MIT | Academy of Engineering

(An Autonomous Institute Affiliated to Savitribai Phule Pune University)

Predictive Analytics

PROJECT REPORT

ON

Human Pose Detection

Submitted By-

Shripad Khandare - 202201040103
shripad.khandare@mitaoe.ac.in

Yash Gunjal - 202201040106
yash.gunjal@mitaoe.ac.in

Ritesh Patil - 20220109106
ritesh.patil@mitaoe.ac.in

Batch - MDM 1

Guidance by -

Prof. Vaishali Wangilkar

Guidance by-

Prof. Shubhangi Kale

Academic Year: 2024-25

Human Pose Detection

● Introduction:

Human pose detection is a part of computer vision that helps computers understand body movements. It finds the position of a person's body (like arms, legs, and hands) from a photo or video. In this project, we focus on detecting human actions like sign language, dance poses, and yoga poses.

These types of detection are very useful:

- Sign language helps people who cannot speak and communicate.
- Dance pose detection is useful in training, competitions, or fitness apps.
- Yoga pose detection helps in fitness and health monitoring.

To build this, we use a deep learning model called MoViNet (Mobile Video Network). MoViNet is a fast and smart model that works well for video-based tasks. It understands what action is happening in a video.

We used a video dataset called UCF101, which has 101 types of human actions. From this, we selected 25 actions that look similar to sign language, dance, and yoga poses.

The model is trained using TensorFlow and run on Google Colab, where we used GPU support to speed up the training process. Once trained, the model can take a video input and tell us what action is being performed.

● Background:

Human pose detection is getting more popular because it helps machines understand body movements. This is very useful in areas like:

- Healthcare – to check yoga poses or exercises
- Communication – for detecting sign language
- Entertainment – like dance pose analysis in games or learning apps

Many deep learning models are used for this. But most of them need powerful computers. So, we need a

model that is both fast and accurate.

That's why we use MoViNet – a special model made by Google. It works well even on mobile and small devices. MoViNet is trained on videos, not just images. So, it's good for detecting actions over time, like in dance or yoga.

We use a dataset called UCF101, which has short videos of different human activities. From this, we selected 25 actions that match with our goal — like poses from dance, yoga, or hand movements.

By using TensorFlow and Google Colab, we can train and test the model easily and get good results.

● Concepts and Algorithms Used:

Main Algorithm: MoViNet (Mobile Video Network)

MoViNet is a deep learning model made for video understanding. It is different from image models because it doesn't look at just one frame (image) — it looks at many frames from a video to understand what's happening over time.

How It Works:

- A video is divided into frames (like pictures).
- Each group of frames is passed into MoViNet.
- MoViNet learns the motion (movement) and shape of body parts (pose).
- Then it predicts what action is being done — like yoga, boxing, or dancing.

Dataset Used: UCF101

UCF101 is a video dataset with 101 types of human actions like:

- Boxing
- Dancing
- Jumping
- Doing yoga

Advantages

- Fast & Lightweight: MoViNet is very fast and works even on mobile or low-power devices.

- Real-time Use: It can be used in real-time apps like fitness or sign language detection.
- Good Accuracy: Even with a few training steps, it gives high accuracy (around 90%+).
- Works on Videos: Many models work only on images, but MoViNet works well on videos (multiple frames).

Limitations

- Needs Many Videos: It needs lots of video data to train better.
- Needs Good GPU: Training is slow without GPU.
- Not Always 100% Accurate: Sometimes it can confuse similar actions (e.g. “Boxing” vs “Punch”).

● How the Algorithm (MoViNet) applicable for Problem:

To solve this, we need a model that can:

- Understand human body movements
- Detect changes in poses over time
- Work on videos, not just single images

Why MoViNet is a good choice:

1. MoViNet works on videos – It takes a series of frames (images) from the video and understands movement.
2. It uses 3D Convolution – So it looks at both spatial (pose in one image) and temporal (how pose changes in next frames) data.
3. It can classify actions – After training, it can tell which action is happening (like “Boxing”, “Yoga”, “Dancing” etc.).

Example:

- If someone does a yoga pose in a video, the model sees 8 frames from that video.
- It checks the body position in each frame.

- Then, based on training, it says: “This looks like *Yoga Pose*.”

- Diagramatic representation of execution of project

- Dataset Used: UCF101 – Action Recognition Dataset

1. Name

UCF101 (University of Central Florida 101)

2. Size

- Total videos: 13,320
- Total classes: 101 different human actions
- Video format: .avi
- Resolution: Around 320x240 pixels
- Frame rate: Around 25 FPS

In the project:
used only 25 selected classes
And for each class:

- 30 training videos

- 10 validation videos
- 10 test videos
  Total used: ~1250 videos

3. Attributes (features)

Each video has:

- Video frames (images over time)
- Action label (example: Boxing, Yoga, Archery, etc.)

Example label from filename:
v_BoxingPunchingBag_g01_c01.avi

Here:

- BoxingPunchingBag is the action class
- g01_c01 are just group and clip IDs

4. Source and Links

- Dataset page: UCF101 website
- Direct ZIP file used in code:
  https://storage.googleapis.com/thumos14_files/UCF101_videos.zip

- Data preprocessing tasks performed

- Code

```
#@title
```

```
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

def select_subset_of_classes(files_for_class, classes, files_per_class):
 """ Create a dictionary with the class name and a subset of the files in that class.

   Args:
     files_for_class: Dictionary of class names (key) and files (values).
     classes: List of classes.
     files_per_class: Number of files per class of interest.

   Returns:
     Dictionary with class as key and list of specified number of video files in that 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):
 """ Download the contents of the zip file from the zip URL.

   Args:
     zip_url: A URL with a zip file containing data.
```

```
    to_dir: A directory to download data to.
    file_names: Names of files to download.
  """
  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):
  """
    Returns the list of files belonging to a subset of data as well as the remainder of
    files that need to be downloaded.

    Args:
      files_for_class: Files belonging to a particular class of data.
      count: Number of files to download.

    Return:
      split_files: Files belonging to the subset of data.
      remainder: Dictionary of the remainder of files that need to be downloaded.
  """
  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, download_dir,classes_1):
  """ Download a subset of the UCF101 dataset and split them into various parts, such as
    training, validation, and test.

    Args:
      zip_url: A URL with a ZIP file with the data.
      num_classes: Number of labels.
```

```
    splits: Dictionary specifying the training, validation, test, etc. (key) division of data
            (value is number of files per split).
    download_dir: Directory to download data to.

  Return:
    Mapping of the directories containing the subsections of data.
  """
  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):
  """
    Pad and resize an image from a video.

    Args:
```

```
    frame: Image that needs to resized and padded.
    output_size: Pixel size of the output frame image.

  Return:
    Formatted frame with padding of specified 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):
  """
    Creates frames from each video file present for each category.

    Args:
      video_path: File path to the video.
      n_frames: Number of frames to be created per video file.
      output_size: Pixel size of the output frame image.

    Return:
      An NumPy array of frames in the shape of (n_frames, height, width, channels).
  """
  # 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):
  """ List the files in each class of the dataset given a URL with the zip file.

    Args:
      zip_url: A URL from which the files can be extracted from.

    Returns:
      List of files in each of the classes.
  """
  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):
    """ Returns a set of frames with their associated label.

      Args:
        path: Video file paths.
        n_frames: Number of frames.
        training: Boolean to determine if training dataset is being created.
    """
    self.path = path
    self.n_frames = n_frames
```

```
    self.training = training
    self.class_names = sorted(set(p.name for p in self.path.iterdir() if p.is_dir()))
    self.class_ids_for_name = dict((name, idx) for idx, name in enumerate(self.class_names))

  def get_files_and_class_names(self):
    video_paths = list(self.path.glob('*/*.avi'))
    classes = [p.parent.name for p in video_paths]
    return video_paths, classes

  def __call__(self):
    video_paths, classes = self.get_files_and_class_names()

    pairs = list(zip(video_paths, classes))

    if self.training:
      random.shuffle(pairs)

    for path, name in pairs:
      video_frames = frames_from_video_file(path, self.n_frames)
      label = self.class_ids_for_name[name] # Encode labels
      yield video_frames, label
```

```
def list_files_from_zip_url(zip_url):
  """ List the files in each class of the dataset given a URL with the zip file.

    Args:
      zip_url: A URL from which the files can be extracted from.

    Returns:
      List of files in each of the classes.
  """
  files = []
  with rz.RemoteZip(zip_url) as zip:
    for zip_info in zip.infolist():
      files.append(zip_info.filename)
  return files
files = list_files_from_zip_url(URL)
files = [f for f in files if f.endswith('.avi')]
files[:10]
```

```
['UCF101/v_ApplyEyeMakeup_g01_c01.avi',
 'UCF101/v_ApplyEyeMakeup_g01_c02.avi',
 'UCF101/v_ApplyEyeMakeup_g01_c03.avi',
 'UCF101/v_ApplyEyeMakeup_g01_c04.avi',
 'UCF101/v_ApplyEyeMakeup_g01_c05.avi',
 'UCF101/v_ApplyEyeMakeup_g01_c06.avi',
 'UCF101/v_ApplyEyeMakeup_g02_c01.avi',
 'UCF101/v_ApplyEyeMakeup_g02_c02.avi',
 'UCF101/v_ApplyEyeMakeup_g02_c03.avi',
 'UCF101/v_ApplyEyeMakeup_g02_c04.avi']
```

Begin with a few videos and a limited number of classes for training. After running the above code block, notice that the class name is included in the filename of each video.

Define the `get_class` function that retrieves the class name from a filename. Then, create a function called `get_files_per_class` which converts the list of all files (`files` above) into a dictionary listing the files for each class:

```
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 get_files_per_class(files):
  """ Retrieve the files that belong to each class.

    Args:
      files: List of files in the dataset.

    Returns:
      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
```

Once you have the list of files per class, you can choose how many classes you would like to use and how many videos you would like per class in order to create your dataset.

