Video Noise Detection by Using Deep Learning

What is video?

So let's begin so what is video so video is sequence of frames arranged in a specific order

Load video data

This task demonstrates how to load and preprocess AVI video data using the UCF101 human action dataset. Once you have preprocessed the data, it can be used for such tasks as video classification/recognition, captioning or clustering. The original dataset contains realistic action videos collected from YouTube with 101 categories, including playing cello, brushing teeth, and applying eye makeup.

You will learn how to:

- Load the data from a zip file.
- Read sequences of frames out of the video files.
- Visualize the video data.
- Wrap the frame-generator tf.data.Dataset.

Setup

Begin by installing and importing some necessary libraries, including:

- remotezip to inspect the contents of a ZIP file,
- tqdm to use a progress bar,

- OpenCV to process video files, and
- tensorflow_docs for embedding data in a Jupyter notebook.

```
$ # The way this tutorial uses the `TimeDistributed` layer requires TF>=2.10
$ pip install -U "tensorflow>=2.10.0"
```

```
$ pip install remotezip tqdm opencv-python
$ pip install -q git+https://github.com/tensorflow/docs
```

```
import tqdm
import random
import pathlib
import itertools
import collections

import os
import cv2
import numpy as np
import remotezip as rz

import tensorflow as tf

# Some modules to display an animation using imageio.
import imageio
from IPython import display
from urllib import request
from tensorflow_docs.vis import embed
```

Download a subset of the UCF101 dataset

The UCF101 dataset contains 101 categories of different actions in video, primarily used in action recognition. You will use a subset of these categories in this demo.

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

The above 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:

```
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

```
NUM_CLASSES = 10
FILES_PER_CLASS = 50
```

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

```
print('Num classes:', len(classes))
print('Num videos for class[0]:', len(files_for_class[classes[0]]))
```

```
Num classes: 101
Num videos for class[0]: 145
```

Create a new function called `select_subset_of_classes` that selects a subset of the classes present within the dataset and a particular number of files per 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
```

```
files_subset = select_subset_of_classes(files_for_class, classes[:NUM_CLASSES], FILES_PER_CLASS)
list(files_subset.keys())
```

```
['ApplyEyeMakeup',
 'ApplyLipstick',
 'Archery',
 'BabyCrawling',
 'BalanceBeam',
 'BandMarching',
 'BaseballPitch',
 'BasketballDunk',
 'Basketball',
 'BenchPress']
```

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 subdirectories.

```
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)
```

The following function returns the remaining data that hasn't already been placed into a subset of data. It allows you to place that remaining data in the next specified subset of data.

```
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.

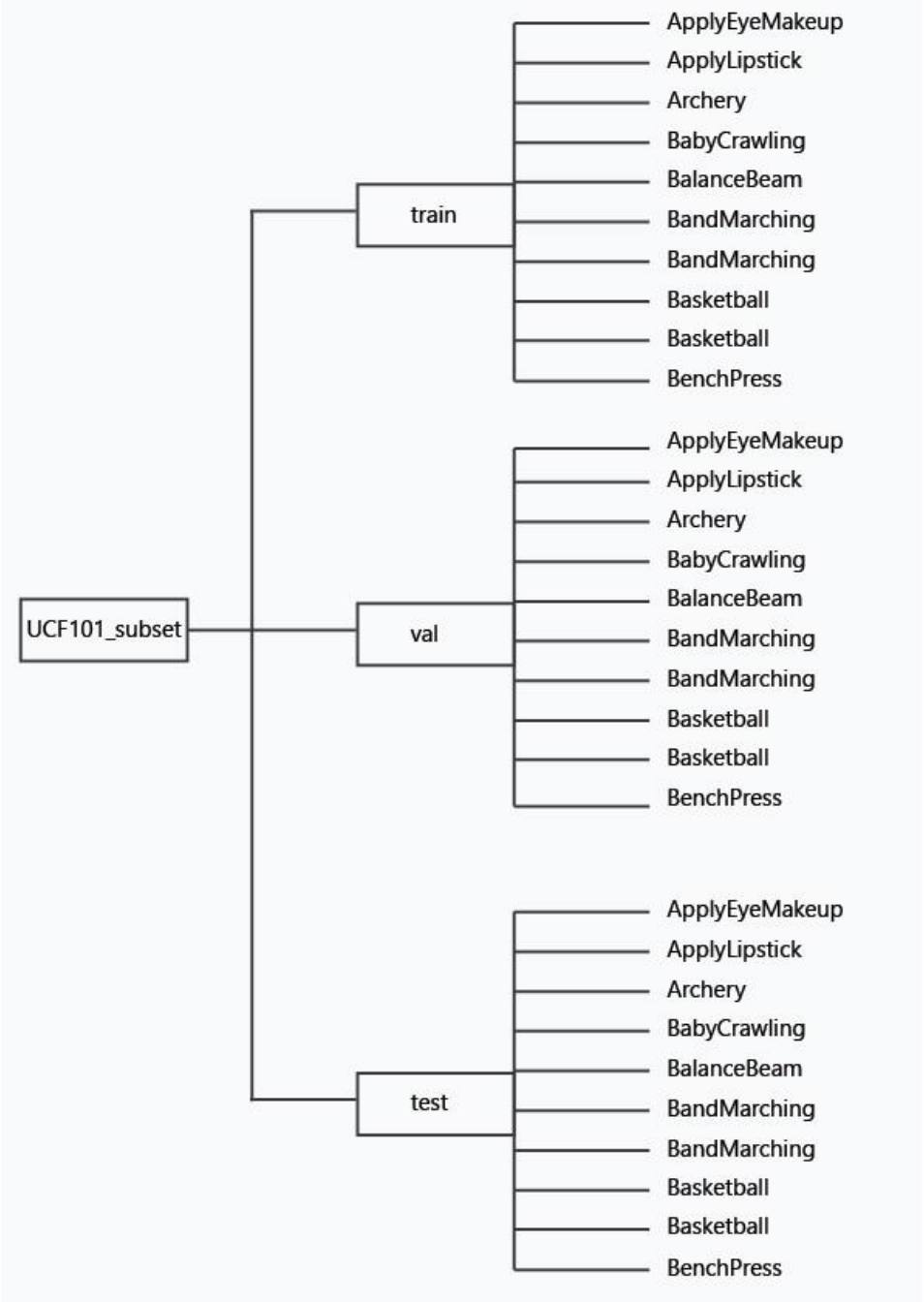
```
def download_ucf_101_subset(zip_url, num_classes, splits, download_dir):
  """ Download a subset of the UCF101 dataset and split them into various parts, such as
    training, validation, and test.

    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]

  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
```

```
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)
```

```
train :
100%|| 300/300 [09:27<00:00,  1.89s/it]
val :
100%|| 100/100 [03:06<00:00,  1.87s/it]
test :
100%|| 100/100 [03:04<00:00,  1.84s/it]
```

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}")
```

```
Total videos: 500
```

You can also preview the directory of data files now.

Create frames from each video file

The `frames_from_video_file` function splits the videos into frames, reads a randomly chosen span of `n_frames` out of a video file, and returns them as a NumPy `array`. To reduce memory and computation overhead, choose a **small** number of frames. In addition, pick the **same** number of frames from each video, which makes it easier to work on batches of data.

```
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
```

Visualize video data

The `frames_from_video_file` function that returns a set of frames as a NumPy array.

```
def to_gif(images):
  converted_images = np.clip(images * 255, 0, 255).astype(np.uint8)
  imageio.mimsave('./animation.gif', converted_images, fps=10)
  return embed.embed_file('./animation.gif')
```

In addition to examining this video, you can also display the UCF-101 data. To do this, run the following code:

```
# docs-infra: no-execute
ucf_sample_video = frames_from_video_file(next(subset_paths['train'].glob('*/*.avi')), 50)
to_gif(ucf_sample_video)
```

Next, define the `FrameGenerator` class in order to create an iterable object that can feed data into the TensorFlow data pipeline. The generator (`__call__`) function yields the frame array produced by `frames_from_video_file` and a one-hot encoded vector of the label associated with the set of frames.

```
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
```

Test out the `FrameGenerator` object before wrapping it as a TensorFlow Dataset object. Moreover, for the training dataset, ensure you enable training mode so that the data will be shuffled.

```
fg = FrameGenerator(subset_paths['train'], 10, training=True)

frames, label = next(fg())

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

```
Shape: (10, 224, 224, 3)
Label: 3
```

Finally, create a TensorFlow data input pipeline. This pipeline that you create from the generator object allows you to feed in data to your deep learning model. In this video pipeline, each element is a single set of frames and its associated label.

```
# Create the training set
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'], 10, training=True),
                                          output_signature = output_signature)
```

Check to see that the labels are shuffled.

