Show code

View on TensorFlow.org Run in Google Colab View source on GitHub Download notebook

Video classification with a 3D convolutional neural network

This tutorial demonstrates training a 3D convolutional neural network (CNN) for video classification using the UCF101 action recognition dataset. A 3D CNN uses a three-dimensional filter to perform convolutions. The kernel is able to slide in three directions, whereas in a 2D CNN it can slide in two dimensions. The model is based on the work published in A Closer Look at Spatiotemporal Convolutions for Action Recognition by D. Tran et al. (2017). In this tutorial, you will:

- Build an input pipeline
- Build a 3D convolutional neural network model with residual connections using Keras functional API
- Train the model
- Evaluate and test the model

This video classification tutorial is the second part in a series of TensorFlow video tutorials. Here are the other three tutorials:

- Load video data: This tutorial explains much of the code used in this document.
- MoViNet for streaming action recognition: Get familiar with the MoViNet models that are available on TF Hub.
- Transfer learning for video classification with MoViNet: This tutorial explains how to use a pre-trained video classification model trained on a different dataset with the UCF-101 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, einops for performing more complex tensor operations, and tensorflow_docs for embedding data in a Jupyter notebook.

```
1 !pip install remotezip tqdm opencv-python einops
2 !pip install -U tensorflow keras
```

```
Collecting remotezip
  Downloading remotezip-0.12.3-py3-none-any.whl.metadata (7.2 kB)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: opencv-python in /usr/local/lib/python3.10/di
Requirement already satisfied: einops in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/di
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/py
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dis
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.
Downloading remotezip-0.12.3-py3-none-any.whl (8.1 kB)
Installing collected packages: remotezip
Successfully installed remotezip-0.12.3
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-
Collecting tensorflow
  Downloading tensorflow-2.18.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packa
Collecting keras
  Downloading keras-3.7.0-py3-none-any.whl.metadata (5.8 kB)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/c
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3
Requirement already satisfied: flatbuffers>=24.3.25 in /usr/local/lib/pyt
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.1
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/py
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3
Collecting tensorboard<2.19,>=2.18 (from tensorflow)
  Downloading tensorboard-2.18.0-py3-none-any.whl.metadata (1.6 kB)
Requirement already satisfied: numpy<2.1.0,>=1.26.0 in /usr/local/lib/python
Requirement already satisfied: h5py>=3.11.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: ml-dtypes<0.5.0,>=0.4.0 in /usr/local/lib/pyt
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/py
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dis
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr
```

```
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/pytho
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/pyt
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.1
```

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

import cv2
import einops
import numpy as np
import remotezip as rz
import seaborn as sns
import matplotlib.pyplot as plt

import tensorflow as tf
import keras
from keras import layers
```

Load and preprocess video data

The hidden cell below defines helper functions to download a slice of data from the UCF-101 dataset, and load it into a `tf.data.Dataset`. You can learn more about the specific preprocessing steps in the [Loading video data tutorial](), which walks you through this code in more detail.

The `FrameGenerator` class at the end of the hidden block is the most important utility here. It creates an iterable object that can feed data into the TensorFlow data pipeline. Specifically, this class contains a Python generator that loads the video frames along with its encoded label. 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.

```
#@title

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

    Args:
      zip_url: Zip URL containing data.
      to_dir: 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 rem
    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 download
  """
  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_ufc_101_subset(zip_url, num_classes, splits, download_dir):
  """
    Download a subset of the UFC101 dataset and split them into various parts,
    training, validation, and test.

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

    Return:
      dir: Posix path of the resulting directories containing the splits of da
  """
  files = list_files_per_class(zip_url)
  for f in files:
    tokens = f.split('/')
    if len(tokens) <= 2:
      files.remove(f) # Remove that item from the list if it does not have a f

  files_for_class = get_files_per_class(files)

  classes = list(files_for_class.keys())[:num_classes]

```

```
  for cls in classes:
    new_files_for_class = files_for_class[cls]
    random.shuffle(new_files_for_class)
    files_for_class[cls] = new_files_for_class

  # Only use the number of classes you want in the dictionary
  files_for_class = {x: files_for_class[x] for x in list(files_for_class)[:num