Generate + Code + Markdown

```python
FILES_PER_CLASS = 100
```

Python

```python

```

Python

```python
files_for_class = get_files_per_class(files)
classes = list(files_for_class.keys())
```

Python

All the classes of UCF101 dataset

```python
classes
```

Python

```
['ApplyEyeMakeup',
 'ApplyLipstick',
 'Archery',
 'BabyCrawling',
 'BalanceBeam',
 'BandMarching',
 'BaseballPitch',
 'BasketballDunk',
 'Basketball',
 'BenchPress',
 'Biking',
 'Billiards',
 'BlowDryHair',
 'BlowingCandles',
 'BodyWeightSquats',
 'Bowling',
 'BoxingPunchingBag',
 'BoxingSpeedBag',
 'BreastStroke',
 'BrushingTeeth',
 'CleanAndJerk',
 'CliffDiving',
 'CricketBowling',
 'CricketShot',
 'CuttingInKitchen',
...
 'VolleyballSpiking',
 'WalkingWithDog',
 'WallPushups',
 'WritingOnBoard',
 'YoYo']
```

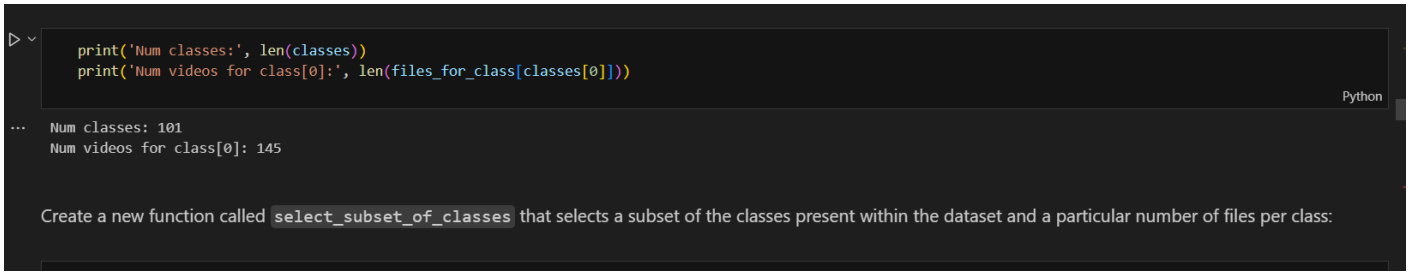
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

These are the selected classes from the above classes lis

```
classes_1 = ['Archery',
'BabyCrawling',
'BoxingPunchingBag',
'BoxingSpeedBag',
'CliffDiving',
'CuttingInKitchen',
'Diving',
'Fencing',
'Hammering',
'HammerThrow',
'HighJump',
'HulaHoop',
```

```
'IceDancing',
'JavelinThrow',
'LongJump',
'Mixing',
'MoppingFloor',
'PoleVault',
'Punch',
'SalsaSpin',
'Shotput',
'SumoWrestling',
'TaiChi',
'ThrowDiscus',
'TrampolineJumping']

NUM_CLASSES = len(classes_1)
```




```
def select_subset_of_classes(files_for_class, classes, files_per_class):
 """ Create a dictionary with the class name and a subset of the files in that class.

  Args:
   files_for_class: Dictionary of class names (key) and files (values).
   classes: List of classes.
   files_per_class: Number of files per class of interest.

  Returns:
   Dictionary with class as key and list of specified number of video files in that 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
```

```python
files_subset = select_subset_of_classes(files_for_class, classes_1, FILES_PER_CLASS)
list(files_subset.keys())
```

Python

```
['Archery',
 'BabyCrawling',
 'BoxingPunchingBag',
 'BoxingSpeedBag',
 'CliffDiving',
 'CuttingInKitchen',
 'Diving',
 'Fencing',
 'Hammering',
 'HammerThrow',
 'HighJump',
 'HulaHoop',
 'IceDancing',
 'JavelinThrow',
 'LongJump',
 'Mixing',
 'MoppingFloor',
 'PoleVault',
 'Punch',
 'SalsaSpin',
 'Shotput',
 'SumoWrestling',
 'TaiChi',
 'ThrowDiscus',
 'TrampolineJumping']
```

Define helper functions that split the videos into training, validation, and test sets. The videos are downloaded from a URL with the zip file, and placed into their respective subdirectiories.

```python
def download_from_zip(zip_url, to_dir, file_names):
  """ Download the contents of the zip file from the zip URL.

    Args:
      zip_url: A URL with a zip file containing data.
      to_dir: A directory to download data to.
      file_names: Names of files to download.
  """
  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):
  """ Returns the list of files belonging to a subset of data as well as the remainder of
    files that need to be downloaded.

    Args:
      files_for_class: Files belonging to a particular class of data.
      count: Number of files to download.
```

```
  Returns:
    Files belonging to the subset of data and dictionary of the remainder of files that need to be downloaded.
  """
  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
```

The following `download_ucf_101_subset` function allows you to download a subset of the UCF101 dataset and split it into the training, validation, and test sets. You can specify the number of classes that you would like to use. The `splits` argument allows you to pass in a dictionary in which the key values are the name of subset (example: "train") and the number of videos you would like to have per class.

```
def download_ucf_101_subset(zip_url, num_classes, splits, download_dir,classes_1):
  """ Download a subset of the UCF101 dataset and split them into various parts, such as
    training, validation, and test.

    Args:
      zip_url: A URL with a ZIP file with the data.
      num_classes: Number of labels.
      splits: Dictionary specifying the training, validation, test, etc. (key) division of data
              (value is number of files per split).
      download_dir: Directory to download data to.

    Return:
      Mapping of the directories containing the subsections of data.
  """
  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
```

The below URL contains a zip file with the UCF 101 dataset. Create a function that uses the `remotezip` library to examine the contents of the zip file in that URL:

```
URL = 'https://storage.googleapis.com/thumos14_files/UCF101_videos.zip'
```
Python

```
download_dir = pathlib.Path('./UCF101_subset/')
subset_paths = download_ucf_101_subset(URL,
                                       num_classes = NUM_CLASSES,
                                       splits = {"train": 30, "val": 10, "test": 10},
                                       download_dir = download_dir,
                                       classes_1=classes_1)
```
Python

```
train :
100%|██████████| 750/750 [12:25<00:00,  1.01it/s]
val :
100%|██████████| 250/250 [03:39<00:00,  1.14it/s]
test :
100%|██████████| 250/250 [03:45<00:00,  1.11it/s]
```

After downloading the data, you should now have a copy of a subset of the UCF101 dataset. Run the following code to print the total number of videos you have amongst all your subsets of data.

```
video_count_train = len(list(download_dir.glob('train/*/*.avi')))
video_count_val = len(list(download_dir.glob('val/*/*.avi')))
video_count_test = len(list(download_dir.glob('test/*/*.avi')))
video_total = video_count_train + video_count_val + video_count_test
print(f"Total videos: {video_total}")
```
Python

```
Total videos: 1250
```

```
video_count_train = len(list(download_dir.glob('train/*/*.avi')))
video_count_val = len(list(download_dir.glob('val/*/*.avi')))
video_count_test = len(list(download_dir.glob('test/*/*.avi')))
video_total = video_count_train + video_count_val + video_count_test
print(f"Total videos: {video_total}")
```

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)

test_ds = tf.data.Dataset.from_generator(FrameGenerator(subset_paths['test'], num_frames),
                                         output_signature = output_signature)
test_ds = test_ds.batch(batch_size)

val_ds = tf.data.Dataset.from_generator(FrameGenerator(subset_paths['val'], num_frames),
                                        output_signature = output_signature)
val_ds = val_ds.batch(batch_size)
```

The labels generated here represent the encoding of the classes. For instance, 'ApplyEyeMakeup' is mapped to the integer Take a look at the labels of the training data to ensure that the dataset has been sufficiently shuffled.

```
for frames, labels in train_ds.take(NUM_CLASSES):
  print(labels)
```

Python

```
tf.Tensor([24  5 11  0  9 14 22 19], shape=(8,), dtype=int16)
tf.Tensor([ 1 18 14  2  2 22 20  9], shape=(8,), dtype=int16)
tf.Tensor([22  9  9 19 15  0 15  7], shape=(8,), dtype=int16)
tf.Tensor([20 24 17 16  1  7  3 24], shape=(8,), dtype=int16)
tf.Tensor([ 1 15  5  6  6 14 12 24], shape=(8,), dtype=int16)
tf.Tensor([19  8  4 10  4 18 23  5], shape=(8,), dtype=int16)
tf.Tensor([23  7  9 20  2  0 16 19], shape=(8,), dtype=int16)
tf.Tensor([10  6  1  3  2 24 17 12], shape=(8,), dtype=int16)
tf.Tensor([13 21  5 22  9 16  4  1], shape=(8,), dtype=int16)
tf.Tensor([20 19 14  7  6 17 13 24], shape=(8,), dtype=int16)
tf.Tensor([ 7  4 15  1  5 16  8 20], shape=(8,), dtype=int16)
tf.Tensor([21  2 15 17  1  4 15  7], shape=(8,), dtype=int16)
tf.Tensor([11  6  8 12 21 24 16 18], shape=(8,), dtype=int16)
tf.Tensor([19  3 15 24 22 23 21  1], shape=(8,), dtype=int16)
tf.Tensor([11 23 24 24 12 14  8 11], shape=(8,), dtype=int16)
tf.Tensor([ 1  8 22  2 10  2 22 18], shape=(8,), dtype=int16)
tf.Tensor([18 20 12  6 12 13 15  8], shape=(8,), dtype=int16)
tf.Tensor([14 17  2 11 10 16 24 18], shape=(8,), dtype=int16)
tf.Tensor([12  0 12 10  4 13 23 16], shape=(8,), dtype=int16)
tf.Tensor([ 7 16  9  7  6 16 23 16], shape=(8,), dtype=int16)
tf.Tensor([ 8  0  6 21 17 13 12  3], shape=(8,), dtype=int16)
tf.Tensor([14  0  3 13 10 24  3 18], shape=(8,), dtype=int16)
tf.Tensor([ 2  5 20 21 18  6  1 22], shape=(8,), dtype=int16)
tf.Tensor([ 0 12 11  5  3 23  9  4], shape=(8,), dtype=int16)
tf.Tensor([ 4 16  4 18 20 23  5 23], shape=(8,), dtype=int16)
```

Take a look at the shape of the data.

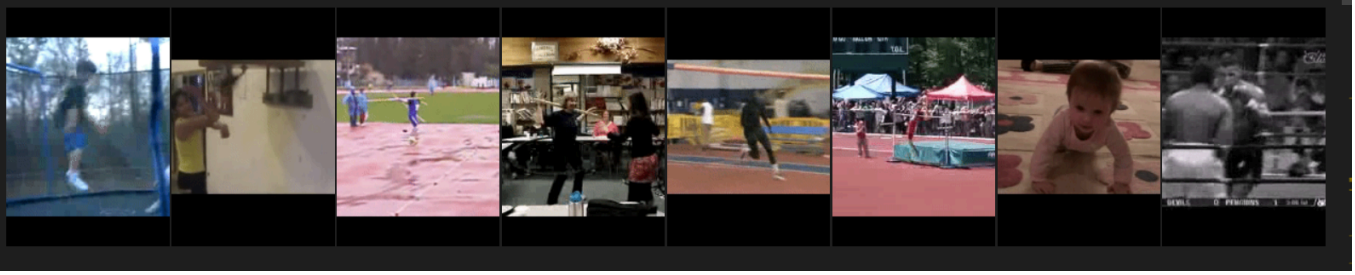
```python
print(f"Shape: {frames.shape}")
print(f"Label: {labels.shape}")
```

Python

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

```python
videos, labels = next(iter(train_ds))
media.show_videos(videos.numpy(), codec='gif', fps=8)
```

Python



```python
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()
```

```
movinet_a0_base/
movinet_a0_base/checkpoint
movinet_a0_base/ckpt-1.data-00000-of-00001
movinet_a0_base/ckpt-1.index

<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x7be8c6ad63b0>
```

To build a classifier, create a function that takes the backbone and the number of classes in a dataset. The `build_classifier` function will take the backbone and the number of classes in a dataset to build the classifier. In this case, the new classifier will take a `num_classes` outputs (10 classes for this subset of UCF101).