```
for frames, labels in train_ds.take(10):
  print(labels)
```

```
tf.Tensor(7, shape=(), dtype=int16)
tf.Tensor(3, shape=(), dtype=int16)
tf.Tensor(0, shape=(), dtype=int16)
tf.Tensor(5, shape=(), dtype=int16)
tf.Tensor(7, shape=(), dtype=int16)
tf.Tensor(2, shape=(), dtype=int16)
tf.Tensor(9, shape=(), dtype=int16)
tf.Tensor(2, shape=(), dtype=int16)
tf.Tensor(8, shape=(), dtype=int16)
tf.Tensor(6, shape=(), dtype=int16)
```

```
# Create the validation set
val_ds = tf.data.Dataset.from_generator(FrameGenerator(subset_paths['val'], 10),
                                        output_signature = output_signature)
```

```
# Print the shapes of the data
train_frames, train_labels = next(iter(train_ds))
print(f'Shape of training set of frames: {train_frames.shape}')
print(f'Shape of training labels: {train_labels.shape}')

val_frames, val_labels = next(iter(val_ds))
print(f'Shape of validation set of frames: {val_frames.shape}')
print(f'Shape of validation labels: {val_labels.shape}')
```

```
Shape of training set of frames: (10, 224, 224, 3)
Shape of training labels: ()
Shape of validation set of frames: (10, 224, 224, 3)
Shape of validation labels: ()
```

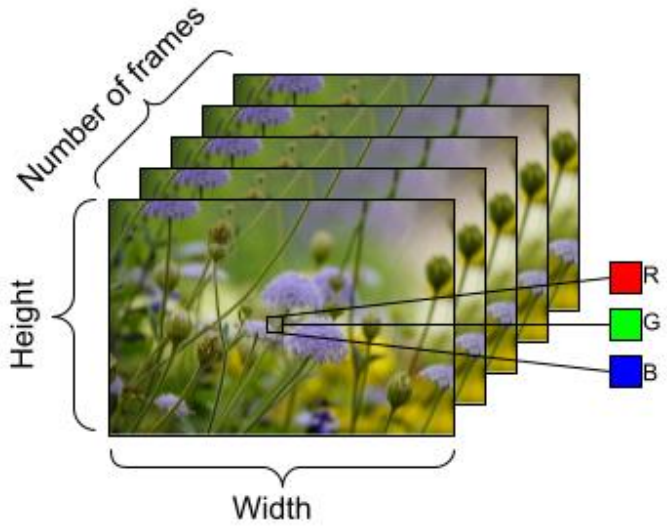
Configure the dataset for performance

Use buffered prefetching such that you can yield data from the disk without having I/O become blocking. Two important functions to use while loading data are:

- Dataset.cache: keeps the sets of frames in memory after they're loaded off the disk during the first epoch. This function ensures that the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.
- Dataset.prefetch: overlaps data preprocessing and model execution while training. Refer to Better performance with the tf.data for details.

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size = AUTOTUNE)
val_ds = val_ds.cache().shuffle(1000).prefetch(buffer_size = AUTOTUNE)
```

To prepare the data to be fed into the model, use batching as shown below. Notice that when working with video data, such as AVI files, the data should be shaped as a five dimensional object. These dimensions are as follows: [`batch_size, number_of_frames, height, width, channels`]. In comparison, an image would have four dimensions: `[batch_size, height, width, channels]`. The image below is an illustration of how the shape of video data is represented.




```
train_ds = train_ds.batch(2)
val_ds = val_ds.batch(2)

train_frames, train_labels = next(iter(train_ds))
print(f'Shape of training set of frames: {train_frames.shape}')
print(f'Shape of training labels: {train_labels.shape}')

val_frames, val_labels = next(iter(val_ds))
print(f'Shape of validation set of frames: {val_frames.shape}')
print(f'Shape of validation labels: {val_labels.shape}')
```

```
Shape of training set of frames: (2, 10, 224, 224, 3)
Shape of training labels: (2,)
Shape of validation set of frames: (2, 10, 224, 224, 3)
Shape of validation labels: (2,)
```

Next steps

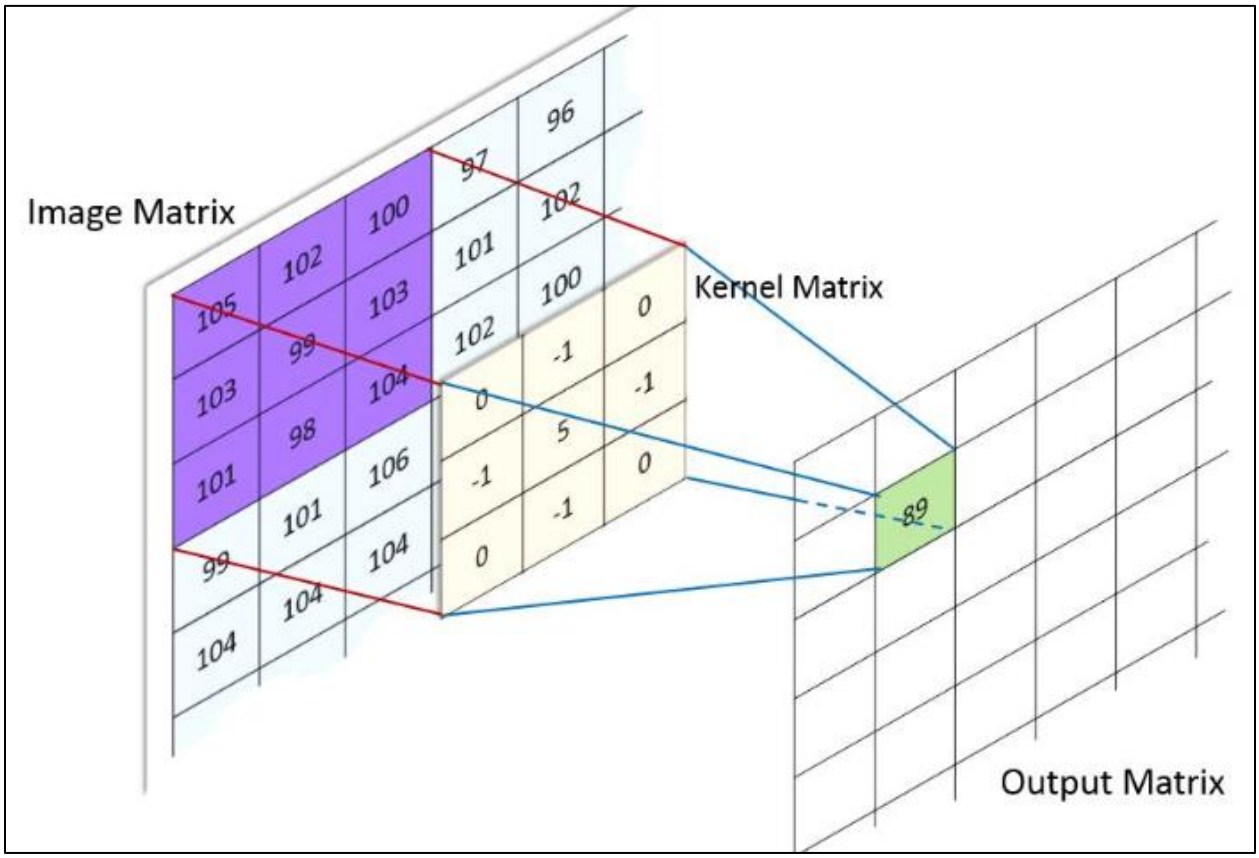
Now that you have created a TensorFlow `Dataset` of video frames with their labels, you can use it with a deep learning model. The following classification model that uses a pre-trained EfficientNet trains to high accuracy in a few minutes:

```
net = tf.keras.applications.EfficientNetB0(include_top = False)
net.trainable = False

model = tf.keras.Sequential([
    tf.keras.layers.Rescaling(scale=255),
    tf.keras.layers.TimeDistributed(net),
    tf.keras.layers.Dense(10),
    tf.keras.layers.GlobalAveragePooling3D()
])

model.compile(optimizer = 'adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True),
              metrics=['accuracy'])

model.fit(train_ds,
          epochs = 10,
          validation_data = val_ds,
          callbacks = tf.keras.callbacks.EarlyStopping(patience = 2, monitor = 'val_loss'))
```

Today, I'll be showing you how to use Keras and TensorFlow to create video classification model using the Conv3D layer. We went over how to load and process video data.

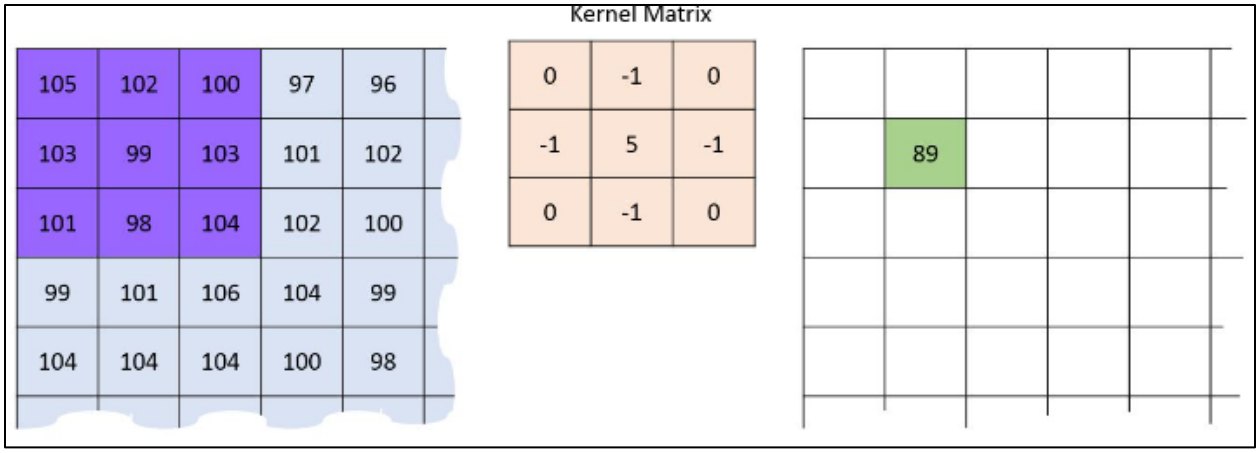
Here I'll outline differences between 2D and 3D. CNN, how to use the Conv3D layer, model architecture, and potential next steps.

