Lab 1

Running TensorFlow Lite Models on Raspberry Pi

Table of Contents

Lab 1: Running TensorFlow Lite Models on Raspberry Pi....1
1.0 Introduction....1
1.1 Installing TensorFlow Lite to your Raspberry Pi....1
1.2 Download MobileNet....1
1.3 Classifying a Single Image....2
1.4 Conclusion....4
1.5 TensorFlow Lite example apps (pre-trained)....5
1.5.1 Classifying Objects (WebCam)....5
1.5.1.1 Install (setup) the example files....5
1.5.1.2 Run the example....5
1.5.1.3 Complete code:....5
To do:....7
1.2.2 Objects Detection....8
1.2.2.1 Run the example....8
1.2.2.2 Complete code....8
To do:....10
1.2.3 Pose estimation....11
1.2.3.1 Install the dependencies....11
1.2.3.2 Run the pose estimation sample....11
1.2.3.3 Run the pose classification sample....11
1.2.3.4 Complete code....11
1.2.3.5 Customization options....14
1.2.3.6 Visualize pose estimation result of test data....14
1.2.3.7 Complete code....14
1.2.4 Segmentation....17
1.2.4.1 Run the example....17
1.2.4.2 Complete code....17
1.2.5 Video classification....21
1.2.5.1 Run the example....21
1.2.5.2 Complete code....21
To do:....23
1.2.6 Sound classification....24
1.2.6.1 Install Voicecard software....24
1.2.6.1 Python Libraries....24
1.2.6.2 Python Usage....24
1.2.6.3 Run the example....25
1.2.6.4 Complete code....26
Lab 2: TensorFlow Lite Speech Recognition....28
Simple audio recognition: Recognizing keywords on a Raspberry Pi using Machine Learning (I2S, TensorFlow Lite)....28
2.0 Introduction....28
2.0.1 Objective....28
2.0.2 The Method....28
2.0.3 Project Architecture....29
2.1 Training the Model....29
2.2 Our Jupyter notebook....32
2.2.1 Setup....32
2.2.2 Import the mini Speech Commands dataset....32
2.2.3 Read the audio files and their labels....33
2.2.4 Convert waveforms to spectrograms....35
2.2.5 Build and train the model....37
2.2.6 Evaluate the model performance....39
2.2.6.1 Display a confusion matrix....39
2.2.7 Run inference on an audio file....40

2.2.8 Saving the model in .tflite format....40
2.3 Inference using TensorFlow Lite on Raspberry Pi....42
2.3.1 Audio from an I2S Microphone on WM8960 audio HAT....42
2.3.1.1 Install software for WM8960 audio HAT....42
2.3.1.2 Python Libraries and usage....43
2.3.2 Processing audio for Inference....44
2.3.3 OLED Display (optional)....46
2.3.4 Putting it all Together....47
2.4 Conclusion....51
2.5 Downloads....51

Lab 1

Running TensorFlow Lite Models on Raspberry Pi

1.0 Introduction

The deep learning models created using **TensorFlow** require high processing capabilities to perform inference. Fortunately, there is a lite version of **TensorFlow** called **TensorFlow Lite** (**TFLite** for short) which allows such models to run on devices with limited capabilities. Inference is performed in less than a second.
In this lab we are going to use the prepared OS with ML libraries.

The **first part** of the lab introduces one simple example of using **TensorFlow Lite** `MobileNet` model for the classification of a simple image. The model has been trained for the classification of simple object images.

1.1 Installing `TensorFlow Lite` to your Raspberry Pi

The following command lines is be used to install the `tflite-runtime`:

Note that this library may already be installed with your OS.

```
echo "deb [signed-by=/usr/share/keyrings/coral-edgetpu-archive-keyring.gpg]
https://packages.cloud.google.com/apt coral-edgetpu-stable main" | sudo tee
/etc/apt/sources.list.d/coral-edgetpu.list

curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo tee
/usr/share/keyrings/coral-edgetpu-archive-keyring.gpg >/dev/null

sudo apt update
sudo apt install python3-tflite-runtime

python3
```

Within the Python CLI, it is straightforward to verify that TensorFlow Lite is installed. All we need to do is use the following line within the interface. All this line is doing is importing the `Interpreter` library.

```
from tflite_runtime.interpreter import Interpreter
```

After installing `tflite_runtime` and making your **RPi** ready for making predictions, the next step is to make the **TFLite** model available in the RPi storage (that is to say, by downloading it). The next section discusses downloading the **TFLite** version of `MobileNet`.

