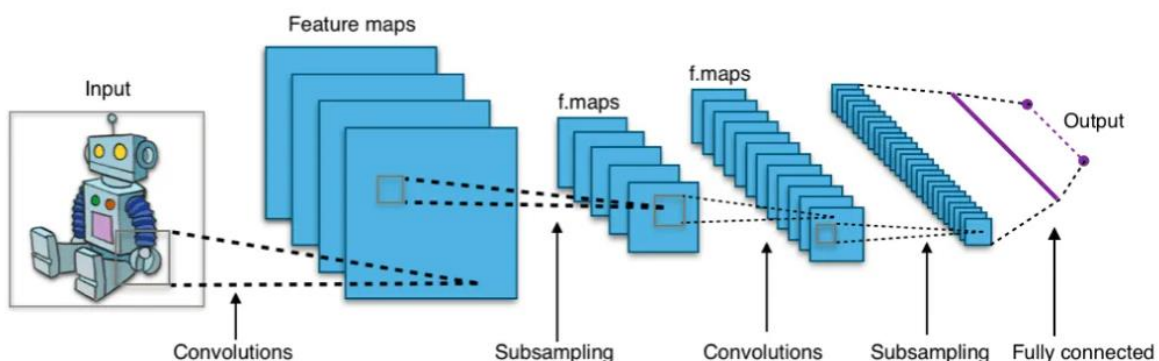


This approach does give decent results but is still not powerful enough. Now before moving to the next approach let's discuss what Convolutional Neural Networks are. So that you get an idea of what that black box named image classifier was, that I was using in the images.

Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN or ConvNet) is a type of deep neural network that is specifically designed to work with image data and excels when it comes to analyzing the images and making predictions on them.

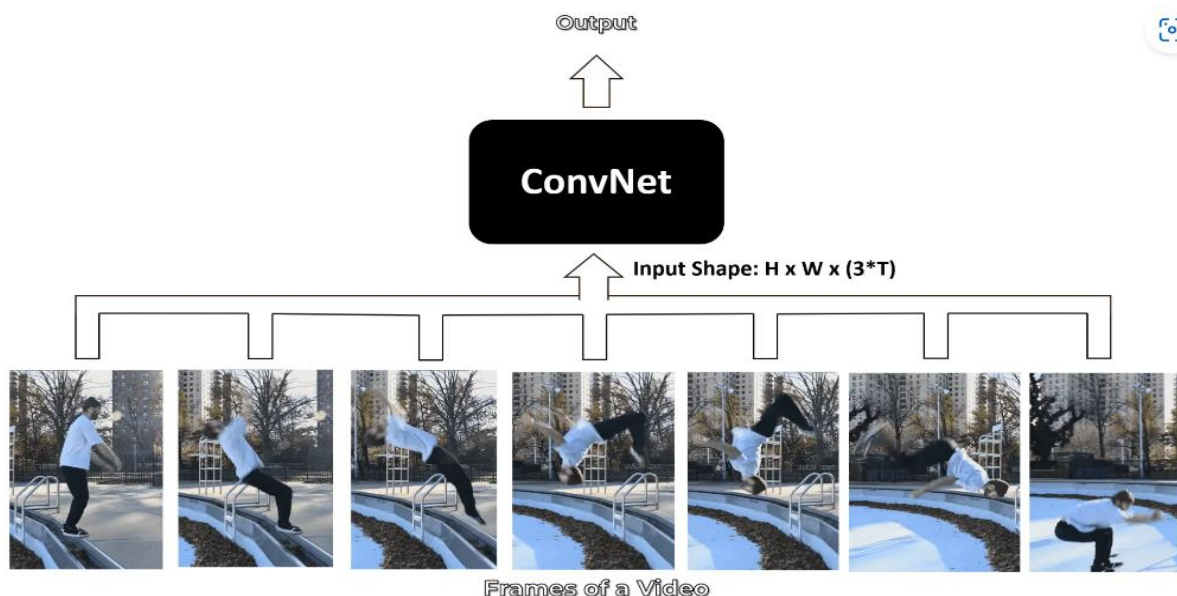
It works with kernels (called filters) that go over the image and generates feature maps (that represent whether a certain feature is present at a location in the image or not) and initially it generates few feature maps and as we go deeper in the network the number of feature maps is increased and the size of maps is decreased using pooling operations without losing critical information.



Each layer of a ConvNet learns features of increasing complexity which means, for example, the first layer may learn to detect edges and corners, while the last layer may learn to recognize humans in different postures.

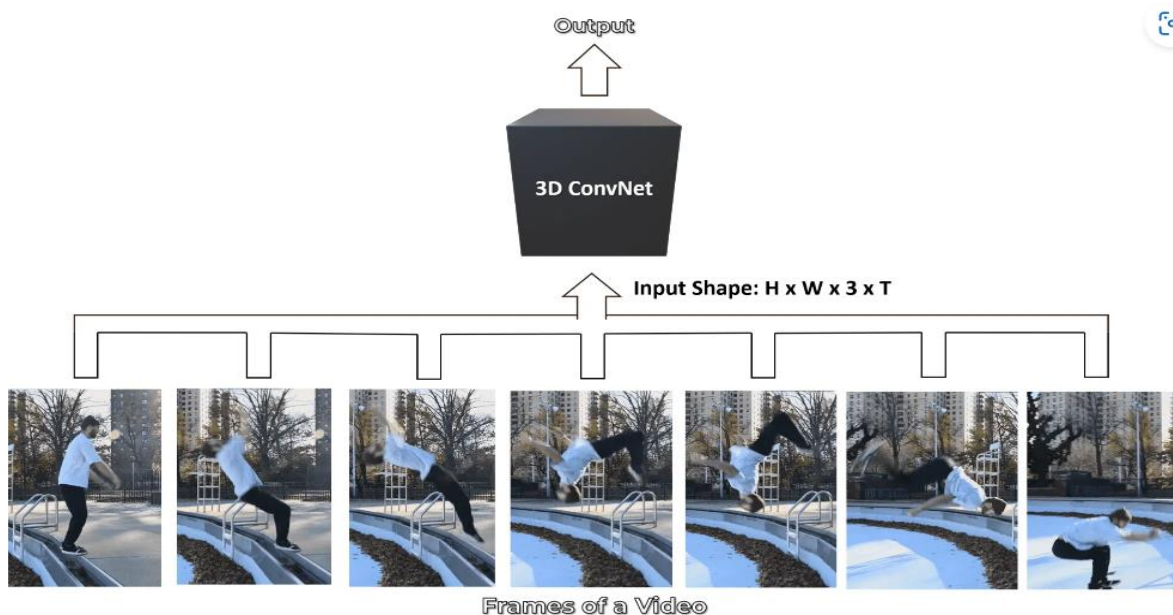
Now let's get back to discussing other approaches for video classification.

Approach 3: Early Fusion



Another approach of video classification is early fusion, in which all the information is merged at the beginning of the network, unlike late fusion which merges the information in the end. This is a powerful approach but still has its own limitations.

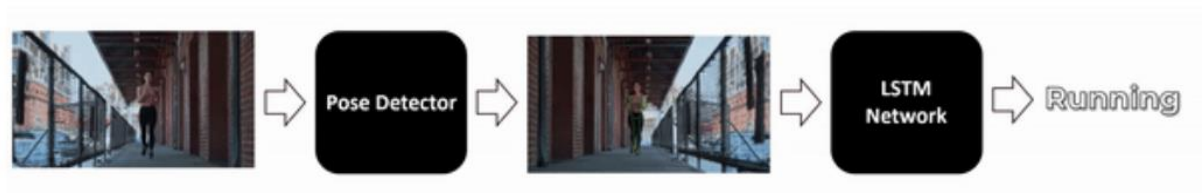
Approach 4: Using 3D CNN's (aka. Slow Fusion)



Another option is to use a 3D Convolutional Network, where the temporal and spatial information are merged slowly throughout the whole network that is why it's called

Slow Fusion. But a disadvantage of this approach is that it is computationally really expensive so it is pretty slow.