Model architecture



First, let's cover what a 2D convolution neural network is. In 2D convolution, use a two-dimensional filter to form convolution operation-otherwise known as element-wise operations-performed on part of the matrix.

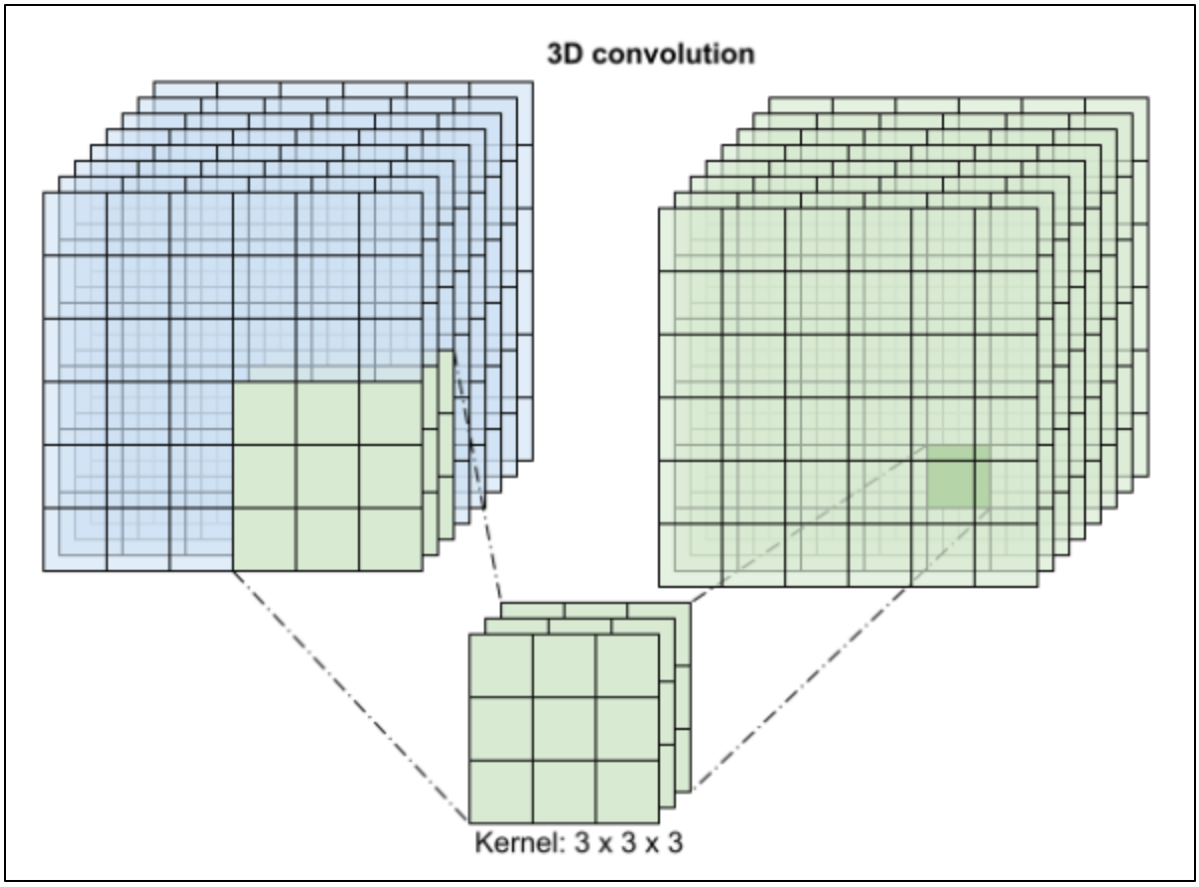
This filter or kernel is matrix of same rank as the input and is multiply by the input in a convolution layer. Filters will learn to do things like edge detection or feature detection by looking at shapes and textures. The Conv2D layer works well for image processing. And weights are learn via gradient descent. When you train the model. That being said, you may not always want to be working with images. When you work with video data, for example using the Conv3D layer is a suitable option.



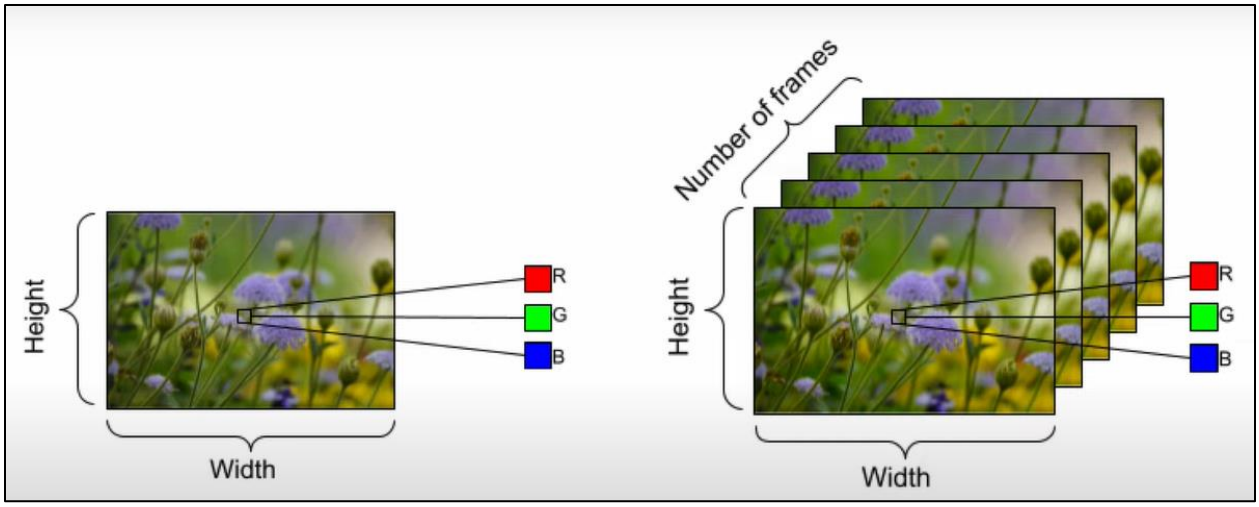
$$105 * 0 + 102 * -1 + 100 * 0$$
$$+103 * -1 + 99 * 5 + 103 * -1$$
$$+101 * 0 + 98 * -1 + 104 * 0 = 89$$

kernel has a specific task to do and the sharpen kernel accentuate edges but with the cost of adding noise to those area of the image which colors are changing gradually. The output of image convolution is calculated as follows:

1. Flip the kernel both horizontally and vertically. As our selected kernel is symmetric, the flipped kernel is equal to the original.
2. Put the first element of the kernel at every pixel of the image (element of the image matrix). Then each element of the kernel will stand on top of an element of the image matrix.
3. Multiply each element of the kernel with its corresponding element of the image matrix (the one which is overlapped with it)
4. Sum up all product outputs and put the result at the same position in the output matrix as the center of kernel in image matrix.



Let' discuss what a3D Convolution neural network and what you can use them for. A 3D CNN uses a three-dimensional filter to perform convolutions. The kernel can slide in three directions, whereas 2D CNN, it can slid in two-dimensions. This means you can use 3D CNNs to process video data, which you can think of as a cube of frames you can also use 3D CNNs to process volumetric data, like you might find in an MRI. 2D convolutions perform operations over space. So you can think of a 3D convolution as a volumetric convolution that goes over space and time or some kind of depth. While 3D CNNs uses more parameters than 2D CNNs, they're able to better handle data in which the time or depth matters. In Keras API, the name for the 3D CNN layer is Conv3D.