1.2 Download `MobileNet`

`MobileNet` has already been converted into a **TFLite** version and is available for download here:

```
https://github.com/nnsuite/testcases/tree/master/DeepLearningModels/tensorflow-lite/
Mobilenet_v1_1.0_224_quant
```

It is a compressed file that not only contains the TFLite model, but also the **class labels** that the **model predictions**.

After decompressing this file, its contents are as follows:

1. `mobilenet_v1_1.0_224_quant.tflite`
2. `labels_mobilenet_quant_v1_224.txt`

There are two versions of **MobileNet**, and each version supports input images of different shapes. Here we'll use **Version 1**, which accepts images of `shape(224,224)`. Finally the model is quantized, which is a step for reducing the model size and reducing the latency of predictions.

We have to create a new folder called `TFLite_MobileNet` in **RPi** to hold these two items, as shown below.

```
/home/pi/TFLite_MobileNet
        mobilenet_v1_1.0_224_quant.tflite
        labels_mobilenet_quant_v1_224.txt
        test.jpg
```

We also included a sample image, **`test.jpg`** (shown below), for being fed to the model for classification.

Now that we've prepared all the required files, in the next section we'll see how to feed this image to the model to predict its class label.

1.3 Classifying a Single Image

The code required for loading the **TFLite** model and classifying an image is listed below. We **start by loading the required libraries**. Then the **paths** of the **model** and the **class labels** are prepared in the **`model_path`** and **`labels`** variables. The model path is then fed to the **`Interpreter`** class constructor for loading it. The loaded model is returned in the **`interpreter`** variable.

```
from tflite_runtime.interpreter import Interpreter
from PIL import Image
import numpy as np
import time

data_folder = "/home/pi/TFLite_MobileNet/"

model_path = data_folder + "mobilenet_v1_1.0_224_quant.tflite"
label_path = data_folder + "labels_mobilenet_quant_v1_224.txt"

interpreter = Interpreter(model_path)
print("Model Loaded Successfully.")

interpreter.allocate_tensors()
_, height, width, _ = interpreter.get_input_details()[0]['shape']
print("Image Shape (", width, ",", height, ")")

# Load an image to be classified.
image = Image.open(data_folder + "test.jpg").convert('RGB').resize((width, height))

# Classify the image.
time1 = time.time()
label_id, prob = classify_image(interpreter, image)
time2 = time.time()
classification_time = np.round(time2-time1, 3)
print("Classificaiton Time =", classification_time, "seconds.")

# Read class labels.
labels = load_labels(label_path)

# Return the classification label of the image.
classification_label = labels[label_id]
print("Image Label is :", classification_label, ", with Accuracy :", np.round(prob*100, 2), "%.")
```

After the model is loaded, the **`allocate_tensors()`** method is called for **allocating memory for the input and output tensors**. After memory allocation, the **`get_input_details()`** method is called to return some information about the input tensor. This includes the **`width`** and **`height`** of the input image. Why do we return this information?

Remember that the loaded model accepts images of **`shape(224,224)`**. If an image of a different size is fed to the model then we'll get an error. By knowing the **`width`** and **`height`** of the image accepted by the model, **we can resize** the input accordingly so that everything will work correctly.

After the **`width`** and **`height`** of the input tensor are returned, the **test image** is read using **`PIL`**, and the returned image size is set equal to the image size that the model accepts.

Now we have both the model and image prepared.
Next, we'll classify the image using the **`classify_image()`** function which is implemented below. Within it, the input tensor of the model is set equal to the test image according to the **`set_input_tensor()`** function.