Approach 5: Using Pose Detection and LSTM



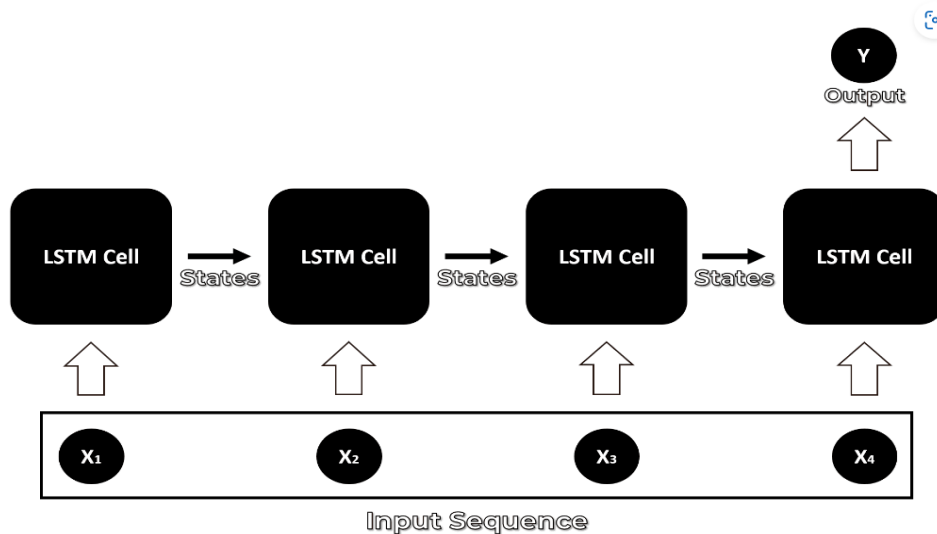
Another method is to use a pose detection network on the video to get the landmark coordinates of the person for each frame in the video. And then feed the landmarks to an LSTM Network to predict the activity of the person.

There are already a lot of efficient pose detectors out there that can be used for this approach. But a disadvantage of using this approach is that you discard all the information other than the landmarks, like the environment information can be very useful, for example for playing football action category the stadium and uniform info can help the model a lot in predicting the action accurately.

Before going to the approach that we will implement in this tutorial, let's briefly discuss what are Long Short-Term Memory (LSTM) networks, as we will be using them in the approach.

Long Short-Term Memory (LSTM)

An LSTM network is specifically designed to work with a data sequence as it takes into consideration all of the previous inputs while generating an output. LSTMs are actually a



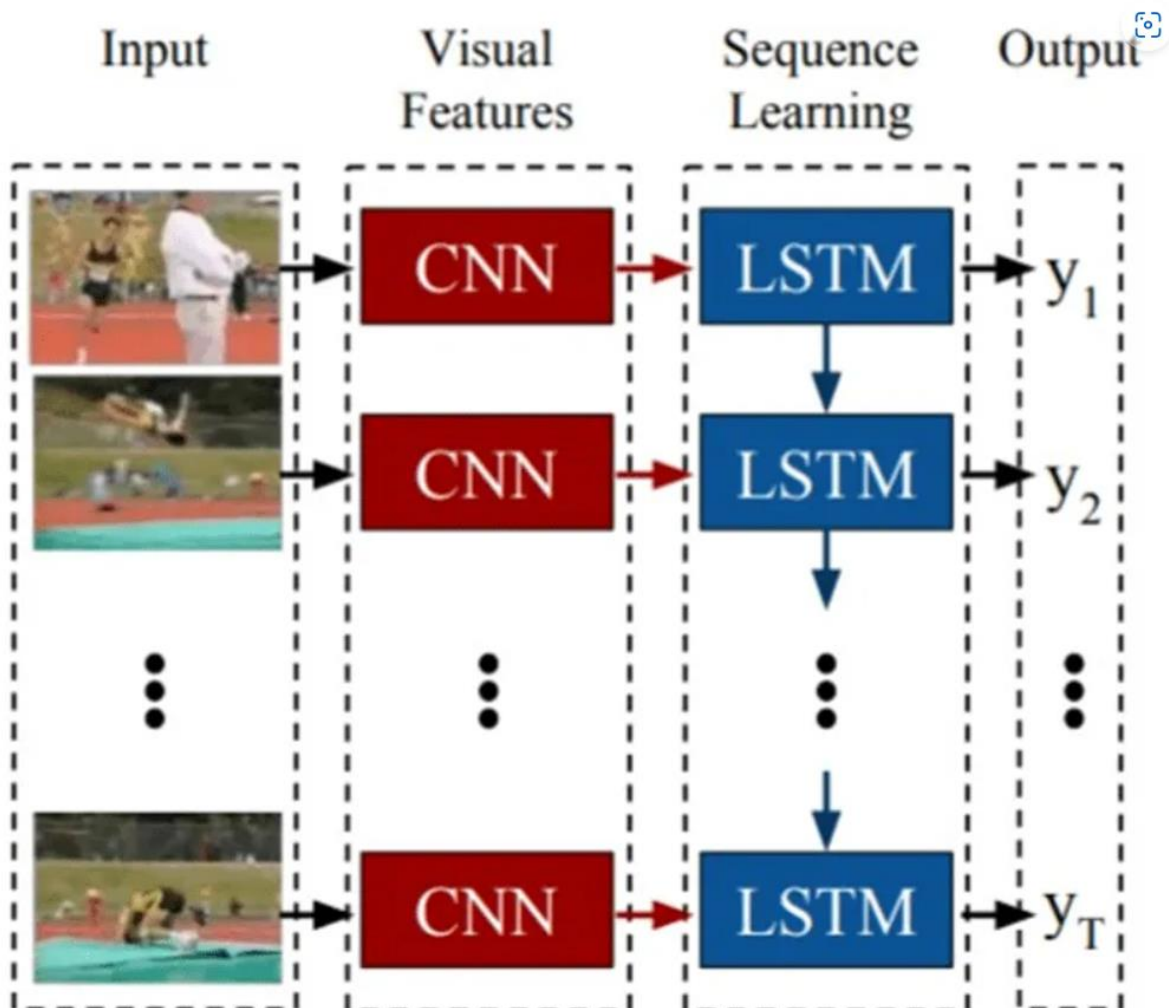
type of neural network called **Recurrent Neural Network**, but RNNs are not known to be effective for dealing with the Long-term dependencies in the input sequence because of a problem called the **Vanishing gradient problem**. LSTMs were developed to overcome the vanishing gradient and so an LSTM cell can remember context for long input sequences.

This makes an LSTM more capable of solving problems involving sequential data such as time series prediction, speech recognition, language translation, or music composition. But for now, we will only explore the role of LSTMs in developing better action recognition models.

Now let's move on towards the approach we will implement in this tutorial to build an Action Recognizer. We will use a Convolution Neural Network (CNN) + Long Short-Term Memory (LSTM) Network to perform Action Recognition while utilizing the Spatial-temporal aspect of the videos.

Approach 6: CNN + LSTM:

We will be using a CNN to extract spatial features at a given time step in the input sequence (video) and then an LSTM to identify temporal relations between frames.



The two architectures that we will be using to use CNN along with LSTM are:

1. **ConvLSTM**
2. **LRCN**

Both of these approaches can be used using TensorFlow. This tutorial also has a video version as well, that you can go and watch for a more detailed overview of the code.

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 YouTube videos

Alright, so without further ado, let's get started.

Import the Libraries

We will start by installing and importing the required libraries.

```
1 # Install the required libraries.  
2 pip install pafy youtube-dl moviepy  
Python
```

```
1 # Import the required libraries.  
2 import os  
3 import cv2  
4 import pafy  
5 import math  
6 import random  
7 import numpy as np  
8 import datetime as dt  
9 import tensorflow as tf  
10 from collections import deque  
11 import matplotlib.pyplot as plt  
12  
13 from moviepy.editor import *  
14 %matplotlib inline  
15  
16 from sklearn.model_selection import train_test_split  
17  
18 from tensorflow.keras.layers import *  
19 from tensorflow.keras.models import Sequential  
20 from tensorflow.keras.utils import to_categorical  
21 from tensorflow.keras.callbacks import EarlyStopping  
22 from tensorflow.keras.utils import plot_model
```

```
Python  
1 seed_constant = 27  
2 np.random.seed(seed_constant)
```

```
3 random.seed(seed_constant)
4 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**, 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

Let's download and extract the dataset.

```
1 # Discard the output of this cell.
2 %%capture
3
4 # Downlaod the UCF50 Dataset
5 wget --no-check-certificate https://www.crcv.ucf.edu/data/UCF50.rar
6
7 #Extract the Dataset
8 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.

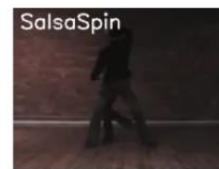
Python

```
1 # Create a Matplotlib figure and specify the size of the figure.
2 plt.figure(figsize = (20, 20))
3
4 # Get the names of all classes/categories in UCF50.
5 all_classes_names = os.listdir('UCF50')
6
7 # Generate a list of 20 random values. The values will be between 0-50,
8 # where 50 is the total number of class in the dataset.
9 random_range = random.sample(range(len(all_classes_names)), 20)
10
11 # Iterating through all the generated random values.
12 for counter, random_index in enumerate(random_range, 1):
13
14     # Retrieve a Class Name using the Random Index.
```

```

15 selected_class_Name = all_classes_names[random_index]
16
17 # Retrieve the list of all the video files present in the randomly selected Class Directory.
18 video_files_names_list = os.listdir(f'UCF50/{selected_class_Name}')
19
20 # Randomly select a video file from the list retrieved from the randomly selected Class Directory.
21 selected_video_file_name = random.choice(video_files_names_list)
22
23 # Initialize a VideoCapture object to read from the video File.
24 video_reader = cv2.VideoCapture(f'UCF50/{selected_class_Name}/{selected_video_file_name}')
25
26 # Read the first frame of the video file.
27 _, bgr_frame = video_reader.read()
28
29 # Release the VideoCapture object.
30 video_reader.release()
31
32 # Convert the frame from BGR into RGB format.
33 rgb_frame = cv2.cvtColor(bgr_frame, cv2.COLOR_BGR2RGB)
34
35 # Write the class name on the video frame.
36 cv2.putText(rgb_frame, selected_class_Name, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255),
37 2)
38
39 # 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.