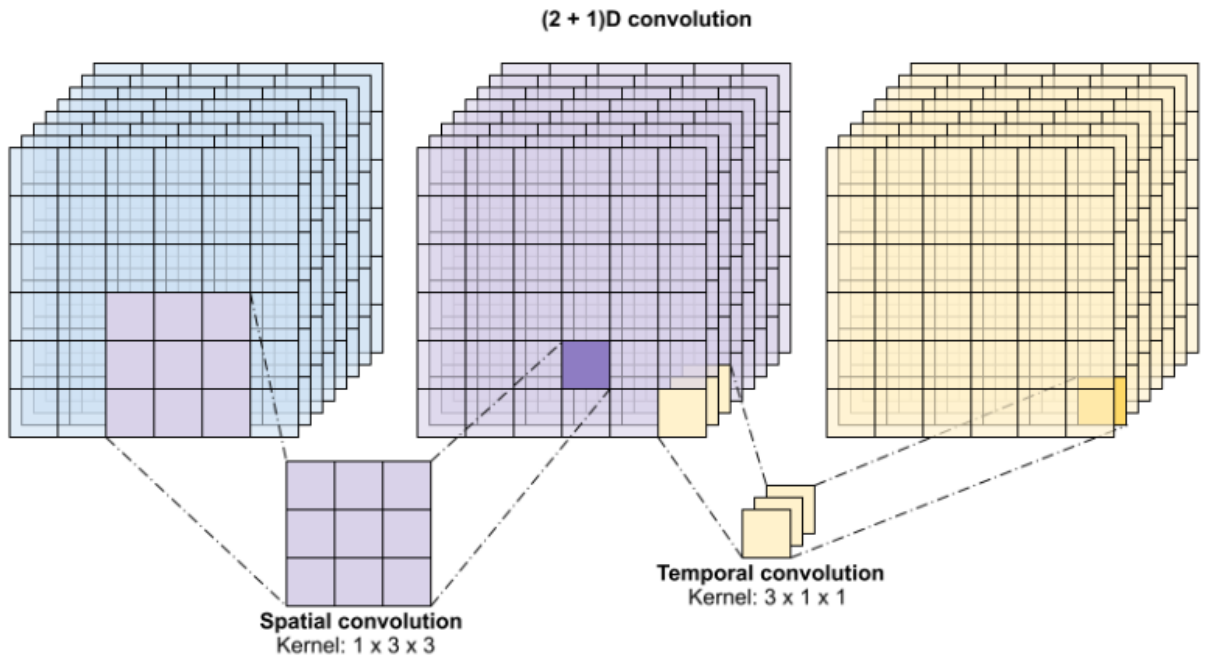


It's important to note that two-dimension data, such as an image, also has 3D filter within it, namely for color space as the RGB planes. Three dimensional data is different because there is a series of images in a video, otherwise known as frames. The 3D aspect as 3D data comes from fact that this type of data has spatial and temporal components outside the number of channels present.

Conv3D Layer

```
tf.keras.layers.Conv3D(
    filters,
    kernel_size,
    strides=(1, 1, 1),
    padding='valid'
)
```

So what does a 3D convolution layer look like in code? Let's take a look at the API and perhaps four arguments you can pass in. The filters parameter represents the dimensionality of the output or the number of output nodes. The kernel size argument refers to the size of kernel. This argument can be tuple of three integers that represent the depth or time, height, and width. It can be a single integer if all of these values are the same. By default, strides will be set to a tuple of 1 by1 by 1. This specifies the strides the convolution takes along each dimension. A padding value of "valid" means that there is no padding. If you change this argument to "same" this means that there will be a padding with zeros evenly to left and right and up and down of the input, such that the output has the same height and width dimension as the input. There are more arguments that can be passed in. And if you curious to see how you can further customize a Conv3d layer, please visit the TensorFlow documentation on the Conv3D layer.



The example I'll use here is a twist on regular 3D CNNs.

I wanted to provide you with an example of using a custom layer use to increase the efficiency of your network by reducing the number of parameters in your model. We'll use a 2 plus 1 D CNN. In a 2 plus 1 D CNN, We start with a spatial convolution followed by a temporal convolution. A spatial convolution is similar to a 2D convolution shown before, while a temporal convolution is a convolution over one dimension. As the names imply, a spatial convolution will perform the convolution operation over the height and width dimensions, while a temporal convolution will perform the convolution over the time domain. While a 3D convolution performs the convolution over space and time together, hence it being volumetric, we just separating here. A 3D convolution layer with a kernel size of 3by 3 by 3 would require a weight matrix with 27 times channel squared entries. When we split a 3D convolution into spatial and temporal aspect, we reduce the amount of parameters used. Instead of having a 3by 3 by 3 filter, we have a 3 by 3 by 1plus 1 by 1 by 3 filter, which results in a 9 times channel squared plus 3 times channel squared entries.

Mathematical comparison

```
3D convolution:
Kernel size of (3 x 3 x 3) requires weight-matrix with 27 *  channels² entries

(2+1)D convolution:
Kernel size of (3 x 3 x 3) requires weight-matrix with (9 * channels² ) + (3 * channels²) entries
```

Now, let me show you how to implement a 2 plus 1 D CNN by creating a custom Keras layer class. There is a tutorial in the TensorFlow documentation on how to make new layers and models via sub-classing, which you can find in the description. We create Keras sequential object and then add two Conv3D layers to it. For the kernel size arguments, the dimensional order of the tuple is time, height, and width. For the spatial part, set the time part of the tuple to 1. For the temporal part, set the height and width parts of the tuple to 1.

```
class Conv2Plus1D(keras.layers.Layer):
  def __init__(self, filters, kernel_size, padding):
    """
      A sequence of convolutional layers that first apply the convolution operation over the
      spatial dimensions, and then the temporal dimension.
    """
    super().__init__()
    self.seq = keras.Sequential([
        # Spatial decomposition
        layers.Conv3D(filters=filters,
                      kernel_size=(1, kernel_size[1], kernel_size[2]),
                      padding=padding),
        # Temporal decomposition
        layers.Conv3D(filters=filters,
                      kernel_size=(kernel_size[0], 1, 1),
                      padding=padding)
        ])

  def call(self, x):
    return self.seq(x)
```

A ResNet model is made from a sequence of residual blocks. A residual block has two branches. The main branch performs the calculation, but is difficult for gradients to flow through. The residual branch bypasses the main calculation and mostly just adds the input to the output of the main branch. Gradients flow easily through this branch. Therefore, an easy path from the loss function to any of the residual block's main branch will be present. This avoids the vanishing gradient problem.

Create the main branch of the residual block with the following class. In contrast to the standard ResNet structure this uses the custom Conv2Plus1D layer instead of layers.Conv2D.

```
class ResidualMain(keras.layers.Layer):
  """
    Residual block of the model with convolution, layer normalization, and the
    activation function, ReLU.
  """
  def __init__(self, filters, kernel_size):
    super().__init__()
    self.seq = keras.Sequential([
        Conv2Plus1D(filters=filters,
                    kernel_size=kernel_size,
                    padding='same'),
        layers.LayerNormalization(),
        layers.ReLU(),
        Conv2Plus1D(filters=filters,
                    kernel_size=kernel_size,
                    padding='same'),
        layers.LayerNormalization()
    ])

  def call(self, x):
    return self.seq(x)
```

To add the residual branch to the main branch it needs to have the same size. The Project layer below deals with cases where the number of channels is changed on the branch. In particular, a sequence of densely-connected layer followed by normalization is added.

```
class Project(keras.layers.Layer):
  """
    Project certain dimensions of the tensor as the data is passed through different
    sized filters and downsampled.
  """
  def __init__(self, units):
    super().__init__()
    self.seq = keras.Sequential([
        layers.Dense(units),
        layers.LayerNormalization()
    ])

  def call(self, x):
    return self.seq(x)
```

Use `add_residual_block` to introduce a skip connection between the layers of the model.

```
def add_residual_block(input, filters, kernel_size):
  """
    Add residual blocks to the model. If the last dimensions of the input data
    and filter size does not match, project it such that last dimension matches.
  """
  out = ResidualMain(filters,
                     kernel_size)(input)

  res = input
  # Using the Keras functional APIs, project the last dimension of the tensor to
  # match the new filter size
  if out.shape[-1] != input.shape[-1]:
    res = Project(out.shape[-1])(res)

  return layers.add([res, out])
```