Next is to use the **`invoke()`** function **to run the model** and propagate the input to get the outputs. The outputs returned are the **class index**, in addition to its **probability**.

```
def classify_image(interpreter, image, top_k=1):
  tensor_index = interpreter.get_input_details()[0]['index']
  input_tensor = interpreter.tensor(tensor_index)()[0]
  input_tensor[:, :] = image

  interpreter.invoke()
  output_details = interpreter.get_output_details()[0]
  output = np.squeeze(interpreter.get_tensor(output_details['index']))

  scale, zero_point = output_details['quantization']
  output = scale * (output - zero_point)

  ordered = np.argpartition(-output, top_k)
  return [(i, output[i]) for i in ordered[:top_k]][0]
```

After returning the classification probability, the class labels are loaded from the text file using the **`load_labels()`** function which is implemented below. It accepts the text file path and returns a **list with the class labels**. The index of the class to which the image is classified is used to return the associated class label.

Finally, this **label is printed**.

```
def load_labels(path): # Read the labels from the text file as a Python list.
  with open(path, 'r') as f:
    return [line.strip() for i, line in enumerate(f.readlines())]
```

The complete code is listed below.

```
from tflite_runtime.interpreter import Interpreter
from PIL import Image
import numpy as np
import time

def load_labels(path): # Read the labels from the text file as a Python list.
  with open(path, 'r') as f:
    return [line.strip() for i, line in enumerate(f.readlines())]

def set_input_tensor(interpreter, image):
  tensor_index = interpreter.get_input_details()[0]['index']
  input_tensor = interpreter.tensor(tensor_index)()[0]
  input_tensor[:, :] = image

def classify_image(interpreter, image, top_k=1):
  set_input_tensor(interpreter, image)

  interpreter.invoke()
  output_details = interpreter.get_output_details()[0]
  output = np.squeeze(interpreter.get_tensor(output_details['index']))

  scale, zero_point = output_details['quantization']
  output = scale * (output - zero_point)

  ordered = np.argpartition(-output, 1)
  return [(i, output[i]) for i in ordered[:top_k]][0]

data_folder = "/home/pi/TFLite_MobileNet/"
```

```
model_path = data_folder + "mobilenet_v1_1.0_224_quant.tflite"
label_path = data_folder + "labels_mobilenet_quant_v1_224.txt"

interpreter = Interpreter(model_path)
print("Model Loaded Successfully.")

interpreter.allocate_tensors()
_, height, width, _ = interpreter.get_input_details()[0]['shape']
print("Image Shape (", width, ",", height, ")")

# Load an image to be classified.
image = Image.open(data_folder + "test.jpg").convert('RGB').resize((width, height))

# Classify the image.
time1 = time.time()
label_id, prob = classify_image(interpreter, image)
time2 = time.time()
classification_time = np.round(time2-time1, 3)
print("Classification Time =", classification_time, "seconds.")

# Read class labels.
labels = load_labels(label_path)

# Return the classification label of the image.
classification_label = labels[label_id]
print("Image Label is :", classification_label, ", with Accuracy :", np.round(prob*100, 2), "%.")
```

The outputs of the print messages are shown below.

```
pi@raspberrypi:~/tensorflow_lite $ python3 imageclass.py
Model Loaded Successfully.
Image Shape ( 224 , 224 )
Classification Time = 0.113 seconds.
Image Label is : Egyptian cat , with Accuracy : 75.78 %.
```

1.4 Conclusion

This **lab** showed how to use TensorFlow Lite on Raspberry Pi. We looked at the sample use case of classifying a single image. There's no need to install the complete TensorFlow package; just **tflite_runtime** is used, which supports the **Interpreter** class. The **MobileNet** model, which is pre-trained and already converted to a **TFLite** model, is used for making predictions.

1.5 TensorFlow Lite example apps (pre-trained)

In this part of the lab we are going to explore pre-trained TensorFlow Lite models and learn how to use them in sample apps for a variety of ML applications.

All examples are available here:

```
git clone https://github.com/tensorflow/examples --depth 1
```

Clone this **Git** repo onto your Raspberry Pi .