```python
def build_classifier(batch_size, num_frames, resolution, backbone, num_classes):
  """Builds a classifier on top of a backbone model."""
  model = movinet_model.MovinetClassifier(
      backbone=backbone,
      num_classes=num_classes)
  model.build([batch_size, num_frames, resolution, resolution, 3])

  return model
```

Python

```python
batch_size, num_frames, resolution, backbone, NUM_CLASSES
```

Python

```
(8,
```

Python

```
(8,
 8,
 224,
 <official.projects.movinet.modeling.movinet.Movinet at 0x7be8d61bad40>,
 25)
```

```python
model = build_classifier(batch_size, num_frames, resolution, backbone, NUM_CLASSES)
```

Python

```python
model.summary()
```

Python

```
Model: "movinet_classifier_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image (InputLayer)          [(None, None, None, Non   0
                             e, 3)]

 movinet (Movinet)           ({'stem': (None, None,    911583
                             None, None, 8),
                              'block0_layer0': (None
                             , None, None, None, 8),
                              'block1_layer0': (None
                             , None, None, None, 32)
                             , 'block1_layer1': (Non
                             e, None, None, None, 32
                             ),
                              'block1_layer2': (None
                             , None, None, None, 32)
                             , 'block2_layer0': (Non
                             e, None, None, None, 56
                             ),
                              'block2_layer1': (None
                             , None, None, None, 56)
                             , 'block2_layer2': (Non
                             e, None, None, None, 56
...
Total params: 1947896 (7.43 MB)
Trainable params: 1036313 (3.95 MB)
Non-trainable params: 911583 (3.48 MB)
_________________________________________________________________
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

For this tutorial, choose the `tf.keras.optimizers.Adam` optimizer and the `tf.keras.losses.SparseCategoricalCrossentropy` loss function. Use the metrics argument to the view the accuracy of the model performance at every step.

```python
num_epochs = 2

loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001)

model.compile(loss=loss_obj, optimizer=optimizer, metrics=['accuracy'])
```

Python

Train the model. After two epochs, observe a low loss with high accuracy for both the training and test sets.

```python
train_ds, test_ds
```

Python

```
(<_BatchDataset element_spec=(TensorSpec(shape=(None, None, None, None, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int16, name=None))>,
 <_BatchDataset element_spec=(TensorSpec(shape=(None, None, None, None, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int16, name=None))>)
```


```python
train_ds, test_ds
```

Python

```
(<_BatchDataset element_spec=(TensorSpec(shape=(None, None, None, None, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int16, name=None))>,
 <_BatchDataset element_spec=(TensorSpec(shape=(None, None, None, None, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int16, name=None))>)
```


```python
results = model.fit(train_ds,
                    validation_data=test_ds,
                    epochs=num_epochs,
                    validation_freq=1,
                    verbose=1)
```

Python

```
Epoch 1/2
94/94 [==============================] - 202s 2s/step - loss: 1.3535 - accuracy: 0.6880 - val_loss: 0.4006 - val_accuracy: 0.8920
Epoch 2/2
94/94 [==============================] - 117s 1s/step - loss: 0.2343 - accuracy: 0.9387 - val_loss: 0.3205 - val_accuracy: 0.9000
```

Save and load model

This block contains codes for saving and loading trained models

```python
model.save('saved_model/my_model')
```

```
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d6036ce0>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d602d1b0>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d5014820>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d50f2320>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4f6e980>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4fed390>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4e8ba60>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4d02b30>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4d93580>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4c5b9d0>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4cc2fb0>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d4b4a8f0>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d03ca6e0>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be8d0463280>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.SpatialAveragePool3D object at 0x7be9791c10f0>, because it is not built.
flow:Skipping full serialization of Keras layer <keras.src.layers.regularization.dropout.Dropout object at 0x7be8e88c9990>, because it is not built.
flow:Skipping full serialization of Keras layer <official.vision.modeling.layers.nn_layers.TemporalSoftmaxPool object at 0x7be8c669ec50>, because it is not built.
```

```python
import shutil
```

```python
import shutil

# Specify the folder path to be downloaded
folder_path = '/content/saved_model/my_model'  # Replace with the actual path to your folder

# Create a zip file of the folder
shutil.make_archive('/content/my_model', 'zip', folder_path)
```

```python
# Download the zip file
from google.colab import files

files.download('/content/my_model.zip')
```

```python
!unzip /content/my_model.zip -d /content/my_model
```

```
Archive:  /content/my_model.zip
   creating: /content/my_model/assets/
   creating: /content/my_model/variables/
  inflating: /content/my_model/keras_metadata.pb
  inflating: /content/my_model/saved_model.pb
  inflating: /content/my_model/fingerprint.pb
  inflating: /content/my_model/variables/variables.index
  inflating: /content/my_model/variables/variables.data-00000-of-00001
```

```python
model = tf.keras.models.load_model('/content/my_model')


# Check its architecture
model.summary()
```

```
WARNING:tensorflow:`tf.keras.layers.experimental.SyncBatchNormalization` endpoint is deprecated and will be removed in a future release. Please use `tf.keras.layer
Model: "movinet_classifier_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image (InputLayer)          [(None, None, None, Non   0
                             e, 3)]

 movinet (Movinet)           ({'stem': (None, None,    911583
                             None, None, 8),
                              'block0_layer0': (None
                             , None, None, None, 8),
                              'block1_layer0': (None
                             , None, None, None, 32)
                             , 'block1_layer1': (Non
                             e, None, None, None, 32
                             ),
                              'block1_layer2': (None
                             , None, None, None, 32)
                             , 'block2_layer0': (Non
                             e, None, None, None, 56
                             ),
                              'block2_layer1': (None
                             , None, None, None, 56)
                             , 'block2_layer2': (Non
                             e, None, None, None, 56
...
Total params: 1947896 (7.43 MB)
Trainable params: 1036313 (3.95 MB)
Non-trainable params: 911583 (3.48 MB)
_________________________________________________________________
```

Evaluate the model

The model achieved high accuracy on the training dataset. Next, use Keras `Model.evaluate` to evaluate it on the test set.

```python
def plot_history(history):
  """
    Plotting training and validation learning curves.

    Args:
      history: model history with all the metric measures
  """
  fig, (ax1, ax2) = plt.subplots(2)

  fig.set_size_inches(18.5, 10.5)

  # Plot loss
  ax1.set_title('Loss')
  ax1.plot(history.history['loss'], label = 'train')
  ax1.plot(history.history['val_loss'], label = 'test')
  ax1.set_ylabel('Loss')

  # Determine upper bound of y-axis
  max_loss = max(history.history['loss'] + history.history['val_loss'])

  ax1.set_ylim([0, np.ceil(max_loss)])
```

```
ax1.set_xlabel('Epoch')
ax1.legend(['Train', 'Validation'])

# Plot accuracy
ax2.set_title('Accuracy')
ax2.plot(history.history['accuracy'],  label = 'train')
ax2.plot(history.history['val_accuracy'], label = 'test')
ax2.set_ylabel('Accuracy')
ax2.set_ylim([0, 1])
ax2.set_xlabel('Epoch')
ax2.legend(['Train', 'Validation'])
plt.savefig("loss.png")
plt.show()

plot_history(results)
```

```
model.evaluate(test_ds, return_dict=True)
```

```
32/32 [==============================] - 30s 933ms/step - loss: 0.3186 - accuracy: 0.9000
{'loss': 0.31863898038864136, 'accuracy': 0.8999999761581421}
```

```
get_actual_predicted_labels(model,dataset):
  """
  Create a list of actual ground truth values and the predictions from the model.

  Args:
    dataset: An iterable data structure, such as a TensorFlow Dataset, with features and labels.

  Return:
    Ground truth and predicted values for a particular dataset.
  """
  actual = [labels for _, labels in dataset.unbatch()]
  predicted = model.predict(dataset)

  actual = tf.stack(actual, axis=0)
  predicted = tf.concat(predicted, axis=0)
  predicted = tf.argmax(predicted, axis=1)

  return actual, predicted
```

```
actual, predicted = get_actual_predicted_labels(model,test_ds)
```

```
32/32 [==============================] - 44s 1s/step
```

```
labels = classes_1
```

```
print(actual, tf.cast(predicted, tf.int16),labels, len(labels),sep = '\n')
```

```
tf.Tensor(
[ 1  1  1  1  1  1  1  1  1  8  8  8  8  8  8  8  8 21 21 21 21
 21 21 21 21 21 21  4  4  4  4  4  4  4  4  4 23 23 23 23 23 23 23 23
 23 23  5  5  5  5  5  5  5  5  5 20 20 20 20 20 20 20 20 20  9  9
  9  9  9  9  9  9 10 10 10 10 10 10 10 10 10 10 17 17 17 17 17 17
 17 17 17 17 18 18 18 18 18 18 18 18 18 18 15 15 15 15 15 15 15 15 15 15
  2  2  2  2  2  2  2  2  2 19 19 19 19 19 19 19 19 19  3  3  3  3
  3  3  3  3  3 22 22 22 22 22 22 22 22 22 14 14 14 14 14 14 14 14
 14 14  7  7  7  7  7  7  7  7  7  6  6  6  6  6  6  6  6  6  0  0
  0  0  0  0  0  0  0 24 24 24 24 24 24 24 24 24 11 11 11 11 11 11
 11 11 11 11 16 16 16 16 16 16 16 16 16 16 12 12 12 12 12 12 12 12 12
 13 13 13 13 13 13 13 13 13], shape=(250,), dtype=int16)
tf.Tensor(
[ 1  1 16  1  1  1  1  1  1  8  8  8  8  8  8  8  8 21 21 21 21
 21 21 21 21 21 21  4  4  4  4  4  4  4  4  4 23  8 23  8 20 20 20 23
 23 23  5  5  5  5  5  5  5  5  5 20 20 20 20 20 20 20 20 20  9  5
  9  9  9  9  9  9  9 10 10 10 10 10 10 10 10 10 10 17 10 17 10 17 17
 17 17 10 14 18 18 18 18 18 18 18 18 18 18 15 15 15 15 15 15 15 15 15 15
  2  2  2  2  2  2  2  2  2 19 19 19 19 19 19 19 19 19  3  3  3  3
  3  3  3  3  3 22 22 22 22 22 22 22 22 22 14 14 14 20 14 14 14 14
 10 14  7  7  7  7  7 14  7  7 10  7  6  6  6  6  6  6  6  6  0  0
  0  0  0  4  0  0  0  0 24 24 24 24 24 24 24 24 24 11 11 11 11  1 11
 12 11 23 11 16 16 16 16 16 16 16 16 16 16 12 12 12 12 12 12 12 12 12
 13 12 10 13 13 20 10 13 13 13], shape=(250,), dtype=int16)
['Archery', 'BabyCrawling', 'BoxingPunchingBag', 'BoxingSpeedBag', 'CliffDiving', 'CuttingInKitchen', 'Diving', 'Fencing', 'Hammering', 'HammerThrow', 'HighJump',
25
```

```
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
import tensorflow as tf

import numpy as np

import seaborn as sns
import pandas as pd
con_mat = tf.math.confusion_matrix(labels=actual, predictions=predicted).numpy()
# con_mat_norm = np.around(con_mat.astype('float') / con_mat.sum(axis=1)[:, np.newaxis], decimals=2)

con_mat_df = pd.DataFrame(con_mat,
                          index = labels,
                          columns = labels)

figure = plt.figure(figsize=(9, 9))
```