```
1 # Specify the height and width to which each video frame will be resized in our dataset.
2 IMAGE_HEIGHT, IMAGE_WIDTH = 64, 64
3
4 # Specify the number of frames of a video that will be fed to the model as one sequence.
5 SEQUENCE_LENGTH = 20
6
7 # Specify the directory containing the UCF50 dataset.
8 DATASET_DIR = "UCF50"
9
10 # Specify the list containing the names of the classes used for training. Feel free to choose any set of classes.
11 CLASSES_LIST = ["WalkingWithDog", "TaiChi", "Swing", "HorseRace"]
```

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.

```
1 def frames_extraction(video_path):
2     """
3     This function will extract the required frames from a video after resizing and normalizing them.
4     Args:
5         video_path: The path of the video in the disk, whose frames are to be extracted.
6     Returns:
7         frames_list: A list containing the resized and normalized frames of the video.
8     """
9
10    # Declare a list to store video frames.
11    frames_list = []
12
13    # Read the Video File using the VideoCapture object.
14    video_reader = cv2.VideoCapture(video_path)
15
16    # Get the total number of frames in the video.
17    video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))
18
19    # Calculate the interval after which frames will be added to the list.
20    skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH), 1)
21
22    # Iterate through the Video Frames.
23    for frame_counter in range(SEQUENCE_LENGTH):
```



```

24
25 # Set the current frame position of the video.
26 video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window)
27
28 # Reading the frame from the video.
29 success, frame = video_reader.read()
30
31 # Check if Video frame is not successfully read then break the loop
32 if not success:
33     break
34
35 # Resize the Frame to fixed height and width.
36 resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))
37
38 # Normalize the resized frame by dividing it with 255 so that each pixel value then lies between 0 and 1
39 normalized_frame = resized_frame / 255
40
41 # Append the normalized frame into the frames list
42 frames_list.append(normalized_frame)
43
44 # Release the VideoCapture object.
45 video_reader.release()
46
47 # Return the frames list.
48 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**).

Python

```

1 def create_dataset():
2     """
3     This function will extract the data of the selected classes and create the required dataset.
4     Returns:
5         features:    A list containing the extracted frames of the videos.
6         labels:      A list containing the indexes of the classes associated with the videos.
7         video_files_paths: A list containing the paths of the videos in the disk.
8     """
9
10    # Declared Empty Lists to store the features, labels and video file path values.
11    features = []
12    labels = []
13    video_files_paths = []
14
15    # Iterating through all the classes mentioned in the classes list
16    for class_index, class_name in enumerate(CLASSES_LIST):
17
18        # Display the name of the class whose data is being extracted.
19        print(f'Extracting Data of Class: {class_name}')
20
21        # Get the list of video files present in the specific class name directory.
22        files_list = os.listdir(os.path.join(DATASET_DIR, class_name))
23

```

```

24 # Iterate through all the files present in the files list.
25 for file_name in files_list:
26
27     # Get the complete video path.
28     video_file_path = os.path.join(DATASET_DIR, class_name, file_name)
29
30     # Extract the frames of the video file.
31     frames = frames_extraction(video_file_path)
32
33     # Check if the extracted frames are equal to the SEQUENCE_LENGTH specified above.
34     # So, ignore the videos having frames less than the SEQUENCE_LENGTH.
35     if len(frames) == SEQUENCE_LENGTH:
36
37         # Append the data to their respective lists.
38         features.append(frames)
39         labels.append(class_index)
40         video_files_paths.append(video_file_path)
41
42 # Converting the list to numpy arrays
43 features = np.asarray(features)
44 labels = np.array(labels)
45
46 # Return the frames, class index, and video file path.
47 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.

```

1 # Create the dataset.
2 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

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

Python

```

1 # Using Keras's to_categorical method to convert labels into one-hot-encoded vectors
2 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.

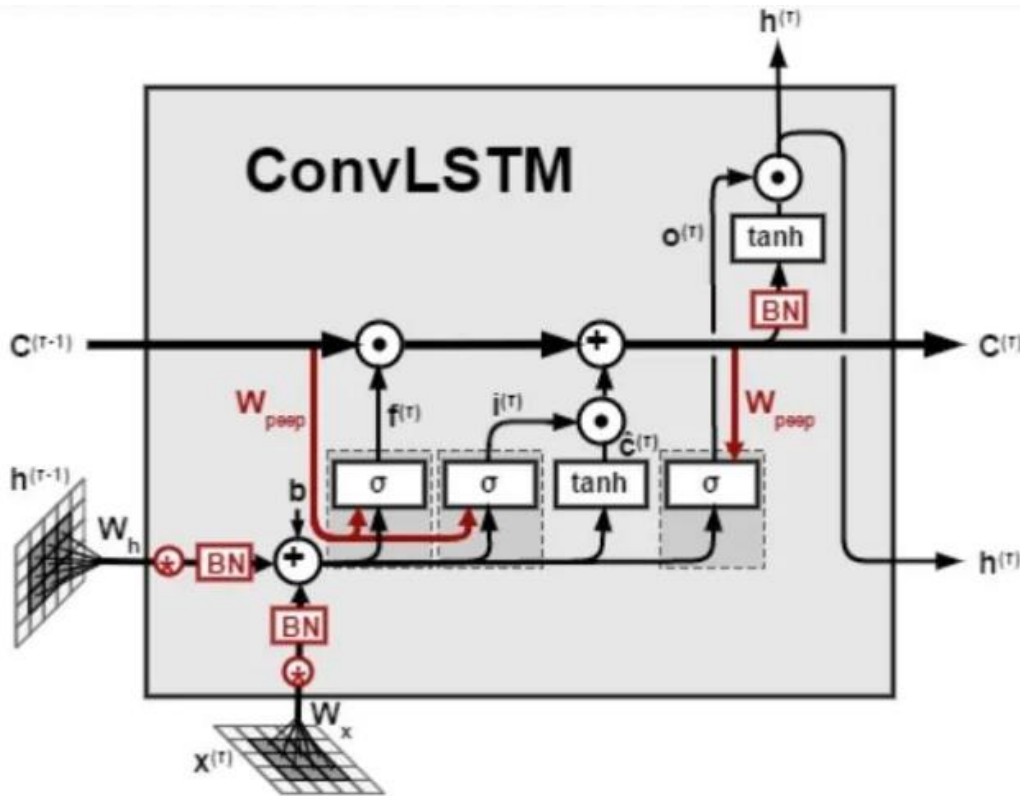
```

1 # Split the Data into Train ( 75% ) and Test Set ( 25% ).
2 features_train, features_test, labels_train, labels_test = train_test_split(features, one_hot_encoded_labels,
3                                     test_size = 0.25, shuffle = True, random_state = seed_constant)

```

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.



For video classification, this approach effectively captures the spatial relation in the individual frames and the temporal relation across the different frames. As a result of this convolution structure, the ConvLSTM is capable of taking in 3-dimensional input (width, height, num_of_channels) whereas a simple LSTM only takes in 1-dimensional input hence an LSTM is incompatible for modeling Spatio-temporal data on its own.

You can read the paper **Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting** by Xingjian Shi (NIPS 2015), to learn more about this architecture.

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.