1.5.1 Classifying Objects (WebCam)

This example uses TensorFlow Lite with Python on a Raspberry Pi to perform real-time image classification using images streamed from the camera.

1.5.1.1 Install (setup) the example files

Go to the :

```
cd examples/lite/examples/image_classification/raspberry_pi
```

Then use the `setup.sh` script to install a couple Python packages, and download the `TFLite` model:

```
sh setup.sh
```

1.5.1.2 Run the example

```
python3 classify.py
```

- You can optionally specify the `model` parameter to set the TensorFlow Lite model to be used:
  - The default value is `efficientnet_lite0.tflite`
  - TensorFlow Lite image classification models **with metadatafrom** (including models from TensorFlow Hub or models trained with TensorFlow Lite Model Maker are supported.)
  -
- You can optionally specify the `maxResults` parameter to limit the list of classification results:
  - Supported value: A positive integer.
  - Default value: 3.

- Example usage:

```
python3 classify.py \
  --model efficientnet_lite0.tflite \
  --maxResults 5
```

1.5.1.3 Complete code:

```
"""Main script to run image classification."""

import argparse
import sys
import time

import cv2
from tflite_support.task import core
from tflite_support.task import processor
from tflite_support.task import vision

# Visualization parameters
_ROW_SIZE = 20  # pixels
_LEFT_MARGIN = 24  # pixels
_TEXT_COLOR = (0, 0, 255)  # red
_FONT_SIZE = 1
_FONT_THICKNESS = 1
_FPS_AVERAGE_FRAME_COUNT = 10
```

```
def run(model: str, max_results: int, score_threshold: float, num_threads: int,
        enable_edgetpu: bool, camera_id: int, width: int, height: int) -> None:

"""Continuously run inference on images acquired from the camera.

  Args:
      model: Name of the TFLite image classification model.
      max_results: Max of classification results.
      score_threshold: The score threshold of classification results.
      num_threads: Number of CPU threads to run the model.
      enable_edgetpu: Whether to run the model on EdgeTPU.
      camera_id: The camera id to be passed to OpenCV.
      width: The width of the frame captured from the camera.
      height: The height of the frame captured from the camera.
  """

  # Initialize the image classification model
  base_options = core.BaseOptions(
      file_name=model, use_coral=enable_edgetpu, num_threads=num_threads)

  # Enable Coral by this setting
  classification_options = processor.ClassificationOptions(
      max_results=max_results, score_threshold=score_threshold)
  options = vision.ImageClassifierOptions(
      base_options=base_options, classification_options=classification_options)

  classifier = vision.ImageClassifier.create_from_options(options)

  # Variables to calculate FPS
  counter, fps = 0, 0
  start_time = time.time()

  # Start capturing video input from the camera
  cap = cv2.VideoCapture(camera_id)
  cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
  cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

  # Continuously capture images from the camera and run inference
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      sys.exit(
          'ERROR: Unable to read from webcam. Please verify your webcam settings.'
      )

    counter += 1
    image = cv2.flip(image, 1)

    # Convert the image from BGR to RGB as required by the TFLite model.
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Create TensorImage from the RGB image
    tensor_image = vision.TensorImage.create_from_array(rgb_image)
    # List classification results
    categories = classifier.classify(tensor_image)

    # Show classification results on the image
    for idx, category in enumerate(categories.classifications[0].classes):
      class_name = category.class_name
      score = round(category.score, 2)
      result_text = class_name + ' (' + str(score) + ')'
      text_location = (_LEFT_MARGIN, (idx + 2) * _ROW_SIZE)
      cv2.putText(image, result_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                  _FONT_SIZE, _TEXT_COLOR, _FONT_THICKNESS)

    # Calculate the FPS
    if counter % _FPS_AVERAGE_FRAME_COUNT == 0:
      end_time = time.time()
      fps = _FPS_AVERAGE_FRAME_COUNT / (end_time - start_time)
      start_time = time.time()

    # Show the FPS
    fps_text = 'FPS = ' + str(int(fps))
    text_location = (_LEFT_MARGIN, _ROW_SIZE)
    cv2.putText(image, fps_text, text_location, cv2.FONT_HERSHEY_PLAIN,
```

```
                _FONT_SIZE, _TEXT_COLOR, _FONT_THICKNESS)

    # Stop the program if the ESC key is pressed.
    if cv2.waitKey(1) == 27:
      break
    cv2.imshow('image_classification', image)

  cap.release()
  cv2.destroyAllWindows()


def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--model',
      help='Name of image classification model.',
      required=False,
      default='efficientnet_lite0.tflite')
  parser.add_argument(
      '--maxResults',
      help='Max of classification results.',
      required=False,
      default=3)
  parser.add_argument(
      '--scoreThreshold',
      help='The score threshold of classification results.',
      required=False,
      type=float,
      default=0.0)
  parser.add_argument(
      '--numThreads',
      help='Number of CPU threads to run the model.',
      required=False,
      default=4)
  parser.add_argument(
      '--enableEdgeTPU',
      help='Whether to run the model on EdgeTPU.',
      action='store_true',
      required=False,
      default=False)
  parser.add_argument(
      '--cameraId', help='Id of camera.', required=False, default=0)
  parser.add_argument(
      '--frameWidth',
      help='Width of frame to capture from camera.',
      required=False,
      default=640)
  parser.add_argument(
      '--frameHeight',
      help='Height of frame to capture from camera.',
      required=False,
      default=480)
  args = parser.parse_args()

  run(args.model, int(args.maxResults),
      args.scoreThreshold, int(args.numThreads), bool(args.enableEdgeTPU),
      int(args.cameraId), args.frameWidth, args.frameHeight)


if __name__ == '__main__':
  main()
```