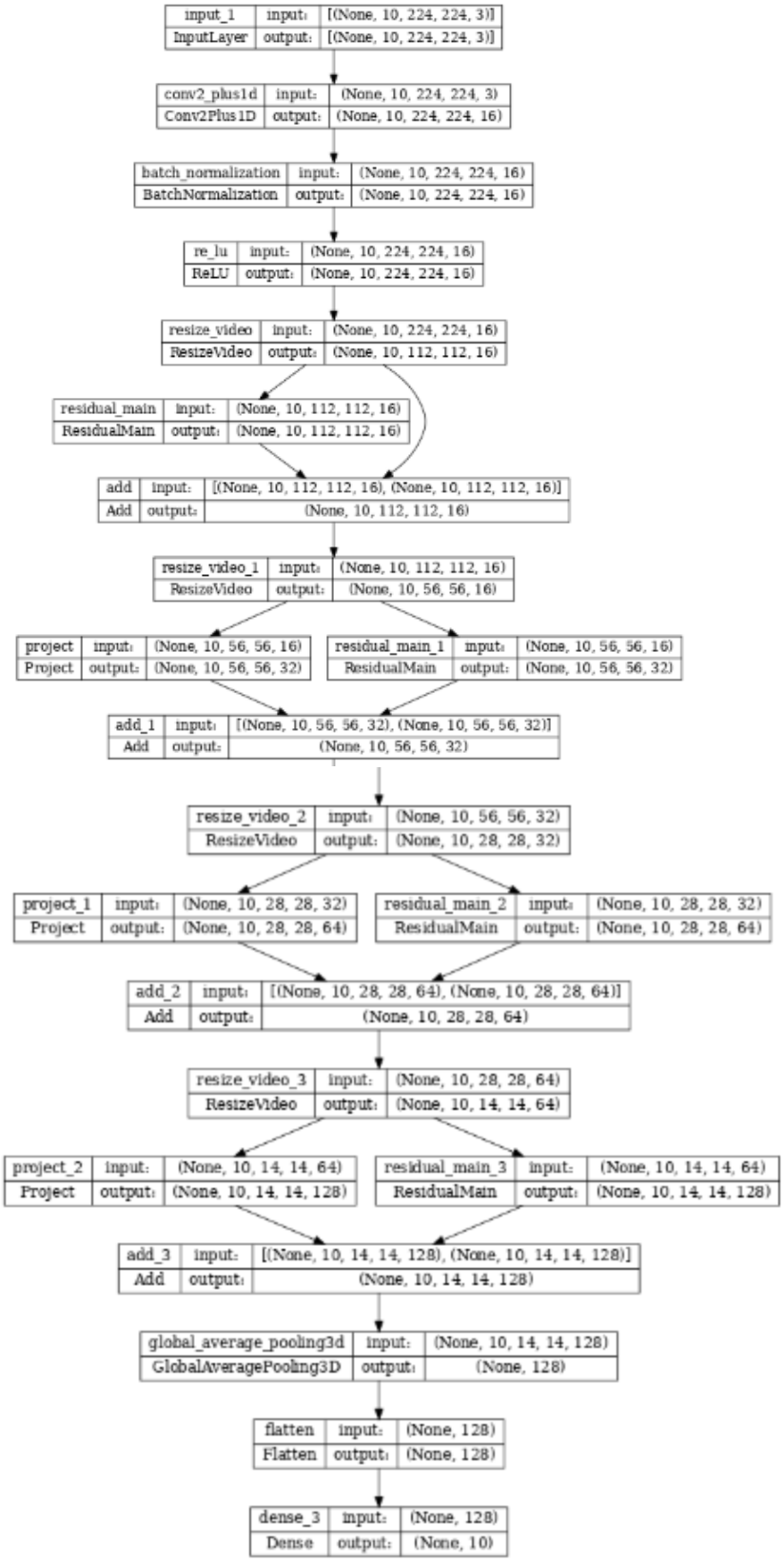
Resizing the video is necessary to perform downsampling of the data. In particular, downsampling the video frames allow for the model to examine specific parts of frames to detect patterns that may be specific to a certain action. Through downsampling, non-essential information can be discarded. Moreoever, resizing the video will allow for dimensionality reduction and therefore faster processing through the model.

```
class ResizeVideo(keras.layers.Layer):
  def __init__(self, height, width):
    super().__init__()
    self.height = height
    self.width = width
    self.resizing_layer = layers.Resizing(self.height, self.width)

  def call(self, video):
    """
      Use the einops library to resize the tensor.

      Args:
        video: Tensor representation of the video, in the form of a set of frames.

      Return:
        A downsampled size of the video according to the new height and width it should be resized to.
    """
    # b stands for batch size, t stands for time, h stands for height,
    # w stands for width, and c stands for the number of channels.
    old_shape = einops.parse_shape(video, 'b t h w c')
    images = einops.rearrange(video, 'b t h w c -> (b t) h w c')
    images = self.resizing_layer(images)
    videos = einops.rearrange(
        images, '(b t) h w c -> b t h w c',
        t = old_shape['t'])
    return videos
```

Use the Keras functional API to build the residual network.

```
input_shape = (None, 10, HEIGHT, WIDTH, 3)
input = layers.Input(shape=(input_shape[1:]))
x = input

x = Conv2Plus1D(filters=16, kernel_size=(3, 7, 7), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = ResizeVideo(HEIGHT // 2, WIDTH // 2)(x)

# Block 1
x = add_residual_block(x, 16, (3, 3, 3))
x = ResizeVideo(HEIGHT // 4, WIDTH // 4)(x)

# Block 2
x = add_residual_block(x, 32, (3, 3, 3))
x = ResizeVideo(HEIGHT // 8, WIDTH // 8)(x)

# Block 3
x = add_residual_block(x, 64, (3, 3, 3))
x = ResizeVideo(HEIGHT // 16, WIDTH // 16)(x)

# Block 4
x = add_residual_block(x, 128, (3, 3, 3))

x = layers.GlobalAveragePooling3D()(x)
x = layers.Flatten()(x)
x = layers.Dense(10)(x)

model = keras.Model(input, x)
```

```
frames, label = next(iter(train_ds))
model.build(frames)
```


input_1 | input: | [(None, 10, 224, 224, 3)]
InputLayer | output: | [(None, 10, 224, 224, 3)]
conv2_plus1d | input: | (None, 10, 224, 224, 3)
Conv2Plus1D | output: | (None, 10, 224, 224, 16)
batch_normalization | input: | (None, 10, 224, 224, 16)
BatchNormalization | output: | (None, 10, 224, 224, 16)
re_lu | input: | (None, 10, 224, 224, 16)
ReLU | output: | (None, 10, 224, 224, 16)
resize_video | input: | (None, 10, 224, 224, 16)
ResizeVideo | output: | (None, 10, 112, 112, 16)
residual_main | input: | (None, 10, 112, 112, 16)
ResidualMain | output: | (None, 10, 112, 112, 16)
add | input: | [(None, 10, 112, 112, 16), (None, 10, 112, 112, 16)]
Add | output: | (None, 10, 112, 112, 16)
resize_video_1 | input: | (None, 10, 112, 112, 16)
ResizeVideo | output: | (None, 10, 56, 56, 16)
project | input: | (None, 10, 56, 56, 16)
Project | output: | (None, 10, 56, 56, 32)
residual_main_1 | input: | (None, 10, 56, 56, 16)
ResidualMain | output: | (None, 10, 56, 56, 32)
add_1 | input: | [(None, 10, 56, 56, 32), (None, 10, 56, 56, 32)]
Add | output: | (None, 10, 56, 56, 32)
resize_video_2 | input: | (None, 10, 56, 56, 32)
ResizeVideo | output: | (None, 10, 28, 28, 32)
project_1 | input: | (None, 10, 28, 28, 32)
Project | output: | (None, 10, 28, 28, 64)
residual_main_2 | input: | (None, 10, 28, 28, 32)
ResidualMain | output: | (None, 10, 28, 28, 64)
add_2 | input: | [(None, 10, 28, 28, 64), (None, 10, 28, 28, 64)]
Add | output: | (None, 10, 28, 28, 64)
resize_video_3 | input: | (None, 10, 28, 28, 64)
ResizeVideo | output: | (None, 10, 14, 14, 64)
project_2 | input: | (None, 10, 14, 14, 64)
Project | output: | (None, 10, 14, 14, 128)
residual_main_3 | input: | (None, 10, 14, 14, 64)
ResidualMain | output: | (None, 10, 14, 14, 128)
add_3 | input: | [(None, 10, 14, 14, 128), (None, 10, 14, 14, 128)]
Add | output: | (None, 10, 14, 14, 128)
global_average_pooling3d | input: | (None, 10, 14, 14, 128)
GlobalAveragePooling3D | output: | (None, 128)
flatten | input: | (None, 128)
Flatten | output: | (None, 128)
dense_3 | input: | (None, 128)
Dense | output: | (None, 10)

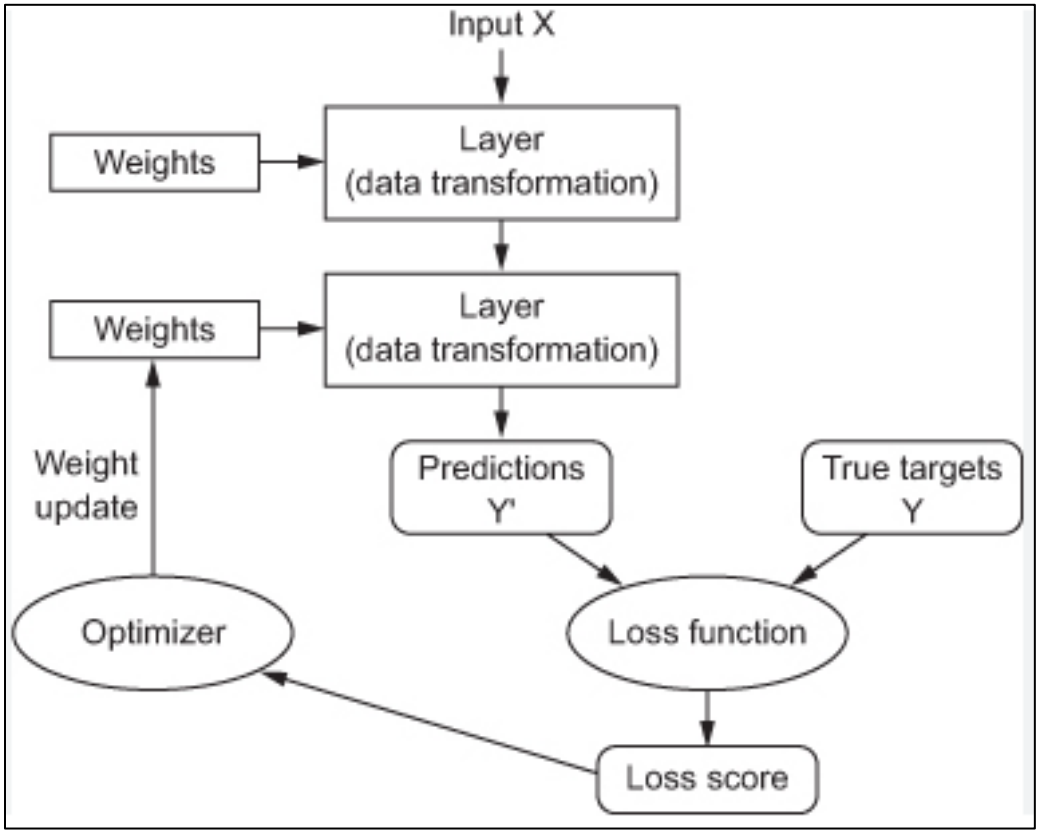
Train the model

For this project, 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.

```
model.compile(loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer = keras.optimizers.Adam(learning_rate = 0.0001),
              metrics = ['accuracy'])
```