```

1 def create_convlstm_model():
2     """
3     This function will construct the required convlstm model.
4     Returns:
5         model: It is the required constructed convlstm model.
6     """
7
8     # We will use a Sequential model for model construction
9     model = Sequential()
10
11     # Define the Model Architecture.
12     #####
13
14     model.add(ConvLSTM2D(filters = 4, kernel_size = (3, 3), activation = 'tanh', data_format = "channels_last",
15         recurrent_dropout=0.2, return_sequences=True, input_shape = (SEQUENCE_LENGTH,
16             IMAGE_HEIGHT, IMAGE_WIDTH, 3)))
17
18     model.add(MaxPooling3D(pool_size=(1, 2, 2), padding='same', data_format='channels_last'))
19     model.add(TimeDistributed(Dropout(0.2)))
20
21     model.add(ConvLSTM2D(filters = 8, kernel_size = (3, 3), activation = 'tanh', data_format = "channels_last",
22         recurrent_dropout=0.2, return_sequences=True))
23
24     model.add(MaxPooling3D(pool_size=(1, 2, 2), padding='same', data_format='channels_last'))
25     model.add(TimeDistributed(Dropout(0.2)))
26
27     model.add(ConvLSTM2D(filters = 14, kernel_size = (3, 3), activation = 'tanh', data_format = "channels_last",
28         recurrent_dropout=0.2, return_sequences=True))
29
30     model.add(MaxPooling3D(pool_size=(1, 2, 2), padding='same', data_format='channels_last'))
31     model.add(TimeDistributed(Dropout(0.2)))
32
33     model.add(ConvLSTM2D(filters = 16, kernel_size = (3, 3), activation = 'tanh', data_format = "channels_last",
34         recurrent_dropout=0.2, return_sequences=True))
35
36     model.add(MaxPooling3D(pool_size=(1, 2, 2), padding='same', data_format='channels_last'))
37     #model.add(TimeDistributed(Dropout(0.2)))
38
39     model.add(Flatten())
40
41     model.add(Dense(len(CLASSES_LIST), activation = "softmax"))
42
43     #####
44
45     # Display the models summary.
46     model.summary()
47
48     # Return the constructed convlstm model.
49     return model

```

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

```

1 # Construct the required convlstm model.
2 convlstm_model = create_convlstm_model()
3
4 # Display the success message.
5 print("Model Created Successfully!")

```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv_lstm2d_8 (ConvLSTM2D)	(None, 20, 62, 62, 4)	1024
max_pooling3d_8 (MaxPooling3)	(None, 20, 31, 31, 4)	0
time_distributed_6 (TimeDist)	(None, 20, 31, 31, 4)	0
conv_lstm2d_9 (ConvLSTM2D)	(None, 20, 29, 29, 8)	3488
max_pooling3d_9 (MaxPooling3)	(None, 20, 15, 15, 8)	0
time_distributed_7 (TimeDist)	(None, 20, 15, 15, 8)	0
conv_lstm2d_10 (ConvLSTM2D)	(None, 20, 13, 13, 14)	11144
max_pooling3d_10 (MaxPooling)	(None, 20, 7, 7, 14)	0
time_distributed_8 (TimeDist)	(None, 20, 7, 7, 14)	0
conv_lstm2d_11 (ConvLSTM2D)	(None, 20, 5, 5, 16)	17344
max_pooling3d_11 (MaxPooling)	(None, 20, 3, 3, 16)	0
flatten_2 (Flatten)	(None, 2880)	0
dense_2 (Dense)	(None, 4)	11524
Total params: 44,524		
Trainable params: 44,524		
Non-trainable params: 0		
Model Created Successfully!		

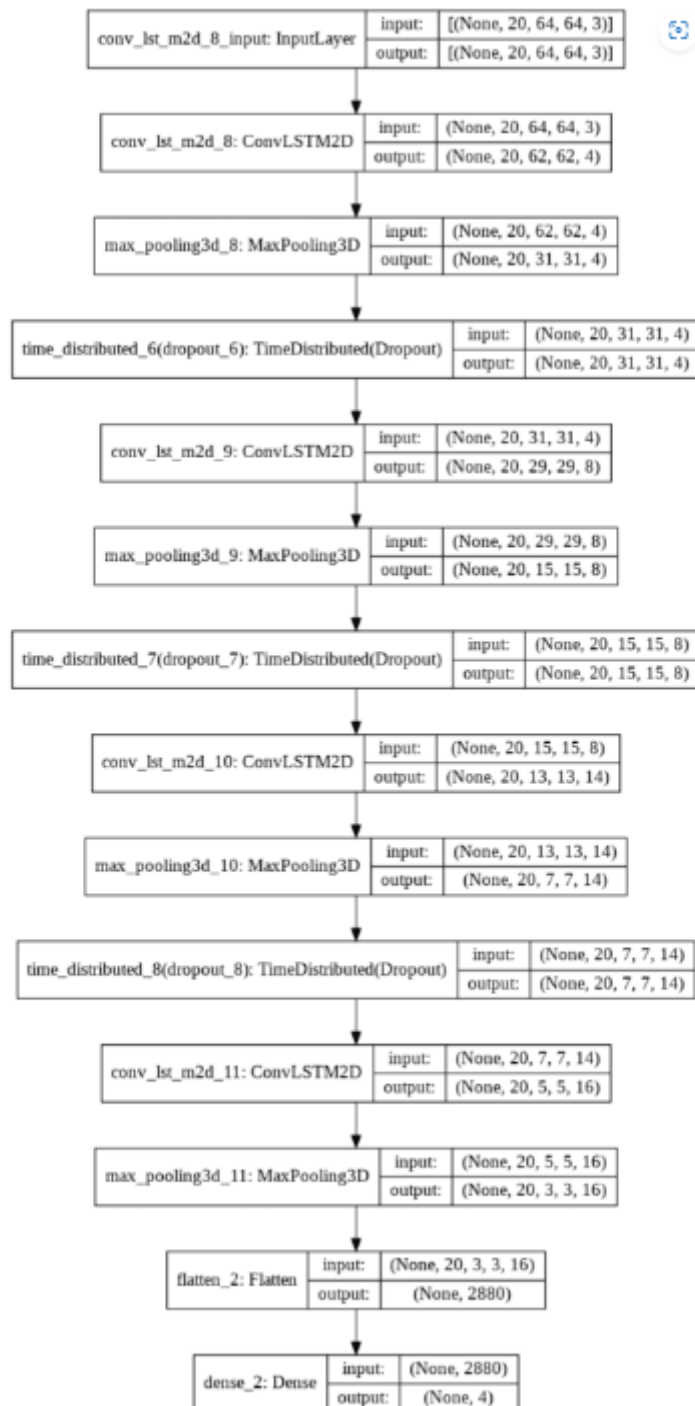
Check Model's Structure:

Now we will use the **plot_model()** function, to check the structure of the constructed model, this is helpful while constructing a complex network and making that the network is created correctly.

```

1 # Plot the structure of the constructed model.
2 plot_model(convlstm_model, to_file = 'convlstm_model_structure_plot.png', show_shapes = True,
3 show_layer_names = True)

```

Step 4.2: Compile & Train the Model

Next, we will add an early stopping callback to prevent **overfitting** and start the training after compiling the model.

```

1 # Create an Instance of Early Stopping Callback
2 early_stopping_callback = EarlyStopping(monitor = 'val_loss', patience = 10, mode = 'min', restore_best_weights =
3 True)
4

```

```

5 # Compile the model and specify loss function, optimizer and metrics values to the model
6 convlstm_model.compile(loss = 'categorical_crossentropy', optimizer = 'Adam', metrics = ["accuracy"])
7
8 # Start training the model.
convlstm_model_training_history = convlstm_model.fit(x = features_train, y = labels_train, epochs = 50, batch_size
= 4, shuffle = True, validation_split = 0.2, callbacks = [early_stopping_callback])

```