To do:

Test the above code with default arguments.

Change the values of: **`--maxResults`**, **`--numThreads`**, **`--frameWidth`**, **`--frameHeight`**

1.2.2 Objects Detection

First go to the following directory:

```
cd examples/lite/examples/object_detection/raspberry_pi
```

Then run the script required dependencies and download the **TFLite** models.

```
sh setup.sh
```

1.2.2.1 Run the example

```
python3 detect.py \
  --model efficientdet_lite0.tflite
```

You should see the camera feed appear on the monitor attached to your Raspberry Pi. Put some objects in front of the camera, like a coffee mug or keyboard, and you'll see **boxes drawn around** those that the model recognizes, including the **label and score** for each. It also prints the number of frames per second (FPS) at the top-left corner of the screen.

As the pipeline contains some processes other than model inference, including visualizing the detection results, you can expect a higher FPS if your inference pipeline runs in headless mode without visualization.

1.2.2.2 Complete code

```
"""Main script to run the object detection routine."""
import argparse
import sys
import time

import cv2
from tflite_support.task import core
from tflite_support.task import processor
from tflite_support.task import vision
import utils


def run(model: str, camera_id: int, width: int, height: int, num_threads: int,
        enable_edgetpu: bool) -> None:
  """Continuously run inference on images acquired from the camera.

  Args:
    model: Name of the TFLite object detection model.
    camera_id: The camera id to be passed to OpenCV.
    width: The width of the frame captured from the camera.
    height: The height of the frame captured from the camera.
    num_threads: The number of CPU threads to run the model.
    enable_edgetpu: True/False whether the model is a EdgeTPU model.
  """

  # Variables to calculate FPS
  counter, fps = 0, 0
  start_time = time.time()

  # Start capturing video input from the camera
  cap = cv2.VideoCapture(camera_id)
  cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
  cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

  # Visualization parameters
  row_size = 20  # pixels
  left_margin = 24  # pixels
  text_color = (0, 0, 255)  # red
  font_size = 1
  font_thickness = 1
  fps_avg_frame_count = 10

  # Initialize the object detection model
  base_options = core.BaseOptions(
```

```
      file_name=model, use_coral=enable_edgetpu, num_threads=num_threads)
  detection_options = processor.DetectionOptions(
      max_results=3, score_threshold=0.3)
  options = vision.ObjectDetectorOptions(
      base_options=base_options, detection_options=detection_options)
  detector = vision.ObjectDetector.create_from_options(options)

  # Continuously capture images from the camera and run inference
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      sys.exit(
          'ERROR: Unable to read from webcam. Please verify your webcam settings.'
      )

    counter += 1
    image = cv2.flip(image, 1)

    # Convert the image from BGR to RGB as required by the TFLite model.
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Create a TensorImage object from the RGB image.
    input_tensor = vision.TensorImage.create_from_array(rgb_image)

    # Run object detection estimation using the model.
    detection_result = detector.detect(input_tensor)

    # Draw keypoints and edges on input image
    image = utils.visualize(image, detection_result)

    # Calculate the FPS
    if counter % fps_avg_frame_count == 0:
      end_time = time.time()
      fps = fps_avg_frame_count / (end_time - start_time)
      start_time = time.time()

    # Show the FPS
