**human actions Classification**

**Installation**

In [1]:

%cd /content/

/content

In [2]:

!mkdir video\_classification

%cd video\_classification

/content/video\_classification

Install Dependencies: Run the following code block to install the necessary dependencies, including remotezip, tqdm, opencv-python, tf-models-official, and other required packages.

In [35]:

%%capture

!pip install remotezip tqdm opencv-python==4.5.2.52 opencv-python-headless==4.5.2.52 tf-models-official

!pip install remotezip

!pip install tf-models-official

!pip install -q mediapy

Import Libraries: Make sure to import the essential libraries for video processing, data manipulation, visualization, and deep learning operations. Include the following modules in your code:

**NOTE!!**: Below code might give you an error but running it again would solve the issue

In [36]:

import tqdm

import random

import pathlib

import itertools

import collections

import os

import cv2

import numpy as np

import remotezip as rz

import seaborn as sns

import matplotlib.pyplot as plt

import keras

import tensorflow as tf

import tensorflow\_hub as hub

from tensorflow.keras import layers

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.losses import SparseCategoricalCrossentropy

*# Import the MoViNet model from TensorFlow Models (tf-models-official) for the MoViNet model*

from official.projects.movinet.modeling import movinet

from official.projects.movinet.modeling import movinet\_model

import mediapy as media

With these installation and import steps completed, you are now ready to leverage the power of MoViNet for accurate video classification.

Feel free to customize the installation and import instructions based on your specific needs.

**Download a subset of the UCF101 dataset**

def get\_class(fname):

""" Retrieve the name of the class given a filename.

Args:

fname: Name of the file in the UCF101 dataset.

Returns:

Class that the file belongs to.

"""

return fname.split('\_')[-3]

def list\_files\_per\_class(zip\_url):

"""

List the files in each class of the dataset given the zip URL.

Args:

zip\_url: URL from which the files can be unzipped.

Return:

files: List of files in each of the classes.

"""

files = []

with rz.RemoteZip(URL) as zip:

for zip\_info in zip.infolist():

files.append(zip\_info.filename)

return files

def get\_class(fname):

"""

Retrieve the name of the class given a filename.

Args:

fname: Name of the file in the UCF101 dataset.

Return:

Class that the file belongs to.

"""

return fname.split('\_')[-3]

def get\_files\_per\_class(files):

"""

Retrieve the files that belong to each class.

Args:

files: List of files in the dataset.

Return:

Dictionary of class names (key) and files (values).

"""

files\_for\_class = collections.defaultdict(list)

for fname in files:

class\_name = get\_class(fname)

files\_for\_class[class\_name].append(fname)

return files\_for\_class

files\_subset = dict()

for class\_name in classes:

class\_files = files\_for\_class[class\_name]

files\_subset[class\_name] = class\_files[:files\_per\_class]

return files\_subset

def download\_from\_zip(zip\_url, to\_dir, file\_names):

with rz.RemoteZip(zip\_url) as zip:

for fn in tqdm.tqdm(file\_names):

class\_name = get\_class(fn)

zip.extract(fn, str(to\_dir / class\_name))

unzipped\_file = to\_dir / class\_name / fn

fn = pathlib.Path(fn).parts[-1]

output\_file = to\_dir / class\_name / fn

unzipped\_file.rename(output\_file)

def split\_class\_lists(files\_for\_class, count):

split\_files = []

remainder = {}

for cls in files\_for\_class:

split\_files.extend(files\_for\_class[cls][:count])

remainder[cls] = files\_for\_class[cls][count:]

return split\_files, remainder

def download\_ucf\_101\_subset(zip\_url, num\_classes, splits,

files = list\_files\_from\_zip\_url(zip\_url)

for f in files:

path = os.path.normpath(f)

tokens = path.split(os.sep)

if len(tokens) <= 2:

files.remove(f) *# Remove that item from the list if it does not have a filename*

files\_for\_class = get\_files\_per\_class(files)

*# classes = list(files\_for\_class.keys())[:num\_classes]*

classes = classes\_1

for cls in classes:

random.shuffle(files\_for\_class[cls])