```

Epoch 1/50
73/73 [=====] - 164s 2s/step - loss: 1.3917 - accuracy: 0.2774 - val_loss: 1.3715 - val_accuracy: 0.3562
Epoch 2/50
73/73 [=====] - 153s 2s/step - loss: 1.3410 - accuracy: 0.3801 - val_loss: 1.2734 - val_accuracy: 0.5205
Epoch 3/50
73/73 [=====] - 153s 2s/step - loss: 1.1869 - accuracy: 0.5068 - val_loss: 1.2266 - val_accuracy: 0.4110
Epoch 4/50
73/73 [=====] - 154s 2s/step - loss: 1.0063 - accuracy: 0.5685 - val_loss: 1.0510 - val_accuracy: 0.4932
Epoch 5/50
73/73 [=====] - 154s 2s/step - loss: 0.8963 - accuracy: 0.6096 - val_loss: 0.7548 - val_accuracy: 0.6301
Epoch 6/50
73/73 [=====] - 153s 2s/step - loss: 0.7132 - accuracy: 0.6712 - val_loss: 0.7022 - val_accuracy: 0.7123
Epoch 7/50
73/73 [=====] - 152s 2s/step - loss: 0.5275 - accuracy: 0.7979 - val_loss: 0.4703 - val_accuracy: 0.8356
Epoch 8/50
73/73 [=====] - 152s 2s/step - loss: 0.4196 - accuracy: 0.8493 - val_loss: 0.5588 - val_accuracy: 0.7808
Epoch 9/50
73/73 [=====] - 152s 2s/step - loss: 0.3807 - accuracy: 0.8459 - val_loss: 0.5261 - val_accuracy: 0.7808
Epoch 10/50
73/73 [=====] - 154s 2s/step - loss: 0.3322 - accuracy: 0.8733 - val_loss: 0.4099 - val_accuracy: 0.8219
Epoch 11/50
73/73 [=====] - 153s 2s/step - loss: 0.2160 - accuracy: 0.9349 - val_loss: 0.5320 - val_accuracy: 0.8219
Epoch 12/50
73/73 [=====] - 153s 2s/step - loss: 0.1624 - accuracy: 0.9521 - val_loss: 0.3715 - val_accuracy: 0.8493
Epoch 13/50
73/73 [=====] - 155s 2s/step - loss: 0.1219 - accuracy: 0.9589 - val_loss: 0.5322 - val_accuracy: 0.8356
Epoch 14/50
73/73 [=====] - 154s 2s/step - loss: 0.1150 - accuracy: 0.9555 - val_loss: 0.5310 - val_accuracy: 0.7534
Epoch 15/50
73/73 [=====] - 153s 2s/step - loss: 0.0767 - accuracy: 0.9692 - val_loss: 0.3330 - val_accuracy: 0.9041
Epoch 16/50
73/73 [=====] - 152s 2s/step - loss: 0.0393 - accuracy: 0.9897 - val_loss: 0.4067 - val_accuracy: 0.8630
Epoch 17/50
73/73 [=====] - 151s 2s/step - loss: 0.0268 - accuracy: 0.9966 - val_loss: 0.2886 - val_accuracy: 0.8767
Epoch 18/50
73/73 [=====] - 153s 2s/step - loss: 0.0219 - accuracy: 0.9966 - val_loss: 0.4479 - val_accuracy: 0.9041
Epoch 19/50
73/73 [=====] - 154s 2s/step - loss: 0.0049 - accuracy: 1.0000 - val_loss: 0.4653 - val_accuracy: 0.8904
Epoch 20/50
73/73 [=====] - 154s 2s/step - loss: 0.0033 - accuracy: 1.0000 - val_loss: 0.4832 - val_accuracy: 0.8904
Epoch 21/50
73/73 [=====] - 154s 2s/step - loss: 0.0034 - accuracy: 1.0000 - val_loss: 0.4561 - val_accuracy: 0.8904
Epoch 22/50
73/73 [=====] - 153s 2s/step - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.4832 - val_accuracy: 0.8904
Epoch 23/50
73/73 [=====] - 152s 2s/step - loss: 0.0015 - accuracy: 1.0000 - val_loss: 0.4463 - val_accuracy: 0.8904
Epoch 24/50
73/73 [=====] - 153s 2s/step - loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.4655 - val_accuracy: 0.8904
Epoch 25/50
73/73 [=====] - 154s 2s/step - loss: 9.4172e-04 - accuracy: 1.0000 - val_loss: 0.4625 - val_accuracy: 0.9041
Epoch 26/50
73/73 [=====] - 153s 2s/step - loss: 8.4565e-04 - accuracy: 1.0000 - val_loss: 0.4504 - val_accuracy: 0.9178
Epoch 27/50
73/73 [=====] - 154s 2s/step - loss: 7.3753e-04 - accuracy: 1.0000 - val_loss: 0.4469 - val_accuracy: 0.9178

```

Evaluate the Trained Model

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

```

1 # Evaluate the trained model.
2 model_evaluation_history = convlstm_model.evaluate(features_test, labels_test)
4/4 [=====] - 14s 3s/step - loss: 0.8976 - accuracy:
0.8033

```

Save the Model

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

```

1 # Get the loss and accuracy from model_evaluation_history.
2 model_evaluation_loss, model_evaluation_accuracy = model_evaluation_history
3
4 # Define the string date format.
5 # Get the current Date and Time in a DateTime Object.
6 # Convert the DateTime object to string according to the style mentioned in date_time_format string.
7 date_time_format = '%Y_%m_%d_%H_%M_%S'
8 current_date_time_dt = dt.datetime.now()
9 current_date_time_string = dt.datetime.strftime(current_date_time_dt, date_time_format)
10
11 # Define a useful name for our model to make it easy for us while navigating through multiple saved models.
12 model_file_name =
13 f'convlstm_model__Date_Time_{current_date_time_string}__Loss_{model_evaluation_loss}__Accuracy_{model_
14 evaluation_accuracy}.h5'
15
16 # Save your Model.
17 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.

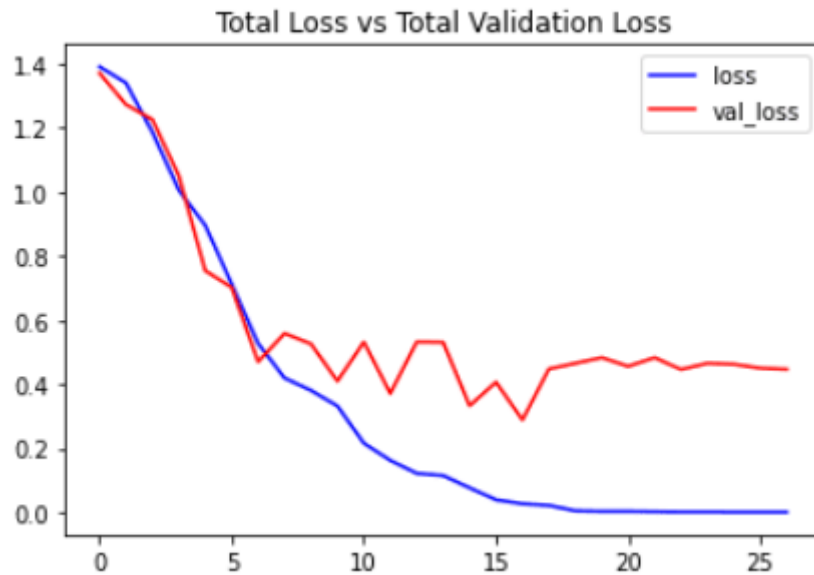
```

1 def plot_metric(model_training_history, metric_name_1, metric_name_2, plot_name):
2     """
3     This function will plot the metrics passed to it in a graph.
4     Args:
5         model_training_history: A history object containing a record of training and validation
6                                 loss values and metrics values at successive epochs
7         metric_name_1:         The name of the first metric that needs to be plotted in the graph.
8         metric_name_2:         The name of the second metric that needs to be plotted in the graph.
9         plot_name:             The title of the graph.
10    """
11
12    # Get metric values using metric names as identifiers.
13    metric_value_1 = model_training_history.history[metric_name_1]
14    metric_value_2 = model_training_history.history[metric_name_2]
15
16    # Construct a range object which will be used as x-axis (horizontal plane) of the graph.
17    epochs = range(len(metric_value_1))
18
19    # Plot the Graph.
20    plt.plot(epochs, metric_value_1, 'blue', label = metric_name_1)
21    plt.plot(epochs, metric_value_2, 'red', label = metric_name_2)
22
23    # Add title to the plot.
24    plt.title(str(plot_name))
25
26    # Add legend to the plot.
27    plt.legend()

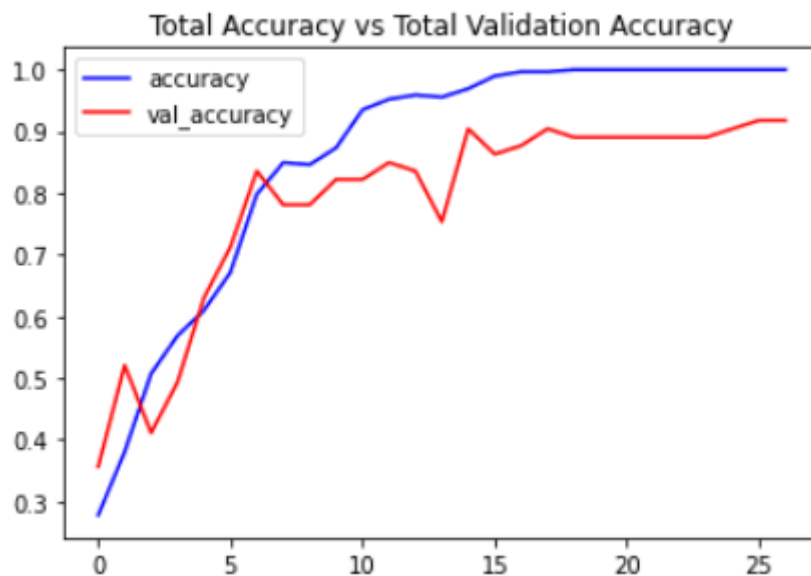
```

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

```
1 # Visualize the training and validation loss metrics.  
2 plot_metric(convlstm_model_training_history, 'loss', 'val_loss', 'Total Loss vs Total Validation Loss')
```



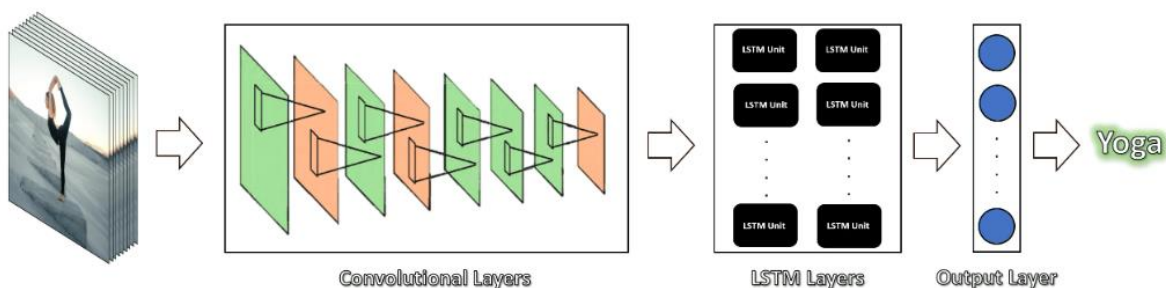
```
1 # Visualize the training and validation accuracy metrics.  
2 plot_metric(convlstm_model_training_history, 'accuracy', 'val_accuracy', 'Total Accuracy vs Total Validation Accuracy')
```



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.