    fps_text = 'FPS = {:.1f}'.format(fps)
    text_location = (left_margin, row_size)
    cv2.putText(image, fps_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                font_size, text_color, font_thickness)

    # Stop the program if the ESC key is pressed.
    if cv2.waitKey(1) == 27:
      break
    cv2.imshow('object_detector', image)

  cap.release()
  cv2.destroyAllWindows()


def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--model',
      help='Path of the object detection model.',
      required=False,
      default='efficientdet_lite0.tflite')
  parser.add_argument(
      '--cameraId', help='Id of camera.', required=False, type=int, default=0)
  parser.add_argument(
      '--frameWidth',
      help='Width of frame to capture from camera.',
      required=False,
      type=int,
      default=640)
  parser.add_argument(
      '--frameHeight',
      help='Height of frame to capture from camera.',
      required=False,
      type=int,
      default=480)
  parser.add_argument(
      '--numThreads',
      help='Number of CPU threads to run the model.',
```

```
      required=False,
      type=int,
      default=4)
  parser.add_argument(
      '--enableEdgeTPU',
      help='Whether to run the model on EdgeTPU.',
      action='store_true',
      required=False,
      default=False)
  args = parser.parse_args()

  run(args.model, int(args.cameraId), args.frameWidth, args.frameHeight,
      int(args.numThreads), bool(args.enableEdgeTPU))


if __name__ == '__main__':
  main()
```

To do:

Test the above code with default arguments.

Change the values of: **`--maxResults`**, **`--numThreads`**, **`--frameWidth`**, **`--frameHeight`**

1.2.3 Pose estimation

First go to the following directory:

```
/home/pi/examples/lite/examples/pose_estimation
```

See this blog post (**https://medium.com/roonyx/pose-estimation-and-matching-with-tensorflow-lite-posenet-model-ea2e9249abbd**) for a full guide on doing pose estimation and classification using TensorFlow Lite.

- **Pose estimation**: Detect keypoints, such as eye, ear, arm etc., from an input image.
    - Input: An image
    - Output: A list of keypoint coordinates and confidence score.

- **Pose classification**: Classify a human pose into predefined classes, such as different yoga poses. Pose classification internally use pose estimation to detect the keypoints, and use the keypoints to classify the pose.
    - Input: An image
    - Output: A list of predefined classes and their confidence score.

This sample can run on Raspberry Pi or any computer that has a camera. It uses **OpenCV** to capture images from the camera and TensorFlow Lite to run inference on the input image.

1.2.3.1 Install the dependencies

Run this script to install the Python dependencies, and download the TFLite models.

```
sh setup.sh
```

1.2.3.2 Run the pose estimation sample

Use this command to run the pose estimation sample using the default **movenet_lightning** model.

```
python3 pose_estimation.py
```

You can optionally specify the **model_name** parameter to try other pose estimation models:
Use values:

Single-pose: **posenet, movenet_lightning**, **movenet_thunder**

Multi-poses: **movenet_multipose**

The default value is **movenet_lightning**.

```
python3 pose_estimation.py --model_name movenet_thunder
```

1.2.3.3 Run the pose classification sample

Use this command to run the pose estimation sample using the default **movenet_lightning** pose estimation model and the **classifier.tflite** yoga pose classification model.

```
python3 pose_estimation.py \
    --classifier classifier --label_file labels.txt
```

If you want to train a custom pose classification model, check out this tutorial.