Train the model for 50 epoches with the Keras Model.fit method.

```
history = model.fit(x = train_ds,
                    epochs = 50,
                    validation_data = val_ds)
```



We are nearing the end of this project, and we should now have a general understanding of what's going on behind the screen in a neural network. What was a magical black box at the start of the project has turn into a clearer picture, as illustrated in figure: the model, composed of layers that are chained together, maps the input data to predictions. The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the models predictions math what was expected. The optimizer uses this loss value to update the model's weights.

Visualize the results

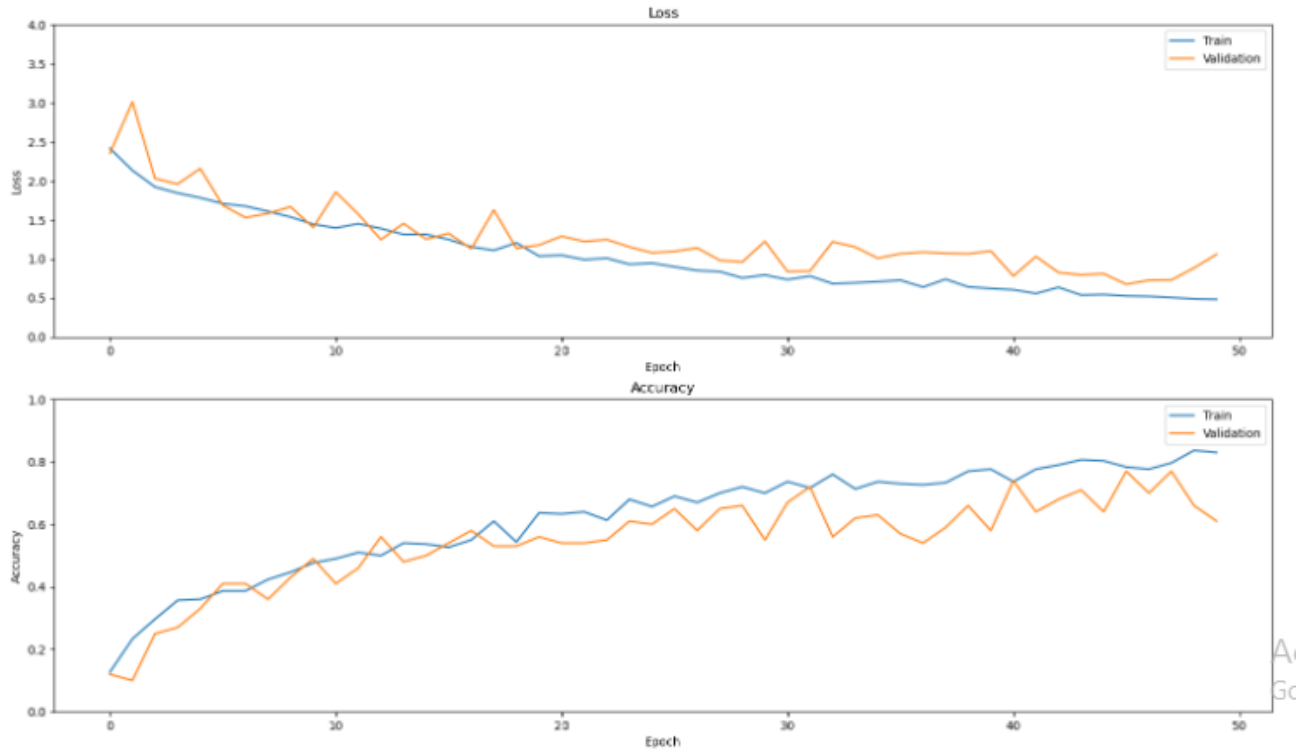
Create plots of the loss and accuracy on the training and validation sets: `Epoch 1/50`

```
150/150 [==============================] - 406s 3s/step - loss: 2.4153 - a
ccuracy: 0.1267 - val_loss: 2.3512 - val_accuracy: 0.1200
Epoch 2/50
150/150 [==============================] - 422s 3s/step - loss: 2.1323 - a
ccuracy: 0.2333 - val_loss: 3.0109 - val_accuracy: 0.1000
Epoch 3/50
150/150 [==============================] - 415s 3s/step - loss: 1.9209 - a
ccuracy: 0.2967 - val_loss: 2.0293 - val_accuracy: 0.2500
Epoch 4/50
150/150 [==============================] - 414s 3s/step - loss: 1.8423 - a
ccuracy: 0.3567 - val_loss: 1.9581 - val_accuracy: 0.2700
Epoch 5/50
150/150 [==============================] - 414s 3s/step - loss: 1.7821 - a
ccuracy: 0.3600 - val_loss: 2.1571 - val_accuracy: 0.3300
Epoch 6/50
150/150 [==============================] - 411s 3s/step - loss: 1.7078 - a
ccuracy: 0.3867 - val_loss: 1.6870 - val_accuracy: 0.4100
Epoch 7/50
150/150 [==============================] - 425s 3s/step - loss: 1.6744 - a
ccuracy: 0.3867 - val_loss: 1.5279 - val_accuracy: 0.4100
Epoch 8/50
150/150 [==============================] - 421s 3s/step - loss: 1.6087 - a
ccuracy: 0.4233 - val_loss: 1.5819 - val_accuracy: 0.3600
Epoch 9/50
150/150 [==============================] - 414s 3s/step - loss: 1.5362 - a
ccuracy: 0.4467 - val_loss: 1.6673 - val_accuracy: 0.4300
Epoch 10/50
150/150 [==============================] - 422s 3s/step - loss: 1.4440 - a
ccuracy: 0.4767 - val_loss: 1.4023 - val_accuracy: 0.4900
Epoch 11/50
150/150 [==============================] - 422s 3s/step - loss: 1.3955 - a
ccuracy: 0.4900 - val_loss: 1.8552 - val_accuracy: 0.4100
Epoch 12/50
150/150 [==============================] - 427s 3s/step - loss: 1.4499 - a
ccuracy: 0.5100 - val_loss: 1.5726 - val_accuracy: 0.4600
Epoch 13/50
150/150 [==============================] - 414s 3s/step - loss: 1.3889 - a
ccuracy: 0.5000 - val_loss: 1.2471 - val_accuracy: 0.5600
Epoch 14/50
150/150 [==============================] - 415s 3s/step - loss: 1.3128 - a
ccuracy: 0.5400 - val_loss: 1.4530 - val_accuracy: 0.4800
Epoch 15/50
150/150 [==============================] - 404s 3s/step - loss: 1.3139 - a
ccuracy: 0.5367 - val_loss: 1.2517 - val_accuracy: 0.5000
Epoch 16/50
150/150 [==============================] - 405s 3s/step - loss: 1.2483 - a
ccuracy: 0.5267 - val_loss: 1.3261 - val_accuracy: 0.5400
Epoch 17/50
150/150 [==============================] - 425s 3s/step - loss: 1.1537 - a
ccuracy: 0.5500 - val_loss: 1.1288 - val_accuracy: 0.5800
```

```
Epoch 18/50
150/150 [==============================] - 404s 3s/step - loss: 1.1094 - a
ccuracy: 0.6100 - val_loss: 1.6262 - val_accuracy: 0.5300
Epoch 19/50
150/150 [==============================] - 426s 3s/step - loss: 1.2051 - a
ccuracy: 0.5433 - val_loss: 1.1362 - val_accuracy: 0.5300
Epoch 20/50
150/150 [==============================] - 418s 3s/step - loss: 1.0354 - a
ccuracy: 0.6367 - val_loss: 1.1767 - val_accuracy: 0.5600
Epoch 21/50
150/150 [==============================] - 440s 3s/step - loss: 1.0471 - a
ccuracy: 0.6333 - val_loss: 1.2901 - val_accuracy: 0.5400
Epoch 22/50
150/150 [==============================] - 460s 3s/step - loss: 0.9900 - a
ccuracy: 0.6400 - val_loss: 1.2230 - val_accuracy: 0.5400
Epoch 23/50
150/150 [==============================] - 522s 3s/step - loss: 1.0084 - a
ccuracy: 0.6133 - val_loss: 1.2464 - val_accuracy: 0.5500
Epoch 24/50
150/150 [==============================] - 457s 3s/step - loss: 0.9307 - a
ccuracy: 0.6800 - val_loss: 1.1540 - val_accuracy: 0.6100
Epoch 25/50
150/150 [==============================] - 408s 3s/step - loss: 0.9451 - a
ccuracy: 0.6567 - val_loss: 1.0780 - val_accuracy: 0.6000
Epoch 26/50
150/150 [==============================] - 527s 4s/step - loss: 0.8980 - a
ccuracy: 0.6900 - val_loss: 1.0966 - val_accuracy: 0.6500
Epoch 27/50
150/150 [==============================] - 467s 3s/step - loss: 0.8508 - a
ccuracy: 0.6700 - val_loss: 1.1401 - val_accuracy: 0.5800
Epoch 28/50
150/150 [==============================] - 462s 3s/step - loss: 0.8369 - a
ccuracy: 0.7000 - val_loss: 0.9815 - val_accuracy: 0.6500
Epoch 29/50
150/150 [==============================] - 464s 3s/step - loss: 0.7562 - a
ccuracy: 0.7200 - val_loss: 0.9604 - val_accuracy: 0.6600
Epoch 30/50
150/150 [==============================] - 453s 3s/step - loss: 0.7936 - a
ccuracy: 0.7000 - val_loss: 1.2242 - val_accuracy: 0.5500
Epoch 31/50
150/150 [==============================] - 459s 3s/step - loss: 0.7343 - a
ccuracy: 0.7367 - val_loss: 0.8378 - val_accuracy: 0.6700
Epoch 32/50
150/150 [==============================] - 465s 3s/step - loss: 0.7797 - a
ccuracy: 0.7167 - val_loss: 0.8445 - val_accuracy: 0.7200
Epoch 33/50
150/150 [==============================] - 452s 3s/step - loss: 0.6823 - a
ccuracy: 0.7600 - val_loss: 1.2184 - val_accuracy: 0.5600
Epoch 34/50
150/150 [==============================] - 447s 3s/step - loss: 0.6933 - a
ccuracy: 0.7133 - val_loss: 1.1517 - val_accuracy: 0.6200
Epoch 35/50
150/150 [==============================] - 412s 3s/step - loss: 0.7079 - a
ccuracy: 0.7367 - val_loss: 1.0083 - val_accuracy: 0.6300
```

```
Epoch 36/50
150/150 [==============================] - 401s 3s/step - loss: 0.7267 - a
ccuracy: 0.7300 - val_loss: 1.0660 - val_accuracy: 0.5700
Epoch 37/50
150/150 [==============================] - 403s 3s/step - loss: 0.6382 - a
ccuracy: 0.7267 - val_loss: 1.0862 - val_accuracy: 0.5400
Epoch 38/50
150/150 [==============================] - 396s 3s/step - loss: 0.7395 - a
ccuracy: 0.7333 - val_loss: 1.0717 - val_accuracy: 0.5900
Epoch 39/50
150/150 [==============================] - 412s 3s/step - loss: 0.6407 - a
ccuracy: 0.7700 - val_loss: 1.0655 - val_accuracy: 0.6600
Epoch 40/50
150/150 [==============================] - 416s 3s/step - loss: 0.6202 - a
ccuracy: 0.7767 - val_loss: 1.1008 - val_accuracy: 0.5800
Epoch 41/50
150/150 [==============================] - 437s 3s/step - loss: 0.6043 - a
ccuracy: 0.7367 - val_loss: 0.7801 - val_accuracy: 0.7400
Epoch 42/50
150/150 [==============================] - 424s 3s/step - loss: 0.5561 - a
ccuracy: 0.7767 - val_loss: 1.0312 - val_accuracy: 0.6400
Epoch 43/50
150/150 [==============================] - 432s 3s/step - loss: 0.6355 - a
ccuracy: 0.7900 - val_loss: 0.8248 - val_accuracy: 0.6800
Epoch 44/50
150/150 [==============================] - 18183s 122s/step - loss: 0.5362
- accuracy: 0.8067 - val_loss: 0.7946 - val_accuracy: 0.7100
Epoch 45/50
150/150 [==============================] - 412s 3s/step - loss: 0.5427 - a
ccuracy: 0.8033 - val_loss: 0.8100 - val_accuracy: 0.6400
Epoch 46/50
150/150 [==============================] - 413s 3s/step - loss: 0.5251 - a
ccuracy: 0.7833 - val_loss: 0.6742 - val_accuracy: 0.7700
Epoch 47/50
150/150 [==============================] - 414s 3s/step - loss: 0.5192 - a
ccuracy: 0.7767 - val_loss: 0.7254 - val_accuracy: 0.7000
Epoch 48/50
150/150 [==============================] - 414s 3s/step - loss: 0.5047 - a
ccuracy: 0.7967 - val_loss: 0.7289 - val_accuracy: 0.7700
Epoch 49/50
150/150 [==============================] - 412s 3s/step - loss: 0.4861 - a
ccuracy: 0.8367 - val_loss: 0.8839 - val_accuracy: 0.6600
Epoch 50/50
150/150 [==============================] - 411s 3s/step - loss: 0.4802 - a
ccuracy: 0.8300 - val_loss: 1.0592 - val_accuracy: 0.6100
```

```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.show()

plot_history(history)
```