You can read the paper [Long-term Recurrent Convolutional Networks for Visual Recognition and Description](#) by Jeff Donahue (CVPR 2015), to learn more about this architecture.

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.

```
1 def create_LRCN_model():
2     """
3     This function will construct the required LRCN model.
4     Returns:
5         model: It is the required constructed LRCN model.
6     """
7
8     # We will use a Sequential model for model construction.
9     model = Sequential()
10
11     # Define the Model Architecture.
12     #####
13
14     model.add(TimeDistributed(Conv2D(16, (3, 3), padding='same', activation = 'relu'),
15                               input_shape = (SEQUENCE_LENGTH, IMAGE_HEIGHT, IMAGE_WIDTH, 3)))
16
17     model.add(TimeDistributed(MaxPooling2D((4, 4))))
18     model.add(TimeDistributed(Dropout(0.25)))
19
20     model.add(TimeDistributed(Conv2D(32, (3, 3), padding='same', activation = 'relu')))
21     model.add(TimeDistributed(MaxPooling2D((4, 4))))
22     model.add(TimeDistributed(Dropout(0.25)))
23
24     model.add(TimeDistributed(Conv2D(64, (3, 3), padding='same', activation = 'relu')))
25     model.add(TimeDistributed(MaxPooling2D((2, 2))))
26     model.add(TimeDistributed(Dropout(0.25)))
27
28     model.add(TimeDistributed(Conv2D(64, (3, 3), padding='same', activation = 'relu')))
29     model.add(TimeDistributed(MaxPooling2D((2, 2))))
30     #model.add(TimeDistributed(Dropout(0.25)))
31
32     model.add(TimeDistributed(Flatten()))
33
34     model.add(LSTM(32))
35
36     model.add(Dense(len(CLASSES_LIST), activation = 'softmax'))
37
38     #####
39
40     # Display the models summary.
41     model.summary()
42
43     # Return the constructed LRCN model.
44     return model
```

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

```
Python
1 # Construct the required LRCN model.
2 LRCN_model = create_LRCN_model()
3
4 # Display the success message.
5 print("Model Created Successfully!")
```

Model: "sequential_4"



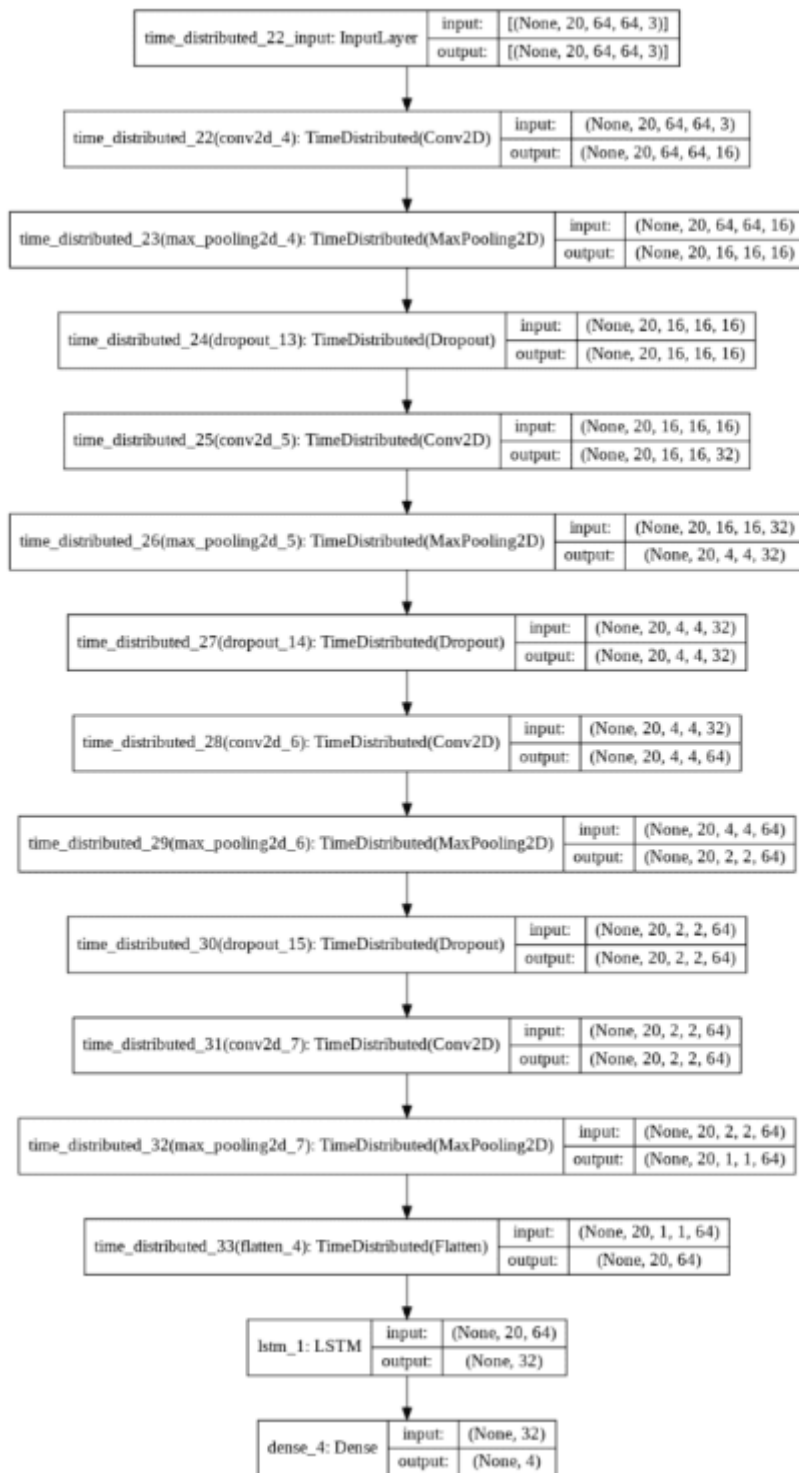
Layer (type)	Output Shape	Param #
=====		
time_distributed_22 (TimeDis	(None, 20, 64, 64, 16)	448
time_distributed_23 (TimeDis	(None, 20, 16, 16, 16)	0
time_distributed_24 (TimeDis	(None, 20, 16, 16, 16)	0
time_distributed_25 (TimeDis	(None, 20, 16, 16, 32)	4640
time_distributed_26 (TimeDis	(None, 20, 4, 4, 32)	0
time_distributed_27 (TimeDis	(None, 20, 4, 4, 32)	0
time_distributed_28 (TimeDis	(None, 20, 4, 4, 64)	18496
time_distributed_29 (TimeDis	(None, 20, 2, 2, 64)	0
time_distributed_30 (TimeDis	(None, 20, 2, 2, 64)	0
time_distributed_31 (TimeDis	(None, 20, 2, 2, 64)	36928
time_distributed_32 (TimeDis	(None, 20, 1, 1, 64)	0
time_distributed_33 (TimeDis	(None, 20, 64)	0
lstm_1 (LSTM)	(None, 32)	12416
dense_4 (Dense)	(None, 4)	132
=====		
Total params: 73,060		
Trainable params: 73,060		
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.