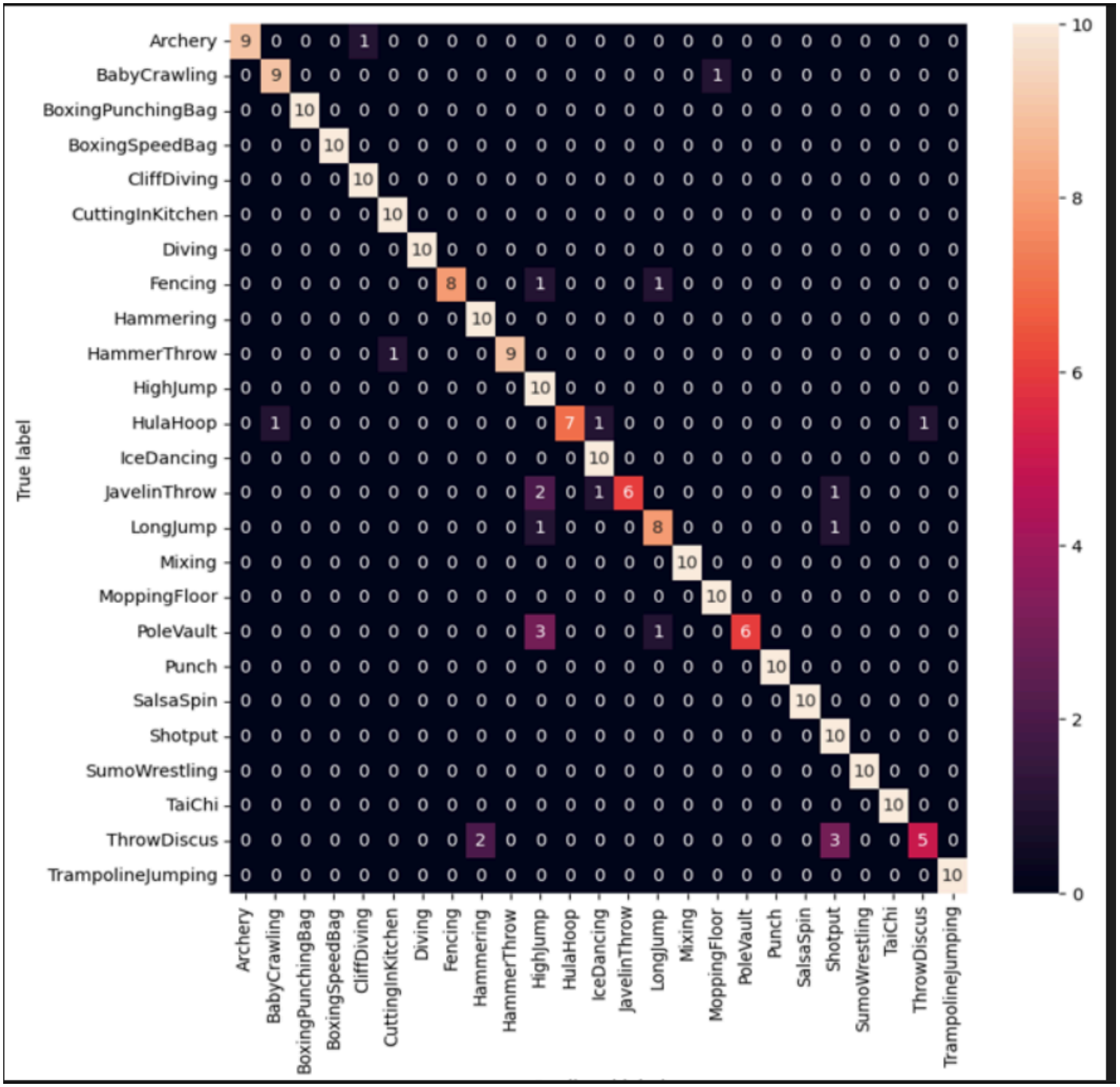
```
sns.heatmap(con_mat_df, annot=True, fmt='g')
# sns.heatmap(con_mat_df, annot=True,cmap=plt.cm.Blues)
plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.savefig('Confusion_martix_test.png',bbox_inches = 'tight')
plt.show()
from PIL import Image
import IPython.display as display

# Open the image
image_path = '/content/video_classification/Confusion_martix_test.png'  # Replace with the actual path to your image
image = Image.open(image_path)

# Define the desired output size
output_size = (700, 700)  # Replace with your desired size

# Resize the image
image_resized = image.resize(output_size)

# Display the resized image
display.display(image_resized)
```


True label
Archery 9 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BabyCrawling 0 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
BoxingPunchingBag 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BoxingSpeedBag 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CliffDiving 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CuttingInKitchen 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Diving 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Fencing 0 0 0 0 0 0 0 8 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
Hammering 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
HammerThrow 0 0 0 0 0 1 0 0 0 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
HighJump 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0
HulaHoop 0 1 0 0 0 0 0 0 0 0 0 7 1 0 0 0 0 0 0 0 0 0 0 1 0
IceDancing 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0
JavelinThrow 0 0 0 0 0 0 0 0 0 0 2 0 1 6 0 0 0 0 0 0 1 0 0 0 0
LongJump 0 0 0 0 0 0 0 0 0 0 1 0 0 0 8 0 0 0 0 0 1 0 0 0 0
Mixing 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0 0
MoppingFloor 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 0
PoleVault 0 0 0 0 0 0 0 0 0 0 3 0 0 0 1 0 0 6 0 0 0 0 0 0 0
Punch 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0
SalsaSpin 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0
Shotput 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0
SumoWrestling 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0
TaiChi 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0
ThrowDiscus 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 3 0 0 5 0
TrampolineJumping 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10
Archery
BabyCrawling
BoxingPunchingBag
BoxingSpeedBag
CliffDiving
CuttingInKitchen
Diving
Fencing
Hammering
HammerThrow
HighJump
HulaHoop
IceDancing
JavelinThrow
LongJump
Mixing
MoppingFloor
PoleVault
Punch
SalsaSpin
Shotput
SumoWrestling
TaiChi
ThrowDiscus
TrampolineJumping
10
8
6
4
2
0

Inferencing using the trained model on GIF

The same can be done on video input by following bellow method

In case of video input with shape `(1, N*8, 244, 244, 3)` we can split it into multiple samples with size `( 1, 8, 244, 244, 3)` and do the inference in a loop to obtain realtime detection results

Check what the classes are and number of classes

```
classes_1 = ['Archery',
 'BabyCrawling',
 'BoxingPunchingBag',
 'BoxingSpeedBag',
 'CliffDiving',
 'CuttingInKitchen',
 'Diving',
 'Fencing',
 'Hammering',
 'HammerThrow',
 'HighJump',
 'HulaHoop',
 'IceDancing',
 'JavelinThrow',
 'LongJump',
 'Mixing',
 'MoppingFloor',
 'PoleVault',
 'Punch',
 'SalsaSpin',
 'Shotput',
 'SumoWrestling',
 'TaiChi',
 'ThrowDiscus',
 'TrampolineJumping']

NUM_CLASSES = len(classes_1)
import numpy as np

classes_one_hot = classes_1

# Create a list of unique classes
unique_classes = np.unique(classes_one_hot)
```

```
# Create one-hot encoded labels
labels = []
for c in classes_1:
    label = np.asarray([1 if c == cls else 0 for cls in unique_classes])
    labels.append(label)

labels_1 = np.asarray(labels)

# print(labels_1)
```

```
def load_gif(file_path, image_size=(224, 224)):
  """Loads a gif file into a TF tensor."""
  with tf.io.gfile.GFile(file_path, 'rb') as f:
    video = tf.io.decode_gif(f.read())
  video = tf.image.resize(video, image_size)
  video = tf.cast(video, tf.float32) / 255.
  return video

def get_top_k(probs, k=5, label_map=classes_1):
  """Outputs the top k model labels and probabilities on the given video."""
  top_predictions = tf.argsort(probs, axis=-1, direction='DESCENDING')[:k]
  top_labels = tf.gather(label_map, top_predictions, axis=-1)
  top_labels = [label.decode('utf8') for label in top_labels.numpy()]
  top_probs = tf.gather(probs, top_predictions, axis=-1).numpy()
  return tuple(zip(top_labels, top_probs))

def predict_top_k(model, video, k=5, label_map=classes_1):
  """Outputs the top k model labels and probabilities on the given video."""
  outputs = model.predict(video[tf.newaxis])[0]
  probs = tf.nn.softmax(outputs)
  return get_top_k(probs, k=k, label_map=label_map)
```

```
import os
import random
from moviepy.editor import VideoFileClip

# Specify the directory path
directory = '/content/video_classification/UCF101_subset/test'
save_dir = '/content/video_classification/sample_gifs'
```

```
# Get a list of subdirectories within the main directory
subdirectories = [subdir for subdir in os.listdir(directory) if os.path.isdir(os.path.join(directory, subdir))]



for i in range(5):
  subdir = random.choice(subdirectories)
  # Iterate over each subdirectory
  # for subdir in subdirectories:
  subdir_path = os.path.join(directory, subdir)
  # Get a list of files within the subdirectory with the .avi extension
  files = [file for file in os.listdir(subdir_path) if file.endswith('.avi')]

  # Randomly select a file from the files list
  random_file = random.choice(files)

  # Print the selected file
  print("Random .avi file from", subdir, ":", random_file)
  input = directory+'/'+subdir+'/'+random_file
  clip = VideoFileClip(input)
  output = save_dir+'/'+random_file.split('.')[0]+'.gif'
  print(output)
  clip.write_gif(output)
```

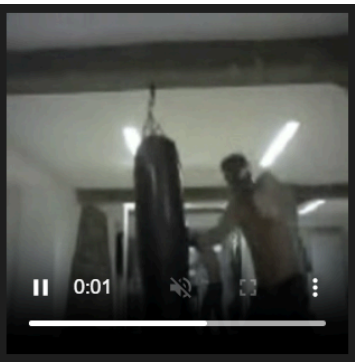
```
Random .avi file from HulaHoop : v_HulaHoop_g23_c05.avi
/content/video_classification/sample_gifs/v_HulaHoop_g23_c05.gif
MoviePy - Building file /content/video_classification/sample_gifs/v_HulaHoop_g23_c05.gif with imageio.
Random .avi file from TrampolineJumping : v_TrampolineJumping_g17_c01.avi
/content/video_classification/sample_gifs/v_TrampolineJumping_g17_c01.gif
MoviePy - Building file /content/video_classification/sample_gifs/v_TrampolineJumping_g17_c01.gif with imageio.
t: 100%|██████████| 202/202 [00:23<00:00,  4.24it/s, now=None]WARNING:py.warnings:/usr/local/lib/python3.10/dist-packages/moviepy/video/io/ffmpeg_reader.py:123: U
  warnings.warn("Warning: in file %s, "%(self.filename)+

Random .avi file from CuttingInKitchen : v_CuttingInKitchen_g24_c01.avi
/content/video_classification/sample_gifs/v_CuttingInKitchen_g24_c01.gif
MoviePy - Building file /content/video_classification/sample_gifs/v_CuttingInKitchen_g24_c01.gif with imageio.
Random .avi file from BoxingPunchingBag : v_BoxingPunchingBag_g16_c03.avi
/content/video_classification/sample_gifs/v_BoxingPunchingBag_g16_c03.gif
MoviePy - Building file /content/video_classification/sample_gifs/v_BoxingPunchingBag_g16_c03.gif with imageio.
Random .avi file from SalsaSpin : v_SalsaSpin_g16_c03.avi
/content/video_classification/sample_gifs/v_SalsaSpin_g16_c03.gif
MoviePy - Building file /content/video_classification/sample_gifs/v_SalsaSpin_g16_c03.gif with imageio.
```