  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_co
    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), fram
  """
    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, chann
  """
  # 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 imag
  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

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_
    self.class_ids_for_name = dict((name, idx) for idx, name in enumerate(self

  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
```

```
URL = 'https://storage.googleapis.com/thumos14_files/UCF101_videos.zip'
download_dir = pathlib.Path('./UCF101_subset/')
subset_paths = download_ufc_101_subset(URL,
                        num_classes = 10,
                        splits = {"train": 30, "val": 10, "test": 10},
                        download_dir = download_dir)
```

```
train :
100%|██████████| 300/300 [02:38<00:00,  1.89it/s]
val :
100%|██████████| 100/100 [00:52<00:00,  1.91it/s]
test :
100%|██████████| 100/100 [00:52<00:00,  1.90it/s]
```

Create the training, validation, and test sets (`train_ds` , `val_ds` , and `test_ds`).

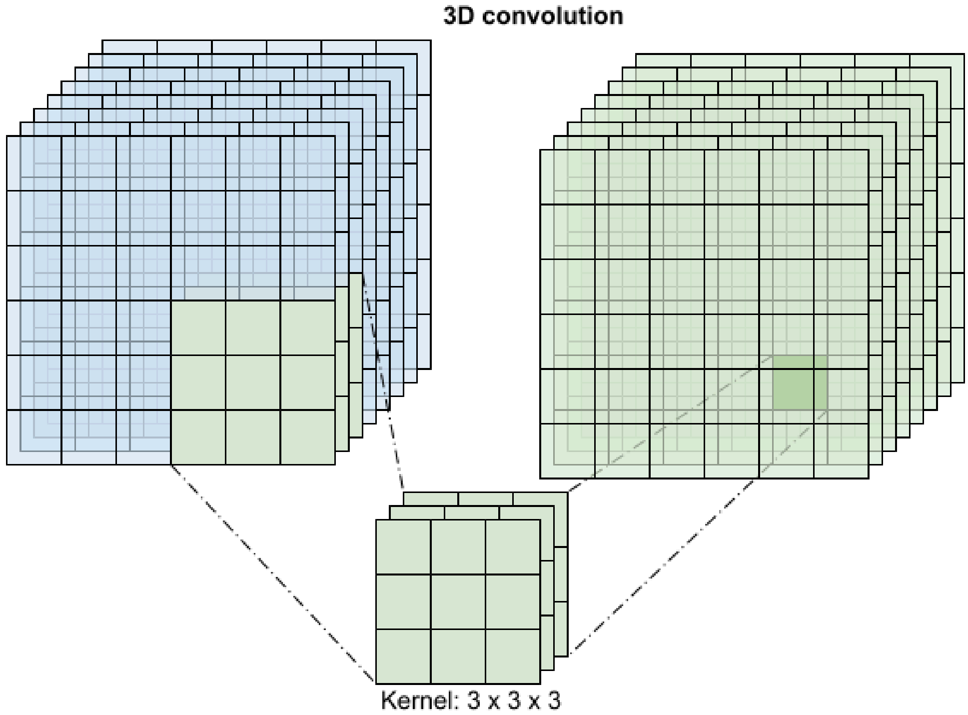
```
n_frames = 10
batch_size = 8

output_signature = (tf.TensorSpec(shape = (None, None, None, 3), dtype = tf.fl
                    tf.TensorSpec(shape = (), dtype = tf.int16))

train_ds = tf.data.Dataset.from_generator(FrameGenerator(subset_paths['train']
                                          output_signature = output_signature)