Evaluate the model

Use Keras Model.evaluate to get the loss and accuracy on the test dataset.

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

```
13/13 [==============================] - 28s 2s/step - loss: 1.1398 - accuracy: 0.5700
```

Out[66]:

```
{'loss': 1.1398175954818726, 'accuracy': 0.5699999928474426}
```

To visualize model performance further, use a confusion matrix. The confusion matrix allows you to assess the performance of the classification model beyond accuracy. In order to build the confusion matrix for this multi-class classification problem, get the actual values in the test set and the predicted values.

```
def get_actual_predicted_labels(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
```

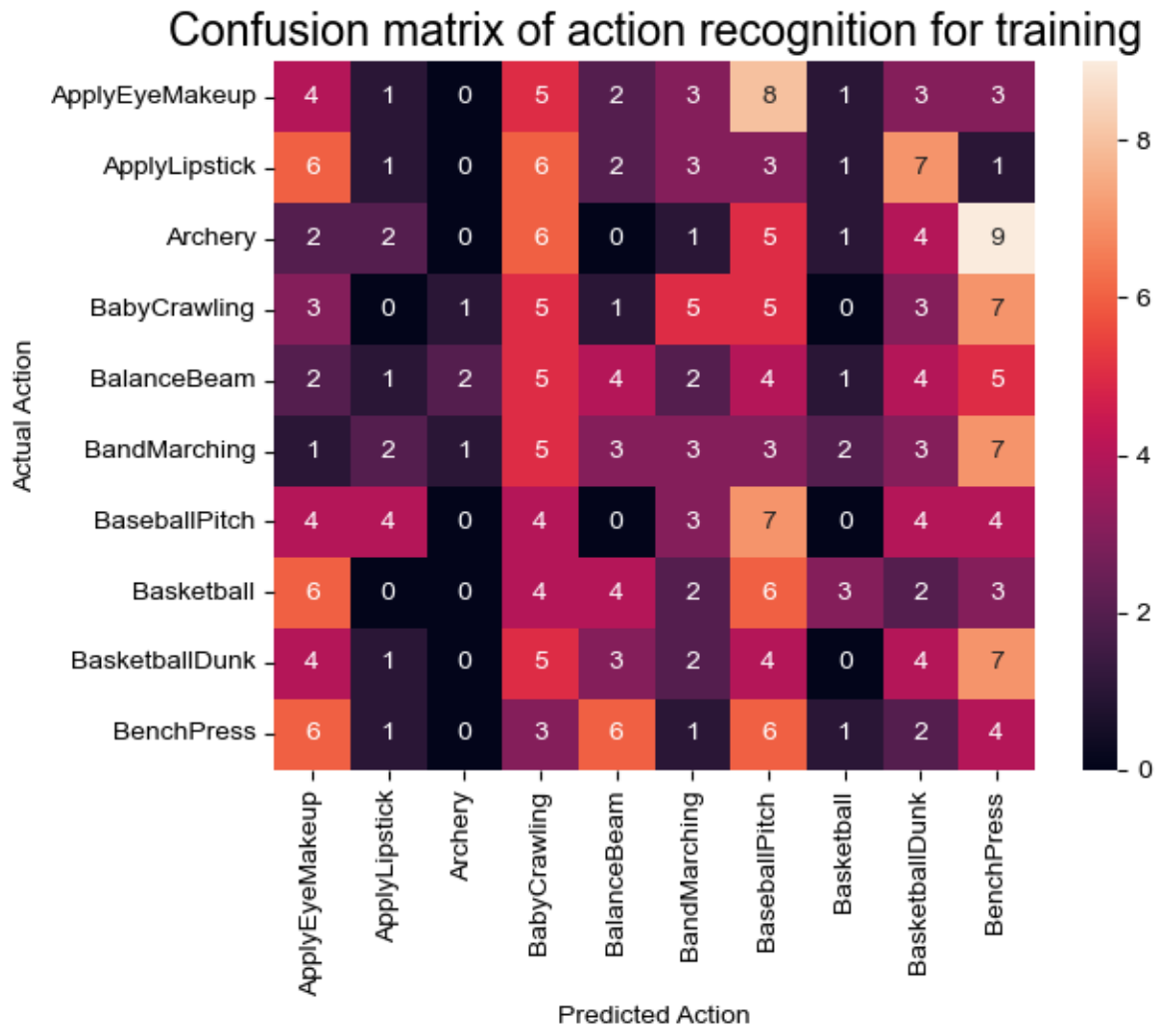
```
def plot_confusion_matrix(actual, predicted, labels, ds_type):
  cm = tf.math.confusion_matrix(actual, predicted)
  ax = sns.heatmap(cm, annot=True, fmt='g')
  sns.set(rc={'figure.figsize':(12, 12)})
  sns.set(font_scale=1.4)
  ax.set_title('Confusion matrix of action recognition for ' + ds_type)
  ax.set_xlabel('Predicted Action')
  ax.set_ylabel('Actual Action')
  plt.xticks(rotation=90)
  plt.yticks(rotation=0)
  ax.xaxis.set_ticklabels(labels)
  ax.yaxis.set_ticklabels(labels)
```

```
fg = FrameGenerator(subset_paths['train'], n_frames, training=True)
labels = list(fg.class_ids_for_name.keys())
```

```
actual, predicted = get_actual_predicted_labels(train_ds)
plot_confusion_matrix(actual, predicted, labels, 'training')
```