Python

```
1 # Plot the structure of the constructed LRCN model.
2 plot_model(LRCN_model, to_file = 'LRCN_model_structure_plot.png', show_shapes = True, show_layer_names
  = True)
```



Step 5.2: Compile & Train the Model

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

```

1 # Create an Instance of Early Stopping Callback.
2 early_stopping_callback = EarlyStopping(monitor = 'val_loss', patience = 15, mode = 'min', restore_best_weights =
3 True)
4

```

```

5# Compile the model and specify loss function, optimizer and metrics to the model.
6LRCN_model.compile(loss = 'categorical_crossentropy', optimizer = 'Adam', metrics = ["accuracy"])
7
8# Start training the model.
LRCN_model_training_history = LRCN_model.fit(x = features_train, y = labels_train, epochs = 70, batch_size = 4 ,
shuffle = True, validation_split = 0.2, callbacks = [early_stopping_callback])

```

```

73/73 [=====] - 13s 384ms/step - loss: 0.1410 - accuracy: 0.9043 - val_loss: 0.4729 - val_accuracy: 0.9413
Epoch 21/70
73/73 [=====] - 13s 385ms/step - loss: 0.2658 - accuracy: 0.8867 - val_loss: 0.2687 - val_accuracy: 0.8767
Epoch 22/70
73/73 [=====] - 14s 386ms/step - loss: 0.2283 - accuracy: 0.9281 - val_loss: 0.2326 - val_accuracy: 0.9041
Epoch 23/70
73/73 [=====] - 14s 385ms/step - loss: 0.1679 - accuracy: 0.9452 - val_loss: 0.1873 - val_accuracy: 0.9315
Epoch 24/70
73/73 [=====] - 13s 384ms/step - loss: 0.0761 - accuracy: 0.9768 - val_loss: 0.3040 - val_accuracy: 0.8403
Epoch 25/70
73/73 [=====] - 13s 384ms/step - loss: 0.0989 - accuracy: 0.9678 - val_loss: 0.1045 - val_accuracy: 0.9589
Epoch 26/70
73/73 [=====] - 13s 384ms/step - loss: 0.1117 - accuracy: 0.9728 - val_loss: 0.1237 - val_accuracy: 0.9452
Epoch 27/70
73/73 [=====] - 13s 384ms/step - loss: 0.0380 - accuracy: 0.9863 - val_loss: 0.0028 - val_accuracy: 0.9589
Epoch 28/70
73/73 [=====] - 13s 384ms/step - loss: 0.0353 - accuracy: 0.9863 - val_loss: 0.0924 - val_accuracy: 0.9728
Epoch 29/70
73/73 [=====] - 13s 388ms/step - loss: 0.0738 - accuracy: 0.9768 - val_loss: 0.0661 - val_accuracy: 0.9728
Epoch 30/70
73/73 [=====] - 13s 389ms/step - loss: 0.1096 - accuracy: 0.9452 - val_loss: 0.1879 - val_accuracy: 0.9315
Epoch 31/70
73/73 [=====] - 14s 389ms/step - loss: 0.0816 - accuracy: 0.9658 - val_loss: 0.4088 - val_accuracy: 0.8004
Epoch 32/70
73/73 [=====] - 14s 393ms/step - loss: 0.1815 - accuracy: 0.9726 - val_loss: 0.3573 - val_accuracy: 0.9452
Epoch 33/70
73/73 [=====] - 15s 200ms/step - loss: 0.0198 - accuracy: 0.9980 - val_loss: 0.1183 - val_accuracy: 0.9726
Epoch 34/70
73/73 [=====] - 13s 382ms/step - loss: 0.0887 - accuracy: 0.9768 - val_loss: 0.2060 - val_accuracy: 0.9452
Epoch 35/70
73/73 [=====] - 13s 383ms/step - loss: 0.0355 - accuracy: 0.9932 - val_loss: 0.1048 - val_accuracy: 0.9588
Epoch 36/70
73/73 [=====] - 13s 383ms/step - loss: 0.0222 - accuracy: 0.9966 - val_loss: 0.1988 - val_accuracy: 0.9178
Epoch 37/70
73/73 [=====] - 13s 383ms/step - loss: 0.0105 - accuracy: 1.0000 - val_loss: 0.1827 - val_accuracy: 0.9588
Epoch 38/70
73/73 [=====] - 13s 380ms/step - loss: 0.0642 - accuracy: 0.9780 - val_loss: 0.1950 - val_accuracy: 0.9178
Epoch 39/70
73/73 [=====] - 13s 384ms/step - loss: 0.1323 - accuracy: 0.9623 - val_loss: 0.1693 - val_accuracy: 0.9178
Epoch 40/70
73/73 [=====] - 13s 383ms/step - loss: 0.0305 - accuracy: 0.9966 - val_loss: 0.1708 - val_accuracy: 0.9315
Epoch 41/70
73/73 [=====] - 13s 384ms/step - loss: 0.0348 - accuracy: 0.9932 - val_loss: 0.3548 - val_accuracy: 0.9178
Epoch 42/70
73/73 [=====] - 13s 384ms/step - loss: 0.0110 - accuracy: 1.0000 - val_loss: 0.2215 - val_accuracy: 0.9315
Epoch 43/70
73/73 [=====] - 13s 384ms/step - loss: 0.0067 - accuracy: 1.0000 - val_loss: 0.2456 - val_accuracy: 0.9452
Epoch 44/70
73/73 [=====] - 13s 380ms/step - loss: 0.0079 - accuracy: 1.0000 - val_loss: 0.3884 - val_accuracy: 0.9178

```

Evaluating the trained Model

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

```

1# Evaluate the trained model.
2model_evaluation_history = LRCN_model.evaluate(features_test, labels_test)
4/4 [=====] - 2s 418ms/step - loss: 0.2242 -
accuracy: 0.9262

```

Save the Model

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

Python

```

1# Get the loss and accuracy from model_evaluation_history.
2model_evaluation_loss, model_evaluation_accuracy = model_evaluation_history
3
4# Define the string date format.
5# Get the current Date and Time in a DateTime Object.
6# Convert the DateTime object to string according to the style mentioned in date_time_format string.
7date_time_format = '%Y_%m_%d_%H_%M_%S'
8current_date_time_dt = dt.datetime.now()
9current_date_time_string = dt.datetime.strftime(current_date_time_dt, date_time_format)
10

```

```

11# Define a useful name for our model to make it easy for us while navigating through multiple saved models.
12model_file_name =
13f'LRCN_model__Date_Time_{current_date_time_string}__Loss_{model_evaluation_loss}__Accuracy_{model_evaluation_accuracy}'
14
15# Save the Model.
16LRCN_model.save(model_file_name)

```

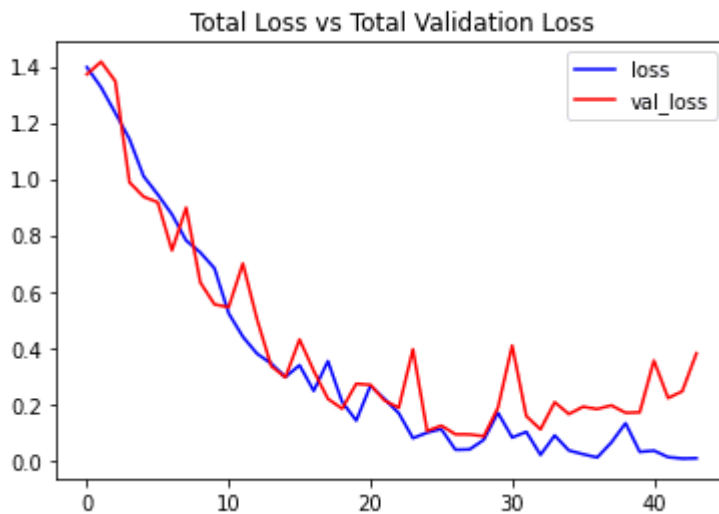
Step 5.3: Plot Model's Loss & Accuracy Curves

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

```

1# Visualize the training and validation loss metrics.
2plot_metric(LRCN_model_training_history, 'loss', 'val_loss', 'Total Loss vs Total Validation Loss')

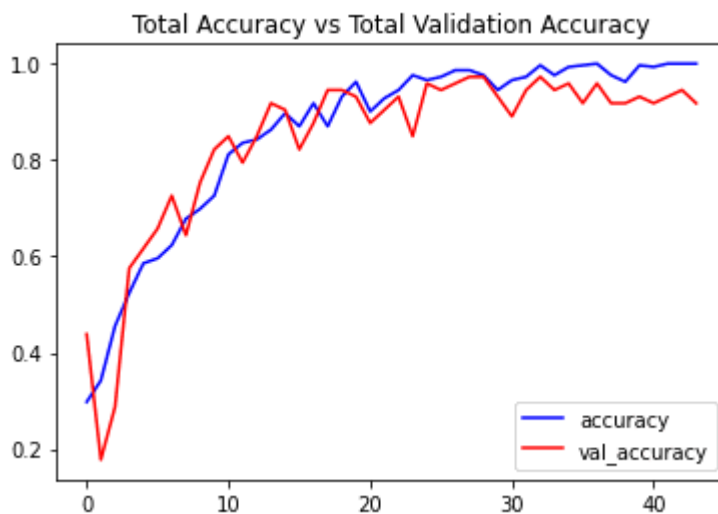
```



```

1# Visualize the training and validation accuracy metrics.
2plot_metric(LRCN_model_training_history, 'accuracy', 'val_accuracy', 'Total Accuracy vs Total Validation Accuracy')