# Batch the data
train_ds = train_ds.batch(batch_size)

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

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

test_ds = test_ds.batch(batch_size)
```

Create the model

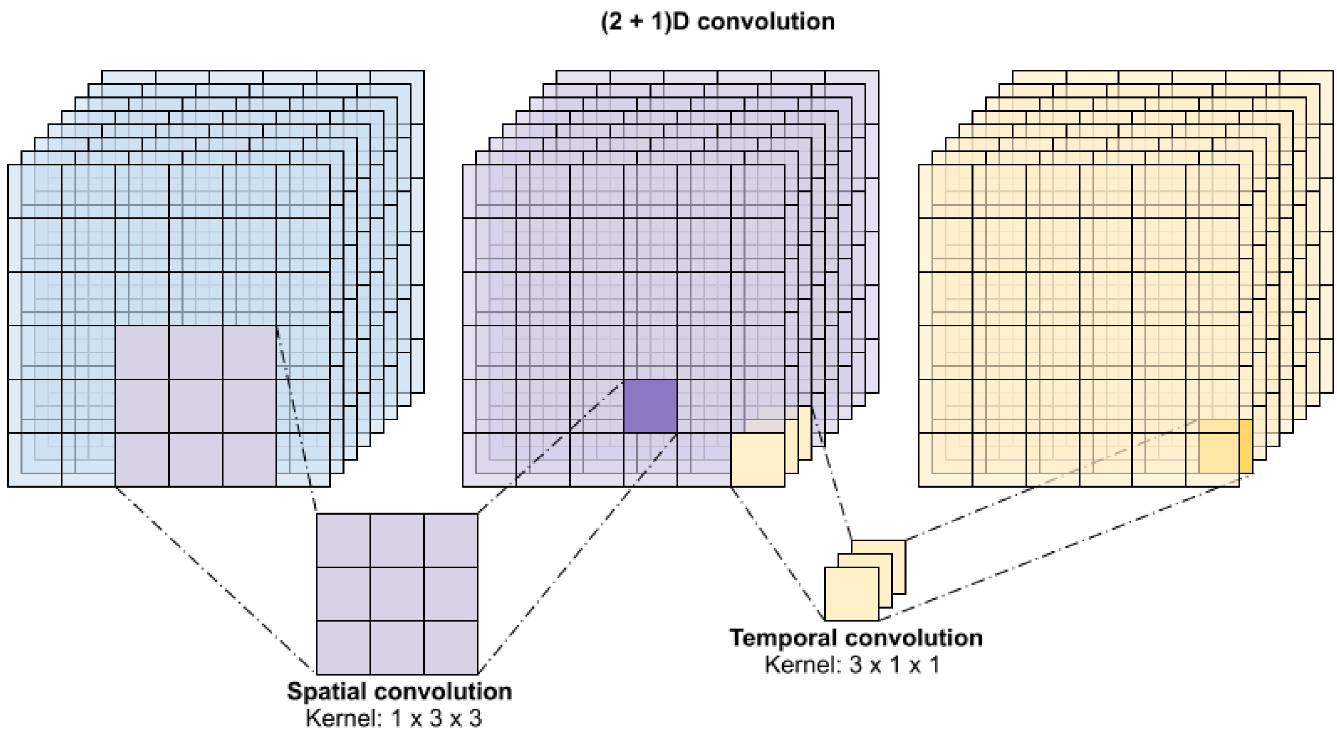
The following 3D convolutional neural network model is based off the paper A Closer Look at Spatiotemporal Convolutions for Action Recognition by D. Tran et al. (2017). The paper compares several versions of 3D ResNets. Instead of operating on a single image with dimensions `(height, width)` , like standard ResNets, these operate on video volume `(time, height, width)` . The most obvious approach to this problem would be replace each 2D convolution (`layers.Conv2D`) with a 3D convolution (`layers.Conv3D`).

This tutorial uses a (2 + 1)D convolution with residual connections. The (2 + 1)D convolution allows for the decomposition of the spatial and temporal dimensions, therefore creating two separate steps. An advantage of this approach is that factorizing the convolutions into spatial and temporal dimensions saves parameters.

For each output location a 3D convolution combines all the vectors from a 3D patch of the volume to create one vector in the output volume.



This operation is takes `time * height * width * channels` inputs and produces `channe` outputs (assuming the number of input and output channels are the same. So a 3D convolution layer with a kernel size of `(3 x 3 x 3)` would need a weight-matrix with `27 * channels ** 2` entries. The reference paper found that a more effective & efficient approach was to factorize the convolution. Instead of a single 3D convolution to process the time and space dimensions, they proposed a "(2+1)D" convolution which processes the space and time dimensions separately. The figure below shows the factored spatial and temporal convolutions of a (2 + 1)D convolution.



The main advantage of this approach is that it reduces the number of parameters. In the (2 + 1)D convolution the spatial convolution takes in data of the shape `(1, width, height)`, while the temporal convolution takes in data of the shape `(time, 1, 1)`. For example, a (2 + 1)D convolution with kernel size `(3 x 3 x 3)` would need weight matrices of size `(9 * channels**2) + (3 * channels**2)`, less than half as many as the full 3D convolution. Th tutorial implements (2 + 1)D ResNet18, where each convolution in the resnet is replaced by a (2+1)D convolution.