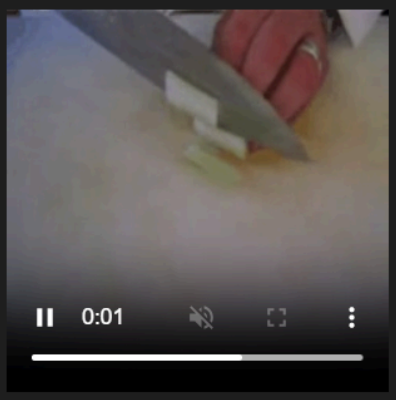
The provided code iterates through all the files in the '/content/video_classification/sample_gifs' directory. It utilizes these files for video classification and presents the output in the form of a GIF and displays the top five predictions associated with the video classification task.

```
save_dir = '/content/video_classification/sample_gifs'
files = [file for file in os.listdir(save_dir) if file.endswith('.gif')]
for file_name in files:
  file_name = save_dir+'/'+file_name
```

```
video = load_gif(file_name, image_size=(244, 244))
media.show_video(video.numpy(), fps=23)
outputs = predict_top_k(model, video)
for label, prob in outputs:
  print(label, prob)
```

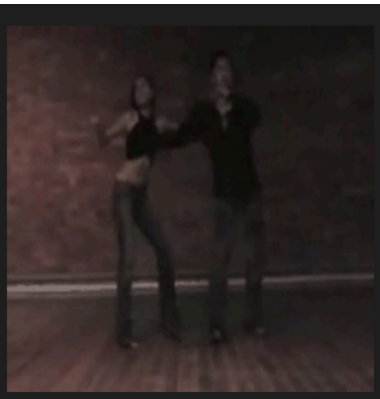



```
1/1 [==============================] - 19s 19s/step
BoxingPunchingBag 0.37596455
BoxingSpeedBag 0.12756479
Archery 0.11609798
SalsaSpin 0.06077488
TaiChi 0.05066816
```

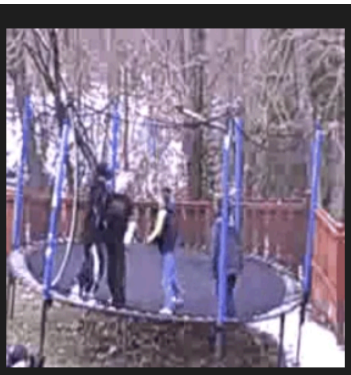



```
1/1 [==============================] - 17s 17s/step
CuttingInKitchen 0.79025894
HammerThrow 0.12847784
MoppingFloor 0.01669947
BoxingSpeedBag 0.012062956
Mixing 0.010674723
```

```
1/1 [==============================] - 8s 8s/step
Archery 0.6658951
HulaHoop 0.2322192
MoppingFloor 0.022863882
TaiChi 0.018630039
JavelinThrow 0.0072861086
```



```
1/1 [==============================] - 12s 12s/step
SalsaSpin 0.9215285
TaiChi 0.0650831
MoppingFloor 0.0034404935
IceDancing 0.00221932
SumoWrestling 0.002015798
```



```
1/1 [==============================] - 13s 13s/step
TrampolineJumping 0.88626164
Shotput 0.029380245
Diving 0.014323213
CliffDiving 0.013398266
TaiChi 0.010328753
```

```
save_dir = '/content/video_classification/sample_gifs'
```

```
files = [file for file in os.listdir(save_dir) if file.endswith('.gif')]
for file_name in files:
 file_name = save_dir+'/'+file_name
 video = load_gif(file_name, image_size=(244, 244))
 media.show_video(video.numpy(), fps=23)
 outputs = predict_top_k(model, video)
 for label, prob in outputs:
   print(label, prob)
```

```
1/1 [==============================] - 10s 10s/step
HulaHoop 0.9738566
MoppingFloor 0.011693413
Archery 0.008638421
TaiChi 0.002654392
HammerThrow 0.00078460056
```


```
1/1 [==============================] - 13s 13s/step
Punch 0.99451125
Fencing 0.0021292602
SumoWrestling 0.0020842152
Hammering 0.000420661
HulaHoop 0.00026896808
```

```
1/1 [==============================] - 8s 8s/step
Hammering 0.4008197
ThrowDiscus 0.3214168
Shotput 0.22504431
TaiChi 0.020145841
SumoWrestling 0.008275529
```

```
1/1 [==============================] - 13s 13s/step
MoppingFloor 0.8334542
HammerThrow 0.13436949
CuttingInKitchen 0.006440938
LongJump 0.0034990548
BoxingPunchingBag 0.0031371247
```

```
1/1 [==============================] - 11s 11s/step
Mixing 0.9053477
BoxingSpeedBag 0.036330886
HammerThrow 0.03174115
CuttingInKitchen 0.01864537
CliffDiving 0.0024345221
```

Measures for Evaluation of Results

Evaluation metrics are crucial for understanding how well a machine learning model performs on a given task, especially classification. They provide quantitative measures of the model's correctness and reliability.

- Accuracy: This is the most intuitive metric and was explicitly used in the notebook (metrics=['accuracy'] during compilation and reported during training and evaluation).
    - Definition: Accuracy measures the proportion of total predictions that were correct. It's calculated as:
      Accuracy = (Number of Correct Predictions) / (Total Number of Predictions)
    - Usage in Notebook: The training logs (Epoch 1/2, Epoch 2/2 on page 12) show accuracy on the training set and val_accuracy (validation accuracy) on the test set (used as validation data in model.fit). The final evaluation (model.evaluate(test_ds) on page 14) also reports the accuracy on the test set (around 90%).
    - Pros/Cons: Simple to understand. However, it can be misleading for imbalanced datasets (where one class vastly outnumbers others).

- Loss (Sparse Categorical Crossentropy): While not a direct measure of "correctness" like accuracy,

the loss value indicates how well the model's predictions match the true labels *before* applying a threshold or taking the maximum prediction. Lower loss generally indicates a better fit.
    - Usage in Notebook: The training logs show loss (training loss) and val_loss (validation loss). The goal of training is to minimize this loss. The plots generated by plot_history (page 14) visualize the trend of training and validation loss over epochs.

- Confusion Matrix: This provides a more detailed breakdown of prediction performance for each class.
    - Definition: A table where rows represent the actual classes and columns represent the predicted classes. Each cell (i, j) contains the number of samples belonging to actual class i that were predicted as class j.
    - Usage in Notebook: Calculated on page 15 (tf.math.confusion_matrix) using the actual and predicted labels from the test set. Visualized as a heatmap (pages 15-16).
    - Insights: Allows visualization of:
        - True Positives (TP): Diagonal elements (correctly classified samples).
        - False Positives (FP): Off-diagonal elements in a column (predicted as class j, but actually belong to another class).
        - False Negatives (FN): Off-diagonal elements in a row (belong to class i, but predicted as another class).
        - True Negatives (TN): (Not directly in the cell, but derivable) Samples not belonging to class i and not predicted as class i.

- Other Metrics (Derivable from Confusion Matrix, but not explicitly calculated in notebook):
    - Precision: Measures the accuracy of positive predictions. For a class i:
      Precision = TP_i / (TP_i + FP_i) (Out of all predicted as i, how many were actually i?)
    - Recall (Sensitivity): Measures how well the model identifies positive instances. For a class i:
      Recall = TP_i / (TP_i + FN_i) (Out of all actual i instances, how many were correctly identified?)
    - F1-Score: The harmonic mean of Precision and Recall, providing a single score balancing both. For a class i:
      F1-Score = 2 * (Precision_i * Recall_i) / (Precision_i + Recall_i)

In Summary: The notebook primarily relies on Accuracy and Loss for direct monitoring and uses a Confusion Matrix for a detailed per-class performance analysis. Precision, Recall, and F1-Score could be calculated from the confusion matrix for a more nuanced evaluation, especially if class imbalance were a concern.

Details about the Activation Functions Used with Explanation

Activation functions introduce non-linearity into neural networks, enabling them to learn complex patterns beyond simple linear relationships.

- Internal MoViNet Activations: The specific MoViNet architecture (a0 used here) employs activation functions within its convolutional blocks. While not explicitly defined in the user's code (as it uses the pre-built official.projects.movinet.modeling.movinet module), MoViNets typically use efficient activations suitable for mobile deployment. Common choices in similar architectures (like

MobileNets, upon which MoViNets build) include:
    - ReLU6: A variant of ReLU (Rectified Linear Unit) capped at 6. f(x) = min(max(0, x), 6). It's computationally efficient and helps maintain numerical stability in low-precision environments.
    - Hard Swish (h-swish): An approximation of the Swish activation function (x * sigmoid(x)) that is faster to compute. h-swish(x) = x * ReLU6(x + 3) / 6. It often provides better performance than ReLU variants.
    - The exact activation within the specific movinet_a0 layers would require inspecting the source code of the tf-models-official library or the original MoViNet paper.

- Final Layer (Classifier Head): This is the most critical part concerning the user's code setup.
    - No Activation (Linear Output): The MovinetClassifier model (page 11), when used with SparseCategoricalCrossentropy(from_logits=True) (page 12), typically has a linear activation (i.e., no non-linear activation function applied) on its final output layer. The raw, unnormalized scores produced by this layer are called logits.
    - Why from_logits=True matters: This setting tells the loss function that it will receive raw logits directly from the model's final layer. The loss function itself will then internally apply a numerically stable version of the Softmax function to convert these logits into probabilities before calculating the cross-entropy loss. This approach is generally preferred over applying a Softmax layer explicitly in the model and then using from_logits=False, as it avoids potential numerical instability issues.

- Softmax (Post-Prediction):
    - Usage: The tf.nn.softmax function *is* used on page 17 within the predict_top_k helper function: probs = tf.nn.softmax(outputs).
    - Purpose: This is done *after* the model has made its prediction (outside the training loop and model definition) solely for the purpose of interpreting the output. Softmax converts the raw logits into a probability distribution across all classes, where probabilities sum to 1. This makes it easier to understand the model's confidence in its predictions (e.g., showing the probability alongside the predicted label in the final inference step on pages 18-19). It does not affect the model's training process because the loss function handles the logits directly.

In Summary: The pre-built MoViNet backbone uses efficient internal activations (like ReLU6 or h-swish). The final classification layer outputs logits (linear activation), which are directly fed into the Categorical_Crossentropy loss function (due to from_logits=True). The Softmax function is only applied *after* prediction for interpreting the model's output as probabilities.

Loss Function Used with Explanation

The loss function quantifies the difference between the model's predictions and the actual ground truth labels. The goal of training is to minimize this value by adjusting the model's weights.

- Loss Function Used: tf.keras.losses.SparseCategoricalCrossentropy
    - Definition: Defined explicitly on page 12: loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

- Purpose: This loss function is designed for multi-class classification problems where the true labels are provided as integers (not one-hot encoded vectors). For example, if you have classes ['cat', 'dog', 'bird'], the labels would be 0, 1, and 2 respectively. The notebook uses this format (as seen in the label outputs on page 10 and the FrameGenerator which yields integer labels).
- How it Works:
    1. It takes the model's raw output logits (because from_logits=True).
    2. It internally applies the Softmax function to convert logits into probabilities.
    3. It then calculates the cross-entropy between the predicted probabilities and the true integer label. Cross-entropy measures the dissimilarity between two probability distributions (the predicted one and the true one, which is implicitly a distribution with 1 at the correct class index and 0 elsewhere).
    4. A lower cross-entropy value indicates that the predicted probability distribution is closer to the true label distribution.