1.2.3.4 Complete code

```
"""Main script to run pose classification and pose estimation."""
import argparse
import logging
import sys
import time
```

```
import cv2
from ml import Classifier
from ml import Movenet
from ml import MoveNetMultiPose
from ml import Posenet
import utils


def run(estimation_model: str, tracker_type: str, classification_model: str,
        label_file: str, camera_id: int, width: int, height: int) -> None:
  """Continuously run inference on images acquired from the camera.

  Args:
    estimation_model: Name of the TFLite pose estimation model.
    tracker_type: Type of Tracker('keypoint' or 'bounding_box').
    classification_model: Name of the TFLite pose classification model.
      (Optional)
    label_file: Path to the label file for the pose classification model. Class
      names are listed one name per line, in the same order as in the
      classification model output. See an example in the yoga_labels.txt file.
    camera_id: The camera id to be passed to OpenCV.
    width: The width of the frame captured from the camera.
    height: The height of the frame captured from the camera.
  """

  # Notify users that tracker is only enabled for MoveNet MultiPose model.
  if tracker_type and (estimation_model != 'movenet_multipose'):
    logging.warning(
        'No tracker will be used as tracker can only be enabled for '
        'MoveNet MultiPose model.')

  # Initialize the pose estimator selected.
  if estimation_model in ['movenet_lightning', 'movenet_thunder']:
    pose_detector = Movenet(estimation_model)
  elif estimation_model == 'posenet':
    pose_detector = Posenet(estimation_model)
  elif estimation_model == 'movenet_multipose':
    pose_detector = MoveNetMultiPose(estimation_model, tracker_type)
  else:
    sys.exit('ERROR: Model is not supported.')

  # Variables to calculate FPS
  counter, fps = 0, 0
  start_time = time.time()

  # Start capturing video input from the camera
  cap = cv2.VideoCapture(camera_id)
  cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
  cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

  # Visualization parameters
  row_size = 20  # pixels
  left_margin = 24  # pixels
  text_color = (0, 0, 255)  # red
  font_size = 1
  font_thickness = 1
  classification_results_to_show = 3
  fps_avg_frame_count = 10
  keypoint_detection_threshold_for_classifier = 0.1
  classifier = None

  # Initialize the classification model
  if classification_model:
    classifier = Classifier(classification_model, label_file)
    classification_results_to_show = min(classification_results_to_show,
                                         len(classifier.pose_class_names))

  # Continuously capture images from the camera and run inference
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      sys.exit(
          'ERROR: Unable to read from webcam. Please verify your webcam settings.'
      )

    counter += 1
```

```
    image = cv2.flip(image, 1)

    if estimation_model == 'movenet_multipose':
      # Run pose estimation using a MultiPose model.
      list_persons = pose_detector.detect(image)
    else:
      # Run pose estimation using a SinglePose model, and wrap the result in an
      # array.
      list_persons = [pose_detector.detect(image)]

    # Draw keypoints and edges on input image
    image = utils.visualize(image, list_persons)

    if classifier:
      # Check if all keypoints are detected before running the classifier.
      # If there's a keypoint below the threshold, show an error.
      person = list_persons[0]
      min_score = min([keypoint.score for keypoint in person.keypoints])
      if min_score < keypoint_detection_threshold_for_classifier:
        error_text = 'Some keypoints are not detected.'
        text_location = (left_margin, 2 * row_size)
        cv2.putText(image, error_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                    font_size, text_color, font_thickness)
        error_text = 'Make sure the person is fully visible in the camera.'
        text_location = (left_margin, 3 * row_size)
        cv2.putText(image, error_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                    font_size, text_color, font_thickness)
      else:
        # Run pose classification
        prob_list = classifier.classify_pose(person)

        # Show classification results on the image
        for i in range(classification_results_to_show):
          class_name = prob_list[i].label
          probability = round(prob_list[i].score, 2)
          result_text = class_name + ' (' + str(probability) + ')'
          text_location = (left_margin, (i + 2) * row_size)
          cv2.putText(image, result_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                      font_size, text_color, font_thickness)

    # Calculate the FPS
    if counter % fps_avg_frame_count == 0:
      end_time = time.time()
      fps = fps_avg_frame_count / (end_time - start_time)
      start_time = time.time()

    # Show the FPS
    fps_text = 'FPS = ' + str(int(fps))
    text_location = (left_margin, row_size)
    cv2.putText(image, fps_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                font_size, text_color, font_thickness)

    # Stop the program if the ESC key is pressed.
    if cv2.waitKey(1) == 27:
      break
    cv2.imshow(estimation_model, image)

  cap.release()
  cv2.destroyAllWindows()


def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--model',
      help='Name of estimation model.',
      required=False,
      default='movenet_lightning')
  parser.add_argument(
      '--tracker',
      help='Type of tracker to track poses across frames.',
      required=False,
      default='bounding_box')
  parser.add_argument(
      '--classifier', help='Name of classification model.', required=False)
```

```
  parser.add_argument(
      '--label_file',
      help='Label file for classification.',
      required=False,
      default='labels.txt')
  parser.add_argument(
      '--cameraId', help='Id of camera.', required=False, default=0)
  parser.add_argument(
      '--frameWidth',
      help='Width of frame to capture from camera.',
      required=False,
      default=640)
  parser.add_argument(
      '--frameHeight',
      help='Height of frame to capture from camera.',
      required=False,
      default=480)
  args = parser.parse_args()

  run(args.model, args.tracker, args.classifier, args.label_file,
      int(args.cameraId), args.frameWidth, args.frameHeight)


if __name__ == '__main__':
  main()
```