```
# Define the dimensions of one frame in the set of frames created
HEIGHT = 224
WIDTH = 224
```

```
class Conv2Plus1D(keras.layers.Layer):
  def __init__(self, filters, kernel_size, padding):
    """
      A sequence of convolutional layers that first apply the convolution oper
      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 dif
    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 matche
  """
  out = ResidualMain(filters,
                     kernel_size)(input)

  res = input
  # Using the Keras functional APIs, project the last dimension of the tensor
  # 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.

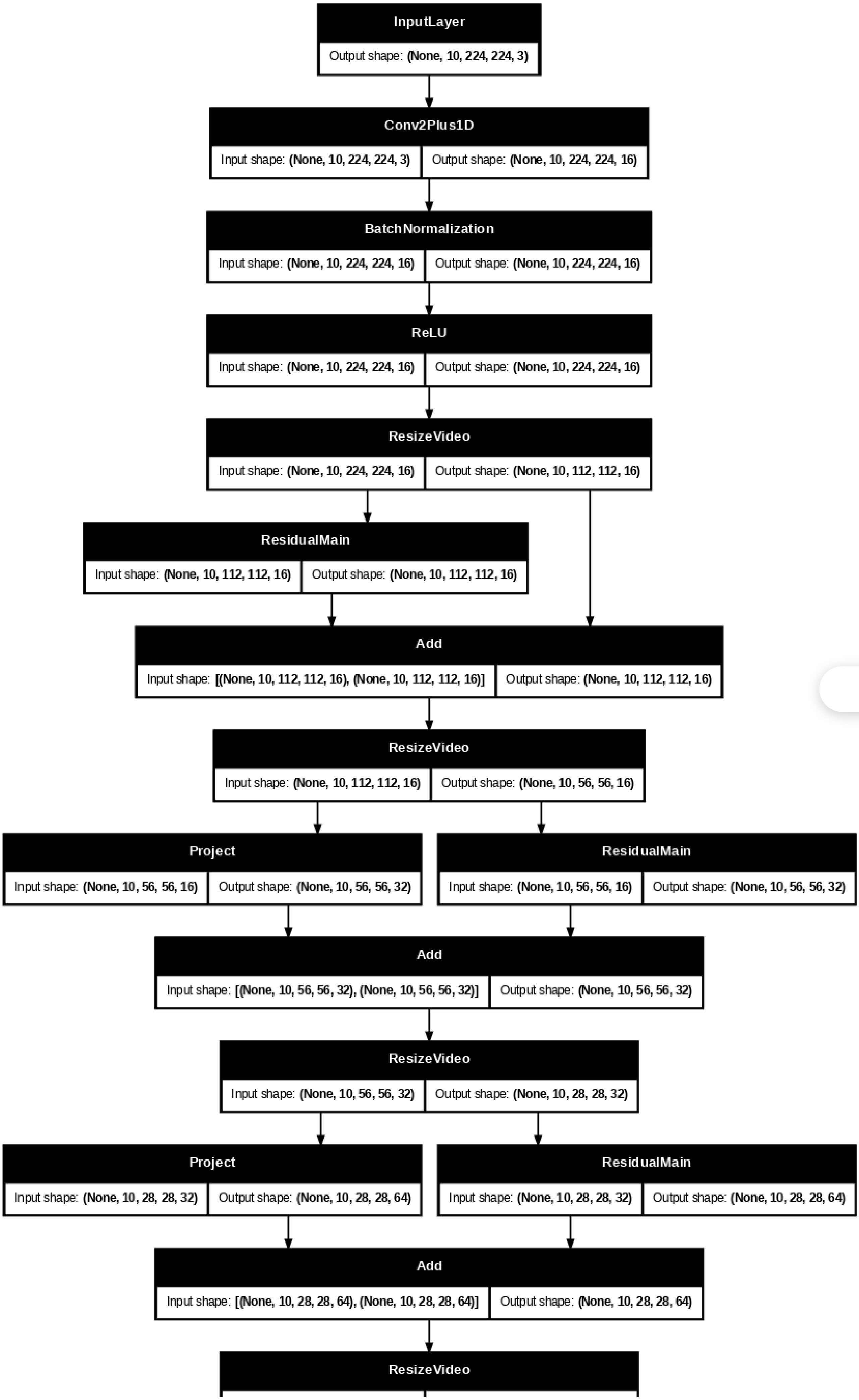
```
class ResizeVideo(keras.layers.Layer):
  def __init__(self, height, width):
    super().__init__()
    self.height = height
    self.width = width
    self.resizing_layer = layers.Resizing(self.height, self.width)

  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 fra

      Return:
        A downsampled size of the video according to the new height and width
    """
    # 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
```