*# Only use the number of classes you want in the dictionary*

files\_for\_class = {x: files\_for\_class[x] for x in classes}

dirs = {}

for split\_name, split\_count in splits.items():

print(split\_name, ":")

split\_dir = download\_dir / split\_name

split\_files, files\_for\_class = split\_class\_lists(files\_for\_class, split\_count)

download\_from\_zip(zip\_url, split\_dir, split\_files)

dirs[split\_name] = split\_dir

return dirs

def format\_frames(frame, output\_size):

frame = tf.image.convert\_image\_dtype(frame, tf.float32)

frame = tf.image.resize\_with\_pad(frame, \*output\_size)

return frame

def frames\_from\_video\_file(video\_path, n\_frames, output\_size = (224,224), frame\_step = 15):

*# Read each video frame by frame*

result = []

src = cv2.VideoCapture(str(video\_path))

video\_length = src.get(cv2.CAP\_PROP\_FRAME\_COUNT)

need\_length = 1 + (n\_frames - 1) \* frame\_step

if need\_length > video\_length:

start = 0

else:

max\_start = video\_length - need\_length

start = random.randint(0, max\_start + 1)

src.set(cv2.CAP\_PROP\_POS\_FRAMES, start)

*# ret is a boolean indicating whether read was successful, frame is the image itself*

ret, frame = src.read()

result.append(format\_frames(frame, output\_size))

for \_ in range(n\_frames - 1):

for \_ in range(frame\_step):

ret, frame = src.read()

if ret:

frame = format\_frames(frame, output\_size)

result.append(frame)

else:

result.append(np.zeros\_like(result[0]))

src.release()

result = np.array(result)[..., [2, 1, 0]]

return result

def list\_files\_from\_zip\_url(zip\_url):

files = []

with rz.RemoteZip(zip\_url) as zip:

for zip\_info in zip.infolist():

files.append(zip\_info.filename)

return files

class FrameGenerator:

def \_\_init\_\_(self, path, n\_frames, training = False):

self.path = path

self.n\_frames = n\_frames

self.training = training

return files\_subset

files\_subset = select\_subset\_of\_classes(files\_for\_class, classes\_1, FILES\_PER\_CLASS)

list(files\_subset.keys())

def download\_from\_zip(zip\_url, to\_dir, file\_names):

with rz.RemoteZip(zip\_url) as zip:

for fn in tqdm.tqdm(file\_names):

class\_name = get\_class(fn)

zip.extract(fn, str(to\_dir / class\_name))

unzipped\_file = to\_dir / class\_name / fn

fn = pathlib.Path(fn).parts[-1]

output\_file = to\_dir / class\_name / fn

unzipped\_file.rename(output\_file)

**Create the training and test datasets**

batch\_size = 8

num\_frames = 8

output\_signature = (tf.TensorSpec(shape = (None, None, None, 3), dtype = tf.float32),

tf.TensorSpec(shape = (), dtype = tf.int16))

train\_ds = tf.data.Dataset.from\_generator(FrameGenerator(subset\_paths['train'], num\_frames, training = True),

output\_signature = output\_signature)

train\_ds = train\_ds.batch(batch\_size)

print(f"Shape: {frames.shape}")

print(f"Label: {labels.shape}")

Shape: (8, 8, 224, 224, 3)

Label: (8,)

In [40]:

videos, labels = next(iter(train\_ds))

media.show\_videos(videos.numpy(), codec='gif', fps=8)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

**Download a pre-trained MoViNet mode**

model\_id = 'a0'

resolution = 224

tf.keras.backend.clear\_session()

backbone = movinet.Movinet(model\_id=model\_id)

backbone.trainable = False

*# Set num\_classes=600 to load the pre-trained weights from the original model*

model = movinet\_model.MovinetClassifier(backbone=backbone, num\_classes=600)

model.build([None, None, None, None, 3])

*# Load pre-trained weights*

!wget https://storage.googleapis.com/tf\_model\_garden/vision/movinet/movinet\_a0\_base.tar.gz -O movinet\_a0\_base.tar.gz -q

!tar -xvf movinet\_a0\_base.tar.gz

checkpoint\_dir = f'movinet\_{model\_id}\_base'

checkpoint\_path = tf.train.latest\_checkpoint(checkpoint\_dir)

checkpoint = tf.train.Checkpoint(model=model)

status = checkpoint.restore(checkpoint\_path)

status.assert\_existing\_objects\_matched()

**Output:**