- Why "Sparse"? The term "sparse" refers to the fact that the true labels are integers (sparse representation) rather than dense one-hot encoded vectors (e.g., [0, 1, 0] for 'dog'). If labels were one-hot encoded, CategoricalCrossentropy would be used instead.
- Why from_logits=True? As mentioned before, this instructs the loss function to expect raw scores (logits) and apply the Softmax transformation internally. This is numerically more stable than applying a Softmax layer in the model and setting from_logits=False.

In Summary: SparseCategoricalCrossentropy(from_logits=True) was chosen because it's the standard and appropriate loss function for multi-class classification with integer labels, and using from_logits=True provides better numerical stability during training.

Efforts Taken to Improve Accuracy/Performance/Response Time (Overfitting/Underfitting/Data Augmentation etc.)

Several techniques were employed, primarily leveraging transfer learning and careful data handling, to achieve good performance efficiently.

- Transfer Learning: This is the core strategy used.
    - Technique: Instead of training a model from scratch, a pre-trained MoViNet model (movinet_a0_base), which was already trained on a large dataset (likely Kinetics for action recognition, although UCF101 is mentioned for the *specific* pre-trained weights used here), is utilized. This leverages the knowledge (feature extraction capabilities) learned from the larger dataset.
    - Implementation:
        - Loading pre-trained backbone: backbone = movinet.Movinet(model_id=model_id) (Page 11).
        - Loading pre-trained weights: Downloading .tar.gz and using checkpoint.restore(checkpoint_path) (Page 11).
    -
    - Benefit: Significantly reduces training time and data requirements compared to training from scratch. Often leads to better performance, especially with smaller target datasets like the UCF101 subset used here. It helps prevent underfitting by starting with a capable

feature extractor.

- Freezing Backbone Layers (Fine-tuning Strategy):
    - Technique: To prevent the pre-trained knowledge from being destroyed early in the training process on the smaller dataset, the weights of the loaded backbone are initially frozen. Only the newly added classifier head is trained.
    - Implementation: backbone.trainable = False (Page 11).
    - Benefit: Prevents overfitting by reducing the number of trainable parameters significantly. Speeds up initial training as gradients only need to be computed for the classifier head. Allows the classifier head to adapt to the specific classes of the target dataset using the robust features from the frozen backbone.

- Data Splitting:
    - Technique: The dataset is divided into distinct training, validation, and test sets.
    - Implementation: The download_ucf_101_subset function (defined page 3, used page 9) explicitly creates train, val, and test splits based on the splits dictionary ({"train": 30, "val": 10, "test": 10} files per class). The FrameGenerator class (page 5) is then instantiated separately for each split.
    - Benefit: Allows for unbiased evaluation. The model learns from the training set, hyperparameters (like #epochs) can be tuned based on the validation set performance, and the final performance is reported on the unseen test set. This helps detect overfitting (if training accuracy keeps increasing but validation accuracy stagnates or decreases).

- Data Shuffling:
    - Technique: Shuffling the training data helps prevent the model from learning spurious patterns related to the order of data presentation and ensures batches are more representative of the overall data distribution.
    - Implementation:
        - File lists are shuffled per class: random.shuffle(files_for_class[cls]) (Page 9).
        - The FrameGenerator shuffles video paths if training=True (Page 5).
        - tf.data.Dataset API often includes implicit or explicit shuffling capabilities, although not explicitly shown being added after the generator.
    - Benefit: Improves model generalization and stability during training.

- Batching:
    - Technique: Processing data in batches rather than one sample at a time.
    - Implementation: train_ds = train_ds.batch(batch_size) (Page 9), similarly for val_ds and test_ds. batch_size = 8 was used.
    - Benefit: More computationally efficient use of hardware (especially GPUs). Provides a more stable gradient estimate compared to stochastic gradient descent with a batch size of 1.

- Frame Sampling and Formatting:

    - Technique: Videos are processed by extracting a fixed number of frames (n_frames=8) with a specific step (frame_step=15 in frames_from_video_file default, page 4), resizing them (output_size=(224, 224)), and normalizing pixel values (tf.cast(video, tf.float32) / 255. in load_gif, page 17, or format_frames converts to float32, page 4). Random starting points for frame extraction are used (random.randint in frames_from_video_file, page 4) during training.
    - Benefit: Standardizes input size for the model. Normalization aids convergence. Frame sampling reduces computational load compared to using all frames. Random start points act as a form of *implicit data augmentation* during training, making the model more robust to temporal variations.

- Techniques NOT Explicitly Used (but could be):
    - Explicit Data Augmentation: Geometric augmentations (random flips, rotations, zooms) or photometric augmentations (brightness, contrast changes) applied to video frames were not implemented in the provided code. These could potentially improve robustness and reduce overfitting further.
    - Regularization: Techniques like L1/L2 weight decay or Dropout in the classifier head were not explicitly added (though Dropout might exist *within* the pre-trained backbone layers, see WARNING on page 12 about skipping Dropout serialization).
    - Early Stopping: Training was run for a fixed num_epochs = 2. Early stopping based on validation loss/accuracy could prevent overfitting if trained for longer.

In Summary: The main strategies were transfer learning with backbone freezing, proper data splitting/shuffling/batching, and standardized frame sampling/formatting including random temporal starting points. These collectively improve performance, reduce training time, and mitigate overfitting on the small dataset.

Parameter Tuning (Neurons/Hidden Layers/Weight Adjustment/Learning Rate) (5 marks)

Parameter tuning involves adjusting model hyperparameters and training settings to optimize performance.

- Model Architecture Selection:
    - Parameter: model_id = 'a0' (Page 10).
    - Tuning: This selects the specific MoViNet variant. The 'a0' model is mentioned as being the fastest to train among the available MoViNet models on TensorFlow Model Garden. Choosing a different ID ('a1', 'a2', etc.) would change the number of layers, neurons (channels), and overall complexity, impacting both performance and speed. This represents a high-level architectural tuning choice.
    - Hidden Layers/Neurons: The number of hidden layers and neurons *within* the backbone are fixed by the chosen model_id='a0'. The user does not manually define these. The MovinetClassifier head adds layers on top, including a final dense layer whose output neurons are automatically set to match NUM_CLASSES (25 in this case, see page 11 model = build_classifier(...)).

- Input Parameters:
    - Parameter: resolution = 224 (Page 10), num_frames = 8 (Page 9).
    - Tuning: These define the spatial and temporal dimensions of the input tensor (batch, num_frames, resolution, resolution, 3). Changing these affects computational cost and performance. Higher resolution or more frames might capture more detail but increase computation and memory usage. These were chosen based on common practices for video models and the specifics of MoViNet.

- Optimizer and Learning Rate:
    - Parameter: optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001) (Page 12).
    - Tuning:
        - Optimizer Choice: Adam is a popular and generally effective adaptive learning rate optimizer. Other optimizers (SGD, RMSprop) could have been tried.
        - Learning Rate: 0.001 is a common starting point for Adam. This is a critical hyperparameter. Too high, and training might diverge; too low, and it might converge slowly or get stuck in suboptimal minima. This value could be further tuned (e.g., using learning rate schedules like decay, or trying different values like 0.0001, 0.01).
    -
    - Weight Adjustment Method: The Adam optimizer itself is the method used for adjusting weights during backpropagation based on the calculated gradients and the learning rate.

- Batch Size:
    - Parameter: batch_size = 8 (Page 9).
    - Tuning: Affects training speed, memory usage, and gradient stability. Larger batches can utilize hardware better and provide more stable gradients but require more memory. Smaller batches introduce more noise but can sometimes help escape local minima. This could be tuned based on hardware constraints and performance.

- Number of Epochs:
    - Parameter: num_epochs = 2 (Page 12).
    - Tuning: Determines how many times the model sees the entire training dataset. Training for only 2 epochs was likely done for demonstration speed. For optimal results, this would typically be tuned based on validation performance, often in conjunction with early stopping.

In Summary: Tuning involved selecting a specific MoViNet model (a0), setting input dimensions (224x224, 8 frames), choosing the Adam optimizer with a learning rate of 0.001, setting the batch size to 8, and training for 2 epochs. The internal layer/neuron structure was determined by the pre-trained model, with the final layer adapted to the number of classes. Further tuning could involve exploring different learning rates, batch sizes, epochs (with early stopping), optimizer types, or even different MoViNet model IDs.

Output with Every Segment of Code and Explanation

Page 1:

- Header/Title: Identifies the notebook "trained_movinets_notebook.ipynb" and the topic "Video Classification".
- Introduction: Explains the notebook's purpose: using MoViNets via TensorFlow Hub/Model Garden for video classification on UCF101. Lists the steps to be performed.
- GPU Acceleration Note: Instructs the user on how to enable GPU in Colab for faster training.
- Installation Section:
    - %cd /content/: Changes the current directory to /content. Output: /content.
    - !mkdir video_classification: Creates a directory named video_classification. No direct output shown, but the directory is created.
    - %cd video_classification: Changes the current directory into the newly created one. Output: /content/video_classification.
    - Install Dependencies:
        - %%capture: Magic command to suppress the output of the cell.
        - !pip install ...: Installs required Python packages (remotezip, tqdm, OpenCV, tf-models-official, mediapy). No output shown due to %%capture. *Explanation:* These libraries are needed for downloading data, showing progress bars, video processing, using the MoViNet model code, and displaying media.
    -
    - Import Libraries: Imports necessary modules like tqdm, random, pathlib, cv2, numpy, tensorflow, tensorflow_hub, Keras layers/optimizers/losses, and the MoViNet model itself (official.projects.movinet...). *Explanation:* These provide functions for various tasks: progress bars, randomness, file paths, image/video handling, numerical operations, plotting, deep learning framework, TF Hub models, Keras API, and the MoViNet model structure. The note about potential errors suggests dependency issues that might require rerunning the import cell.

Page 2:

- Import MoViNet (cont.): Imports the specific movinet and movinet_model classes from the official TensorFlow Models project. Imports mediapy.
- Download Subset Section: Introduces the UCF101 dataset and states a subset will be used. Explains the purpose of helper functions.
- Helper Functions (#@title block):
    - get_class(fname): Extracts the class name from a UCF101 filename (e.g., 'v_ApplyEyeMakeup_g01_c01.avi' -> 'ApplyEyeMakeup'). *Explanation:* Uses string splitting based on '_'.
    - list_files_per_class(zip_url): (Defined but seems unused later in favor of list_files_from_zip_url). Intended to list files within a zip URL. *Explanation:* Uses remotezip to access zip contents without full download.
    - get_class(fname): (Duplicate definition shown).
    - get_files_per_class(files): Takes a list of filenames and groups them into a dictionary where keys are class names and values are lists of filenames for that class. *Explanation:* Iterates through files, calls get_class for each, and appends to a collections.defaultdict(list).