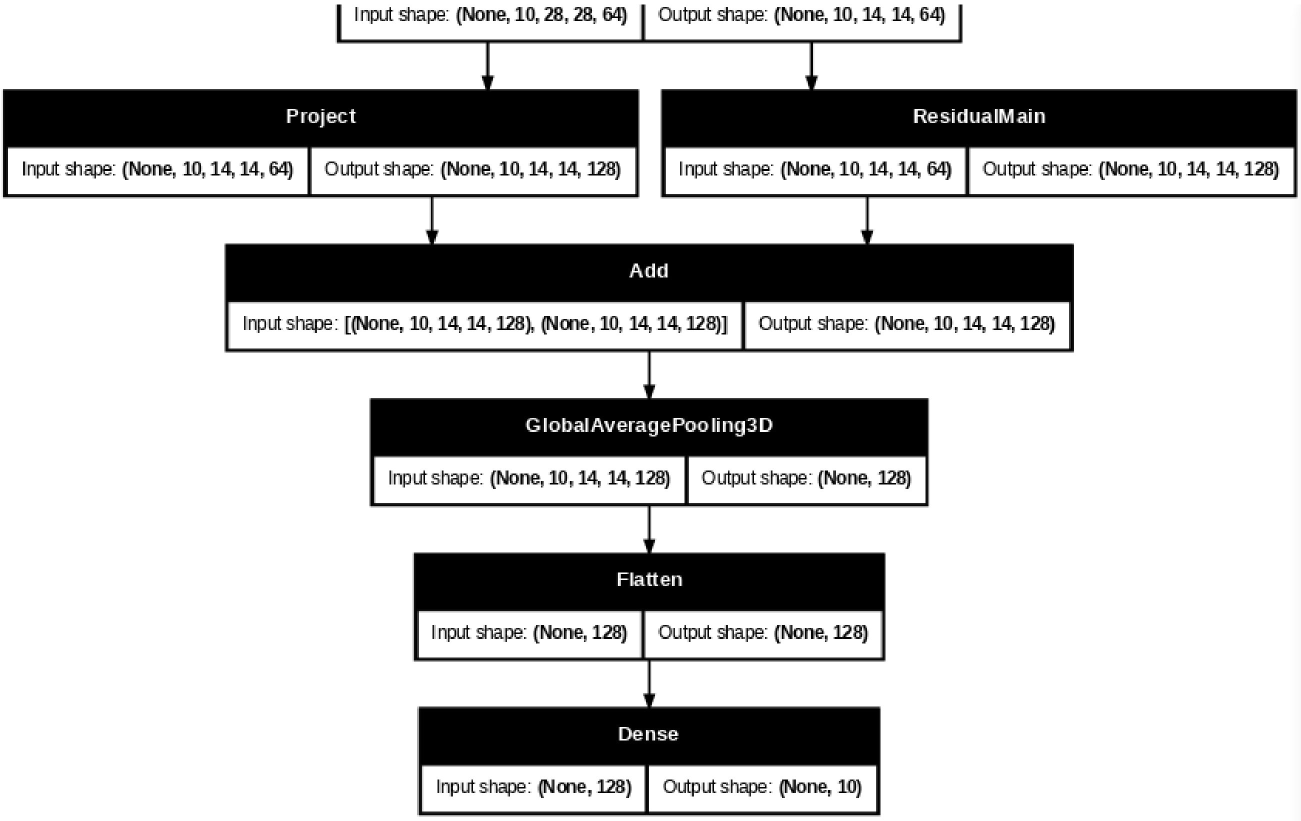
Use the [Keras functional API](https://www.tensorflow.org/guide/keras/functional) to build the residual network.

```
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)
```

```
# Visualize the model
keras.utils.plot_model(model, expand_nested=True, dpi=60, show_shapes=True)
```