```



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.

```
1 def download_youtube_videos(youtube_video_url, output_directory):
2     """
3     This function downloads the youtube video whose URL is passed to it as an argument.
4     Args:
5         youtube_video_url: URL of the video that is required to be downloaded.
6         output_directory: The directory path to which the video needs to be stored after downloading.
7     Returns:
8         title: The title of the downloaded youtube video.
9     """
10
11     # Create a video object which contains useful information about the video.
12     video = pafy.new(youtube_video_url)
13
14     # Retrieve the title of the video.
15     title = video.title
16
17     # Get the best available quality object for the video.
18     video_best = video.getbest()
19
20     # Construct the output file path.
21     output_file_path = f'{output_directory}/{title}.mp4'
22
23     # Download the youtube video at the best available quality and store it to the constructed path.
24     video_best.download(filepath = output_file_path, quiet = True)
25
26     # Return the video title.
27     return title
```

Download a Test Video:

Now we will utilize the function **download_youtube_videos()** created above to download a youtube video on which the LRCN model will be tested.

```
1 # Make the Output directory if it does not exist
2 test_videos_directory = 'test_videos'
3 os.makedirs(test_videos_directory, exist_ok = True)
4
5 # Download a YouTube Video.
6 video_title = download_youtube_videos('https://www.youtube.com/watch?v=8u0qjmHIOcE',
7 test_videos_directory)
8
9 # Get the YouTube Video's path we just downloaded.
10 input_video_file_path = f'{test_videos_directory}/{video_title}.mp4'
```

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.

```
1 def predict_on_video(video_file_path, output_file_path, SEQUENCE_LENGTH):
2     """
3     This function will perform action recognition on a video using the LRCN model.
4     Args:
5     video_file_path: The path of the video stored in the disk on which the action recognition is to be
6     performed.
7     output_file_path: The path where the output video with the predicted action being performed overlayed
8     will be stored.
9     SEQUENCE_LENGTH: The fixed number of frames of a video that can be passed to the model as one
10    sequence.
11    """
12
13    # Initialize the VideoCapture object to read from the video file.
14    video_reader = cv2.VideoCapture(video_file_path)
15
16    # Get the width and height of the video.
17    original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))
18    original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))
19
20    # Initialize the VideoWriter Object to store the output video in the disk.
21    video_writer = cv2.VideoWriter(output_file_path, cv2.VideoWriter_fourcc('M', 'P', '4', 'V'),
22                                   video_reader.get(cv2.CAP_PROP_FPS), (original_video_width, original_video_height))
23
24    # Declare a queue to store video frames.
25    frames_queue = deque(maxlen = SEQUENCE_LENGTH)
26
27    # Initialize a variable to store the predicted action being performed in the video.
28    predicted_class_name = ""
29
30    # Iterate until the video is accessed successfully.
31    while video_reader.isOpened():
32
33        # Read the frame.
34        ok, frame = video_reader.read()
35
36        # Check if frame is not read properly then break the loop.
37        if not ok:
38            break
39
40        # Resize the Frame to fixed Dimensions.
41        resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))
42
43        # Normalize the resized frame by dividing it with 255 so that each pixel value then lies between 0 and 1.
44        normalized_frame = resized_frame / 255
45
46        # Appending the pre-processed frame into the frames list.
47        frames_queue.append(normalized_frame)
48
49        # Check if the number of frames in the queue are equal to the fixed sequence length.
50        if len(frames_queue) == SEQUENCE_LENGTH:
51
52            # Pass the normalized frames to the model and get the predicted probabilities.
53            predicted_labels_probabilities = LRCN_model.predict(np.expand_dims(frames_queue, axis = 0))[0]
54
55            # Get the index of class with highest probability.
56            predicted_label = np.argmax(predicted_labels_probabilities)
```

```

57
58     # Get the class name using the retrieved index.
59     predicted_class_name = CLASSES_LIST[predicted_label]
60
61     # Write predicted class name on top of the frame.
62     cv2.putText(frame, predicted_class_name, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
63
64     # Write The frame into the disk using the VideoWriter Object.
65     video_writer.write(frame)
66
67     # Release the VideoCapture and VideoWriter objects.
68     video_reader.release()
69     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 overlaid on it.

```

1 # Construct the output video path.
2 output_video_file_path = f'{test_videos_directory}/{video_title}-Output-SeqLen{SEQUENCE_LENGTH}.mp4'
3
4 # Perform Action Recognition on the Test Video.
5 predict_on_video(input_video_file_path, output_video_file_path, SEQUENCE_LENGTH)
6
7 # Display the output video.
8 VideoFileClip(output_video_file_path, audio=False, target_resolution=(300, None)).ipython_display()

```

100%  867/867 [00:02<00:00, 306.08it/s]

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.

```

1 def predict_single_action(video_file_path, SEQUENCE_LENGTH):
2     """
3     This function will perform single action recognition prediction on a video using the LRCN model.
4     Args:
5     video_file_path: The path of the video stored in the disk on which the action recognition is to be
6     performed.
7     SEQUENCE_LENGTH: The fixed number of frames of a video that can be passed to the model as one
8     sequence.
9     """
10
11     # Initialize the VideoCapture object to read from the video file.
12     video_reader = cv2.VideoCapture(video_file_path)
13
14     # Get the width and height of the video.

```

```

15 original_video_width = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))
16 original_video_height = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))
17
18 # Declare a list to store video frames we will extract.
19 frames_list = []
20
21 # Initialize a variable to store the predicted action being performed in the video.
22 predicted_class_name = ""
23
24 # Get the number of frames in the video.
25 video_frames_count = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT))
26
27 # Calculate the interval after which frames will be added to the list.
28 skip_frames_window = max(int(video_frames_count/SEQUENCE_LENGTH),1)
29
30 # Iterating the number of times equal to the fixed length of sequence.
31 for frame_counter in range(SEQUENCE_LENGTH):
32
33     # Set the current frame position of the video.
34     video_reader.set(cv2.CAP_PROP_POS_FRAMES, frame_counter * skip_frames_window)
35
36     # Read a frame.
37     success, frame = video_reader.read()
38
39     # Check if frame is not read properly then break the loop.
40     if not success:
41         break
42
43     # Resize the Frame to fixed Dimensions.
44     resized_frame = cv2.resize(frame, (IMAGE_HEIGHT, IMAGE_WIDTH))
45
46     # Normalize the resized frame by dividing it with 255 so that each pixel value then lies between 0 and 1.
47     normalized_frame = resized_frame / 255
48
49     # Appending the pre-processed frame into the frames list
50     frames_list.append(normalized_frame)
51
52     # Passing the pre-processed frames to the model and get the predicted probabilities.
53     predicted_labels_probabilities = LRCN_model.predict(np.expand_dims(frames_list, axis = 0))[0]
54
55     # Get the index of class with highest probability.
56     predicted_label = np.argmax(predicted_labels_probabilities)
57
58     # Get the class name using the retrieved index.
59     predicted_class_name = CLASSES_LIST[predicted_label]
60
61     # Display the predicted action along with the prediction confidence.
62     print(f'Action Predicted: {predicted_class_name}\nConfidence:
63 {predicted_labels_probabilities[predicted_label]}')
64
65     # Release the VideoCapture object.
66     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.

```

1 # Download the youtube video.
2 video_title = download_youtube_videos('https://youtu.be/fc3w827kwyA', test_videos_directory)
3
4 # Construct the input youtube video path
5 input_video_file_path = f'{test_videos_directory}/{video_title}.mp4'
6
7 # Perform Single Prediction on the Test Video.
8 predict_single_action(input_video_file_path, SEQUENCE_LENGTH)
9
10 # Display the input video.
11 VideoFileClip(input_video_file_path, audio=False, target_resolution=(300, None)).ipython_display()

```

Action Predicted: TaiChi

Confidence: 0.94

Summary

In this tutorial, we discussed a number of approaches to perform video classification and learned about the importance of the temporal aspect of data to gain higher accuracy in video classification and implemented two CNN + LSTM architectures in TensorFlow to perform Human Action Recognition on videos by utilizing the temporal as well as spatial information of the data.

We also learned to perform pre-processing on videos using the OpenCV library to create an image dataset, we also looked into getting youtube videos using just their URLs with the help of the Pafy library for testing our model.

Now let's discuss a limitation in our application that you should know about, our action recognizer cannot work on multiple people performing different activities. There should be only one person in the frame to correctly recognize the activity of that person by our action recognizer, this is because the data was in this manner, on which we had trained our model.

You can use some different dataset that has been annotated for more than one person's activity and also provides the bounding box coordinates of the person along with the activity he is performing, to overcome this limitation.

Or a hacky way is to crop out each person and perform activity recognition separately on each person but this will be computationally very expensive.