38

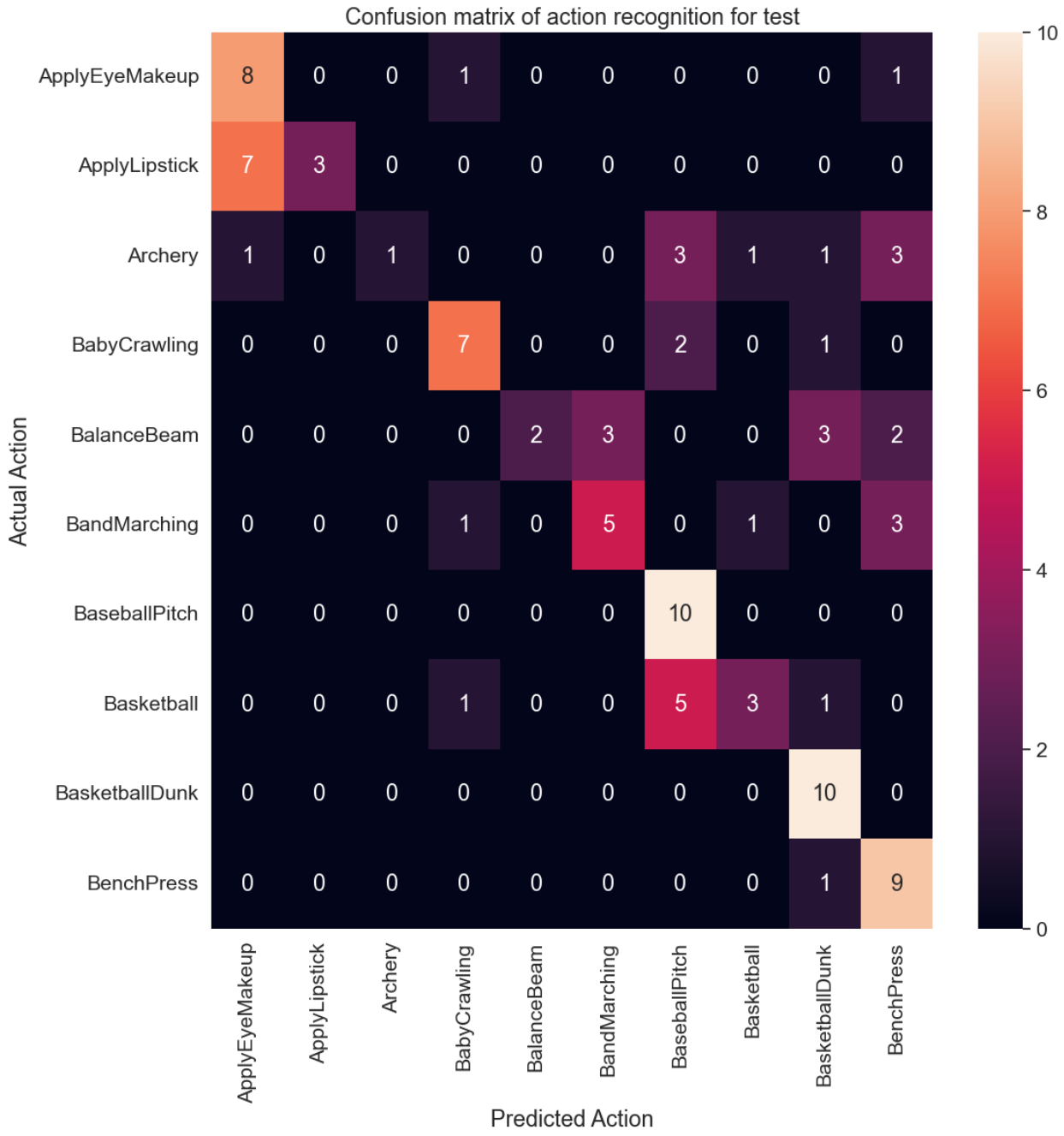
```
/38 [==============================] - 85s 2s/step
```



```
actual, predicted = get_actual_predicted_labels(test_ds)
plot_confusion_matrix(actual, predicted, labels, 'test')
```

1

```
3/13 [==============================] - 29s 2s/step
```



The precision and recall values for each class can also be calculated using a confusion matrix.

```
def calculate_classification_metrics(y_actual, y_pred, labels):
  """
    Calculate the precision and recall of a classification model using the ground truth and
    predicted values.

    Args:
      y_actual: Ground truth labels.
      y_pred: Predicted labels.
      labels: List of classification labels.

    Return:
      Precision and recall measures.
  """
  cm = tf.math.confusion_matrix(y_actual, y_pred)
  tp = np.diag(cm) # Diagonal represents true positives
  precision = dict()
  recall = dict()
  for i in range(len(labels)):
    col = cm[:, i]
    fp = np.sum(col) - tp[i] # Sum of column minus true positive is false negative

    row = cm[i, :]
    fn = np.sum(row) - tp[i] # Sum of row minus true positive, is false negative

    precision[labels[i]] = tp[i] / (tp[i] + fp) # Precision

    recall[labels[i]] = tp[i] / (tp[i] + fn) # Recall

  return precision, recall
```

```
precision, recall = calculate_classification_metrics(actual, predicted, labels) # Test dataset
```

```
precision
```

```
{'ApplyEyeMakeup': 0.5,
 'ApplyLipstick': 1.0,
 'Archery': 1.0,
 'BabyCrawling': 0.7,
 'BalanceBeam': 1.0,
 'BandMarching': 0.625,
 'BaseballPitch': 0.5,
 'Basketball': 0.6,
 'BasketballDunk': 0.5882352941176471,
 'BenchPress': 0.5}
```

```
recall
```

```
{'ApplyEyeMakeup': 0.8,
 'ApplyLipstick': 0.3,
 'Archery': 0.1,
 'BabyCrawling': 0.7,
 'BalanceBeam': 0.2,
 'BandMarching': 0.5,
 'BaseballPitch': 1.0,
 'Basketball': 0.3,
 'BasketballDunk': 1.0,
 'BenchPress': 0.9}
```

```
df=pd.DataFrame([precision,recall],index=['precision','recall'])
df
```

	ApplyEyeMakeup	ApplyLipstick	Archery	BabyCrawling	BalanceBeam	BandMarching	BaseballPitch	Basketball	BasketballDunk	BenchPress
precision	0.5	1.0	1.0	0.7	1.0	0.625	0.5	0.6	0.588235	0.5
recall	0.8	0.3	0.1	0.7	0.2	0.500	1.0	0.3	1.000000	0.9

Conclueion

The confusion matrix gives you a lot of information, but sometimes you may prefere a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called precision of the cassifier.precision =TP/(TP+FP), TP is the number of true positive , and FP is the number of false positives. A trival way to have perfect precision is to make one single positive prediction and ensure it is correct (precision=1/1=100%). This would not not be very useful since the classifier would ignore all but one positive instance. So precision is pypically used along with another metric named recall, also called sensitivity or true positive rate(TPR): this is the ratio of positive instances that are currectly detected by the classifier recall=TP/(TP+FN), FN is of course the number of false nagative.From above tabe we can see that precision of BabyCrawling 70% also recall 70% here currectly classified but In Rrchary and BalanceBeam precision high 100% but recall 10 and 20 percent respectively.These are not curectly classified.Here need to collect more data to tain model.