InputLayer
Output shape: (None, 10, 224, 224, 3)
Conv2Plus1D
Input shape: (None, 10, 224, 224, 3)
Output shape: (None, 10, 224, 224, 16)
BatchNormalization
Input shape: (None, 10, 224, 224, 16)
Output shape: (None, 10, 224, 224, 16)
ReLU
Input shape: (None, 10, 224, 224, 16)
Output shape: (None, 10, 224, 224, 16)
ResizeVideo
Input shape: (None, 10, 224, 224, 16)
Output shape: (None, 10, 112, 112, 16)
ResidualMain
Input shape: (None, 10, 112, 112, 16)
Output shape: (None, 10, 112, 112, 16)
Add
Input shape: [(None, 10, 112, 112, 16), (None, 10, 112, 112, 16)]
Output shape: (None, 10, 112, 112, 16)
ResizeVideo
Input shape: (None, 10, 112, 112, 16)
Output shape: (None, 10, 56, 56, 16)
Project
Input shape: (None, 10, 56, 56, 16)
Output shape: (None, 10, 56, 56, 32)
ResidualMain
Input shape: (None, 10, 56, 56, 16)
Output shape: (None, 10, 56, 56, 32)
Add
Input shape: [(None, 10, 56, 56, 32), (None, 10, 56, 56, 32)]
Output shape: (None, 10, 56, 56, 32)
ResizeVideo
Input shape: (None, 10, 56, 56, 32)
Output shape: (None, 10, 28, 28, 32)
Project
Input shape: (None, 10, 28, 28, 32)
Output shape: (None, 10, 28, 28, 64)
ResidualMain
Input shape: (None, 10, 28, 28, 32)
Output shape: (None, 10, 28, 28, 64)
Add
Input shape: [(None, 10, 28, 28, 64), (None, 10, 28, 28, 64)]
Output shape: (None, 10, 28, 28, 64)
ResizeVideo



Train the model

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.

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

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

Note: This example model is trained on fewer data points (300 training and 100 validation examples) to keep training time reasonable for this tutorial. Moreover, this example model may take over one hour to train.

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

```
38/38 ━━━━━━━━━━━━━━━━━━━━ 457s 12s/step - accuracy: 0.7793 - loss: 0.6159 -
Epoch 44/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 443s 12s/step - accuracy: 0.7771 - loss: 0.6183 -
Epoch 45/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 464s 12s/step - accuracy: 0.8065 - loss: 0.5311 -
Epoch 46/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 460s 12s/step - accuracy: 0.8445 - loss: 0.5099 -
Epoch 47/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 442s 12s/step - accuracy: 0.8427 - loss: 0.4836 -
Epoch 48/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 460s 12s/step - accuracy: 0.8750 - loss: 0.4656 -
Epoch 49/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 505s 12s/step - accuracy: 0.7711 - loss: 0.6210 -
Epoch 50/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 502s 12s/step - accuracy: 0.8838 - loss: 0.4403 -
```

Visualize the results

Create plots of the loss and accuracy on the training and validation sets:

```
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.

Note: The example model in this tutorial uses a subset of the UCF101 dataset to keep training time reasonable. The accuracy and loss can be improved with further hyperparameter tuning or more training data.

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

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 42s 3s/step - accuracy: 0.7599 - loss: 0.7216
{'accuracy': 0.7300000190734863, 'loss': 0.8357596397399902}
```

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 

    Args:
      dataset: An iterable data structure, such as a TensorFlow Dataset, with

    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
```