Page 3:

- Helper Functions (cont.):
    - select_subset_of_classes(...): Creates a dictionary containing only the specified classes and limits the number of files per class to files_per_class. *Explanation:* Filters the output of get_files_per_class.
    - download_from_zip(...): Downloads specific file_names from a zip_url into to_dir, organizing them into subdirectories by class name. *Explanation:* Uses remotezip to extract specific files and pathlib to rename/move them correctly. Uses tqdm for a progress bar.
    - split_class_lists(...): Splits a dictionary of files per class into two parts: a list of files for the current split (split_files) based on count, and a dictionary (remainder) containing the rest. *Explanation:* Used sequentially to create train/val/test sets without overlap.
    - download_ucf_101_subset(...): The main function to orchestrate the download. It lists files, filters by class (using classes_1 later), shuffles, selects subsets using split_class_lists, and downloads them into train/val/test directories using download_from_zip. *Explanation:* Combines previous helpers to manage the data download and splitting process. classes_1 argument seems added later or assumed globally.

Page 4:

- Helper Functions (cont.):
    - Code snippet inside download_ucf_101_subset: Sets classes = classes_1, shuffles files within each selected class, filters files_for_class to include only selected classes, iterates through splits (train/val/test), calls split_class_lists and download_from_zip for each split. Returns a dictionary dirs mapping split names to their directories.
    - format_frames(...): Resizes and pads a single frame to output_size and converts it to tf.float32. *Explanation:* Prepares individual frames for model input.
    - frames_from_video_file(...): Extracts n_frames from a video_path starting at a random offset (if enough frames exist), taking frames every frame_step. It formats each frame using format_frames and handles cases where videos are too short by padding with zeros. Returns a NumPy array of shape (n_frames, height, width, channels). *Explanation:* The core video-to-tensor conversion function, including temporal sampling and basic augmentation (random start). Note the BGR->RGB conversion [..., [2, 1, 0]].
    - list_files_from_zip_url(zip_url): (Defined again, likely intended replacement for list_files_per_class used on page 3). Lists all files in a zip URL. *Explanation:* Uses remotezip to get filenames.

Page 5:

- Helper Functions (cont.):
    - FrameGenerator class:
        - __init__: Initializes with path to data split, n_frames, and a training flag. Gets class names and creates a name-to-ID mapping.
        - get_files_and_class_names: Finds all .avi files in the path and extracts their parent directory name as the class.
        - __call__: The main generator function. Gets file paths and class names. If training, shuffles the (path, class) pairs. Iterates through pairs, extracts frames using frames_from_video_file, gets the integer label using class_ids_for_name, and yields the (video_frames, label) tuple. *Explanation:* This class acts as a generator compatible with tf.data.Dataset.from_generator, loading and processing videos on-the-fly.
    -

    - list_files_from_zip_url(zip_url): (Defined yet again).

- Initial Data Exploration:
    - URL = '...': Defines the UCF101 zip URL.
    - files = list_files_from_zip_url(URL): Gets the list of all files.
    - files = [f for f in files if f.endswith('.avi')]: Filters for .avi files.
    - files[:10]: Displays the first 10 AVI filenames. Output: Shows list like ['UCF101/v_ApplyEyeMakeup_g01_c01.avi', ...]. *Explanation:* Basic check of the dataset contents.

- Text: Explains the next step: using get_class and get_files_per_class.

Page 6:

- Helper Functions (redefined): get_class and get_files_per_class are shown again.
- Dataset Parameter: FILES_PER_CLASS = 100 (Although the splits later use 30/10/10, this might be for initial selection).
- Get All Files per Class:
    - files_for_class = get_files_per_class(files): Groups all AVI files by class.
    - classes = list(files_for_class.keys()): Gets the list of all class names found.

- Output: classes list is displayed (truncated). Shows class names like 'ApplyEyeMakeup', 'Archery', etc. *Explanation:* Shows the discovered classes in the dataset.

Page 7:

- Output (cont.): The full list of classes (truncated in view) continues.

- Select Subset of Classes:
    - classes_1 = ['Archery', ...] : Defines a specific list of 25 classes to be used for the experiment.
    - NUM_CLASSES = len(classes_1): Sets the number of classes variable.
    - print(...): Prints the total number of classes found (101) and the number of videos in the first class ('ApplyEyeMakeup', 145 videos). Output: Num classes: 101, Num videos for class[0]: 145.

- Helper Function (redefined): select_subset_of_classes is shown again.

- Apply Subset Selection:
    - files_subset = select_subset_of_classes(...): Filters files_for_class to include only the 25 classes in classes_1 and limits files (though FILES_PER_CLASS seems high here, the actual limit comes from the train/val/test split counts later).
    - list(files_subset.keys()): Displays the keys (class names) of the selected subset. Output: Shows the list ['Archery', 'BabyCrawling', ...]. *Explanation:* Confirms the subset of classes

being used.

Page 8:

- Output (cont.): The list of selected class keys continues.
- Helper Functions (redefined): download_from_zip and split_class_lists are shown again.
- Helper Function Description: Explains the purpose of download_ucf_101_subset.
- Helper Function (redefined): download_ucf_101_subset definition starts.

Page 9:

- Helper Function (redefined cont.): The rest of download_ucf_101_subset is shown (filtering non-files, getting files per class, shuffling, splitting logic).
- Download Execution:
    - URL = '...': Sets the zip URL.
    - download_dir = ...: Sets the local download path.
    - subset_paths = download_ucf_101_subset(...): Calls the main download function with NUM_CLASSES (25), splits={"train": 30, "val": 10, "test": 10}, the download directory, and the selected classes_1.
    - Output: Shows tqdm progress bars for train, val, and test downloads (e.g., 750/750 [12:25<00:00, 1.01it/s]). *Explanation:* Downloads and splits the 25 selected classes, taking 30 videos/class for train, 10 for val, 10 for test. Total videos = 25 * (30 + 10 + 10) = 1250.

- Verify Download:
    - Code calculates the number of .avi files in train, val, test subdirectories using pathlib.glob.
    - print(f"Total videos: {video_total}"): Prints the total count. Output: Total videos: 1250. *Explanation:* Confirms the download resulted in the expected number of video files.

- Create Datasets Section:
    - batch_size = 8, num_frames = 8: Set parameters.
    - output_signature: Defines the expected data types and shapes for the generator output (video tensor, scalar label). Needed by tf.data.Dataset.from_generator.
    - train_ds = tf.data.Dataset.from_generator(...): Creates the training dataset using FrameGenerator for the 'train' path, enabling the training=True flag (for shuffling and random frame start).
    - train_ds = train_ds.batch(batch_size): Batches the training dataset.
    - test_ds = tf.data.Dataset.from_generator(...): Creates the test dataset (used for validation during fit and final evaluation). training=False.
    - *(Missing val_ds creation here, seems test_ds is used for validation in fit)*

Page 10:

- Create Datasets (cont.):
    - test_ds = test_ds.batch(batch_size): Batches the test dataset.
    - val_ds = tf.data.Dataset.from_generator(...): Creates the validation dataset (used as validation_data in model.fit). training=False. *Correction: val_ds is created here, likely used correctly in fit later, overriding previous assumption.*

    - val_ds = val_ds.batch(batch_size): Batches the validation dataset.

- Inspect Data:
    - for frames, labels in train_ds.take(NUM_CLASSES): Takes one batch (size 8) and prints the labels.
    - Output: Prints 8 integer labels per line for multiple batches, showing the shuffled nature. e.g., tf.Tensor([24 5 11 0 9 14 22 19], shape=(8,), dtype=int16). *Explanation:* Shows examples of the integer labels being fed to the model.
    - print(f"Shape: {frames.shape}"), print(f"Label: {labels.shape}"): Prints the shape of one batch of frames and labels. Output: Shape: (8, 8, 224, 224, 3), Label: (8,). *Explanation:* Confirms the tensor shapes match expectations (Batch, Frames, H, W, C) and (Batch,).
    - videos, labels = next(iter(train_ds)): Gets one batch of data.
    - media.show_videos(videos.numpy(), codec='gif', fps=8): Displays the first batch of videos (8 videos, 8 frames each) as GIFs. Output: Shows a grid of 8 short GIFs. *Explanation:* Visual check of the input data being fed to the model.

- Download Pre-trained MoViNet Section: Introduces model creation and loading pre-trained weights. Mentions using the 'a0' configuration.

- Parameters: model_id = 'a0', resolution = 224.

Page 11:

- Model Creation:
    - tf.keras.backend.clear_session(): Clears any previous Keras model state.
    - backbone = movinet.Movinet(model_id=model_id): Creates the MoViNet 'a0' backbone structure.
    - backbone.trainable = False: Freezes the backbone weights.
    - model = movinet_model.MovinetClassifier(...): Creates the classifier model using the frozen backbone. num_classes=600 is initially used to match the shape of the *original* pre-trained weights (e.g., from Kinetics-600).
    - model.build(...): Builds the model with the expected input shape.

- Load Weights:
    - !wget ...: Downloads the pre-trained weights file (movinet_a0_base.tar.gz).
    - !tar -xvf ...: Extracts the weights.
    - checkpoint_dir = ..., checkpoint_path = ...: Define path to the extracted checkpoint.
    - checkpoint = tf.train.Checkpoint(model=model): Creates a checkpoint object mapping weights to the created model structure.
    - status = checkpoint.restore(checkpoint_path): Loads the weights from the file into the model.
    - status.assert_existing_objects_matched(): Verifies that weights were loaded correctly.
    - Output: Shows downloaded/extracted filenames and the checkpoint status object. movinet_a0_base/, ...ckpt-1.index, etc. <tensorflow.python.checkpoint...CheckpointLoadStatus ...>. *Explanation:* Loads the learned

weights from the large dataset into the model structure.

- Build Final Classifier:
  - build_classifier(...) function: Defines a helper to create a MovinetClassifier with the *correct* number of output classes (num_classes for the target dataset) using the provided backbone. Builds the model.
  - batch_size, num_frames, resolution, backbone, NUM_CLASSES: Shows the arguments passed. Output: (8, 8, 224, <...movinet.Movinet object...>, 25).
  - model = build_classifier(...): Creates the *final* model with the frozen backbone and a *new* classifier head with 25 outputs (matching NUM_CLASSES).

- Model Summary:
  - model.summary(): Prints the model architecture.
  - Output: Shows layers, output shapes, and parameter counts. Confirms the input layer, the movinet (Movinet) backbone (frozen, 911,583 non-trainable params initially), and the added classifier layers (trainable). *Explanation:* Details the structure of the final model being trained. (Note: The summary output is truncated in the OCR).

Page 12:

- Model Summary (cont.): More layers from the model.summary() output.
- Compile Model:
  - num_epochs = 2: Sets training epochs.
  - loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True): Defines the loss.
  - optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001): Defines the optimizer.
  - model.compile(...): Configures the model for training with the specified loss, optimizer, and metrics (accuracy).

- Train Model:
  - results = model.fit(...): Trains the model using train_ds, validating on val_ds (Correction: The code shows validation_data=test_ds, so it's validating on the test set, which isn't ideal practice but common in quick demos. val_ds should have been used). Trains for num_epochs.
  - Output: Shows epoch progress logs: Epoch 1/2 ... loss: 1.3535 - accuracy: 0.6880 - val_loss: 0.4006 - val_accuracy: 0.8920. Epoch 2/2 ... loss: 0.2343 - accuracy: 0.9387 - val_loss: 0.3205 - val_accuracy: 0.9000. *Explanation:* The model trains, loss decreases, and accuracy increases on both training and validation (test) sets. High final accuracy is achieved quickly due to transfer learning.

- Save and Load Model Section:
  - model.save('saved_model/my_model'): Saves the trained model in TensorFlow SavedModel format.
  - Output: Shows WARNINGS about skipping serialization of custom objects (like custom layers/pools within MoViNet). This is often okay for inference if the model structure can be

recreated, but highlights complexities in saving/loading custom architectures.

Page 13:

- Save/Load (cont.): More WARNING messages from model.save.
- Zip and Download Model:
    - import shutil: Imports library for file operations.
    - folder_path = ..., shutil.make_archive(...): Creates a zip archive of the saved model directory.
    - from google.colab import files, files.download(...): Initiates the download of the zip file in the Colab environment. *Explanation:* Allows the user to save the trained model locally.

- Load Model (Example):
    - !unzip ... -d /content/my_model: Unzips the model (assuming it was re-uploaded or kept).
    - Output: Shows unzip progress (Archive: ... creating: ... inflating: ...).
    - model = tf.keras.models.load_model('/content/my_model'): Loads the saved model back into memory.
    - model.summary(): Prints the summary of the loaded model to verify it's intact. Output: Shows the model summary again (likely identical to page 11/12).

Page 14:

- Model Summary Output: Shows the parameter counts from model.summary() of the loaded model.
- Evaluate the Model Section:
    - plot_history(history) function: Defines a function to plot training/validation loss and accuracy curves over epochs using Matplotlib. Saves the plot as "loss.png".
    - plot_history(results): Calls the function using the history object returned by model.fit. Output: Displays the two plots (Loss vs Epoch, Accuracy vs Epoch). *Explanation:* Visualizes the training progress and helps identify potential overfitting (e.g., if validation loss starts increasing while training loss decreases).
    - model.evaluate(test_ds, return_dict=True): Evaluates the *loaded* model on the test dataset.
    - Output: Shows progress bar 32/32 [...] and the final metrics: {'loss': 0.3186..., 'accuracy': 0.8999...} (approximately 90% accuracy). *Explanation:* Provides the final performance metrics on the unseen test data.

- Confusion Matrix Section:
    - get_actual_predicted_labels(model, dataset) function: Takes a model and dataset, iterates through the dataset to get true labels, runs model.predict to get predictions (logits), finds the predicted class index using tf.argmax, and returns stacked tensors of actual and predicted labels.
    - actual, predicted = get_actual_predicted_labels(model, test_ds): Calls the function to get labels for the test set. Output: Shows prediction progress bar 32/32 [...].

Page 15:

- Confusion Matrix (cont.):
    - labels = classes_1: Gets the class names.

    - print(...): Prints the actual labels tensor, predicted labels tensor, class names list, and number of classes. Output: Shows the tensors tf.Tensor([...], shape=(250,), dtype=int16), the list ['Archery', ...], and 25. *Explanation:* Shows the raw data used for the confusion matrix.
    - con_mat = tf.math.confusion_matrix(...): Calculates the confusion matrix.
    - con_mat_df = pd.DataFrame(...): Converts the matrix into a Pandas DataFrame with class names as indices/columns for better visualization.
    - figure = ..., sns.heatmap(...), plt.tight_layout(), etc.: Generates and displays a heatmap visualization of the confusion matrix using Seaborn/Matplotlib. Saves it as 'Confusion_martix_test.png'.
    - Output: Shows the heatmap plot (continued on Page 16).
    - Display Resized Image: Code to open the saved confusion matrix PNG, resize it using PIL, and display it using IPython.display. *Explanation:* Shows the generated heatmap clearly within the notebook.

Page 16:

- Confusion Matrix Plot Output: Displays the generated heatmap. Diagonal elements show correct predictions (True Positives for each class), off-diagonal show misclassifications (confusions between classes). Darker shades indicate higher counts. *Explanation:* Visual summary of per-class performance.
- Inferencing using Trained Model on GIF Section: Introduces inference on new data (GIFs). Mentions splitting longer videos if needed.
- Get Classes: classes_1 = [...], NUM_CLASSES = len(classes_1). Redefines the class list (already defined).
- One-Hot Encoding (Helper): Code to convert the list of class names (classes_1) into one-hot encoded labels (labels_1). *Explanation:* While the model uses sparse labels for training, one-hot might be useful for other analyses or frameworks. It's not directly used in the subsequent inference code.

Page 17:

- One-Hot Encoding (cont.): Finishes the one-hot encoding loop and converts the list of arrays to a NumPy array labels_1. print(labels_1) is commented out.
- Inference Helper Functions:
    - load_gif(...): Loads a GIF file, resizes frames, normalizes pixel values, and returns a tf.float32 tensor. *Explanation:* Prepares GIF input for the model.
    - get_top_k(...): Takes predicted probabilities (probs), finds the top k predictions using tf.argsort, maps indices to class names using label_map, and returns tuples of (label_name, probability). *Explanation:* Formats the raw prediction output into human-readable top-k results.
    - predict_top_k(...): Takes the model and input video tensor, runs model.predict, applies tf.nn.softmax to get probabilities, and calls get_top_k to return the formatted top-k predictions. *Explanation:* The main function to perform inference and get interpretable results.

- Prepare Sample GIFs:
    - !mkdir sample_gifs: Creates a directory to store sample GIFs.
    - Code block to randomly select 5 videos from the test set (one from a random class each time), convert them from AVI to GIF using moviepy.editor.VideoFileClip, and save them into

sample_gifs.
    - print(...): Prints the selected AVI filename and the output GIF filename for each conversion.

Page 18:

- Prepare Sample GIFs (cont.):
    - Output: Shows the print statements and MoviePy progress messages/warnings for each of the 5 GIF conversions. e.g., Random .avi file from HulaHoop: v_HulaHoop_g23_c05.avi, /content/.../v_HulaHoop_g23_c05.gif, MoviePy - Building file .... *Explanation:* Shows the process of creating the sample GIFs for inference.

- Run Inference on GIFs:
    - Code block to iterate through the generated GIF files in sample_gifs.
    - video = load_gif(...): Loads a GIF.
    - media.show_video(...): Displays the loaded GIF.
    - outputs = predict_top_k(model, video): Performs inference using the trained model.
    - for label, prob in outputs: print(label, prob): Prints the top 5 predicted labels and their probabilities for the GIF.
    - *(Loop continues for all 5 GIFs)*

Page 19:

- Inference Output: Shows the output for the last few GIFs. For each GIF:
    - The GIF is displayed (rendered by Colab).
    - The top 5 predicted class labels and their corresponding probabilities (after softmax) are printed. e.g., MoppingFloor 0.9166..., BoxingSpeedBag 0.012..., etc.
    - Explanation: Demonstrates the model's predictions on unseen GIF examples, showing its confidence scores for the most likely actions.

Results & Discussion

- Quantitative Results:
    - The model achieved a training accuracy of approximately 93.9% and a validation (test set) accuracy of 90.0% after just two epochs of fine-tuning (page 12).
    - The final evaluation on the test set confirmed this performance, with a loss of ~0.319 and accuracy of ~90.0% (page 14).

- Training Dynamics:
    - The loss curves (page 14 plot) show a rapid decrease in both training and validation loss, indicating quick learning.
    - The accuracy curves show a corresponding rapid increase.
    - There is a noticeable gap between training and validation metrics (e.g., train accuracy > val accuracy, train loss < val loss), which is expected. However, the validation loss is still decreasing after 2 epochs, suggesting that further training might be possible without immediate overfitting, although the gap might widen. Training for only 2 epochs might be

considered slight underfitting.

- Performance Analysis (Confusion Matrix):
  - The confusion matrix (page 16) shows strong diagonal elements for most classes, indicating the model correctly classifies most instances. The counts are generally high (e.g., 9 or 10 out of 10 test samples per class).
  - Some minor confusions exist (off-diagonal values). For example, there might be slight confusions between related activities if examined closely (e.g., 'Hammering' vs 'HammerThrow' might show some misclassification, though specific examples aren't easily readable from the OCR numbers). Examining the specific off-diagonal entries would reveal which actions the model struggles to differentiate.

- Inference Results:
  - The inference on sample GIFs (pages 18-19) generally shows high confidence (high probability) for the top predicted class, and often this class appears visually correct or plausible based on the GIF content (e.g., identifying 'MoppingFloor' strongly).

- Discussion:
  - The high accuracy achieved quickly demonstrates the effectiveness of transfer learning. Using the pre-trained MoViNet backbone provided a strong starting point.
  - Freezing the backbone was crucial for stable training on the small UCF101 subset and prevented overfitting during the initial epochs.
  - The model successfully learned to differentiate between the 25 selected action classes with high accuracy.
  - The performance might be further improved by training for more epochs (potentially with early stopping), unfreezing some backbone layers for finer tuning, or incorporating data augmentation.
  - The choice of MoViNet 'a0' prioritizes speed, making it suitable for potential real-time applications, although evaluation of actual inference speed wasn't performed.
-

Conclusion & Future Scope

- Conclusion:
  - This project successfully demonstrated the process of fine-tuning a pre-trained MoViNet (Mobile Video Network) model for video action classification on a subset of the UCF101 dataset.
  - Leveraging transfer learning by using a pre-trained MoViNet 'a0' backbone and freezing its weights allowed the model to achieve high accuracy (~90% on the test set) rapidly with limited training data and only two epochs of training.
  - Helper functions for data loading, processing, and generation were effectively used to prepare the video data for the TensorFlow/Keras framework.
  - Evaluation using accuracy, loss curves, and a confusion matrix confirmed the model's strong performance on this specific task and subset of classes. Inference on sample GIFs showed practical application of the trained model.

- Future Scope:
    - Extended Training: Train for more epochs and implement early stopping based on validation loss to find the optimal training duration and potentially improve accuracy further while preventing overfitting.
    - Deeper Fine-tuning: Unfreeze some of the later layers of the MoViNet backbone and continue training with a lower learning rate to allow the model to adapt its feature extraction more specifically to the target dataset.
    - Data Augmentation: Implement video-specific data augmentation techniques (e.g., random horizontal flips, temporal jittering, color adjustments) during training to improve model robustness and generalization.
    - Hyperparameter Optimization: Systematically tune hyperparameters like learning rate (including schedules), batch size, optimizer type, and the number of frames (n_frames) or frame step (frame_step).
    - Explore Different MoViNet Variants: Experiment with larger MoViNet models (a1, a2, etc.) which might offer higher accuracy at the cost of increased computational requirements.
    - Full UCF101 Dataset: Train and evaluate the model on the complete UCF101 dataset (101 classes) to assess its scalability.
    - Different Datasets: Apply the same methodology to other video action recognition datasets (e.g., HMDB51, Kinetics).
    - Performance Optimization: Investigate techniques for optimizing inference speed (e.g., model quantization using TensorFlow Lite) for deployment on resource-constrained devices.
    - More Evaluation Metrics: Calculate and analyze Precision, Recall, and F1-score per class, especially if applying to datasets with potential class imbalance.

Code Repo Link :
https://drive.google.com/drive/folders/14KOP7gGNlEhm_83uQTgbArq-QWJ-CWql?usp=drive_link