1.2.3.5 Customization options

- Here is the full list of parameters supported by the sample: **`python3 pose_classification.py`**
- **`model`**: Name of the TFLite pose estimation model to be used.
  * One of these values: **`posenet, movenet_lightning, movenet_thunder, movenet_multipose`**
  * Default value is **`movenet_lightning`**.
- **`tracker`**: Type of tracker to track poses across frames.
  * One of these values: **`bounding_box, keypoint`** - Only supported in multi-poses models.
  * Default value is **`bounding_box`**.
- **`classifier`**: Name of the TFLite pose classification model to be used.
  * Default value is **empty**. * If no classification model specified, the sample will only run the pose estimation step.
- **`camera_id`**: Specify the camera for OpenCV to capture images from.
  * Default value is **`0`**. **`/dev/video0`**
- **`frameWidth, frameHeight`**: Resolution of the image to be captured from the camera.
  * Default value is **`(640, 480)`**.

1.2.3.6 Visualize pose estimation result of test data

Run this script to visualize the pose estimation on test data:

```
python3 visualizer.py
```

1.2.3.7 Complete code

```
"""Script to run visualize pose estimation on test data."""
import argparse
import logging

import cv2
from data import BodyPart
from data import person_from_keypoints_with_scores
from ml import Movenet
from ml import Posenet
import numpy as np
import pandas as pd
import utils

_MODEL_POSENET = 'posenet'
_MODEL_LIGHTNING = 'movenet_lightning'
```

```
_MODEL_THUNDER = 'movenet_thunder'
_GROUND_TRUTH_CSV = 'test_data/pose_landmark_truth.csv'
_TEST_IMAGE_PATHS = ['test_data/image1.png', 'test_data/image2.jpeg']

# Load test images
_TEST_IMAGES = [cv2.imread(path) for path in _TEST_IMAGE_PATHS]

# Load pose estimation models
_POSENET = Posenet(_MODEL_POSENET)
_MOVENET_LIGHTNING = Movenet(_MODEL_LIGHTNING)
_MOVENET_THUNDER = Movenet(_MODEL_THUNDER)

# Load pose landmarks truth
_POSE_LANDMARKS_TRUTH = pd.read_csv(_GROUND_TRUTH_CSV)
_KEYPOINTS_TRUTH_LIST = [
    row.to_numpy().reshape((17, 2)) for row in _POSE_LANDMARKS_TRUTH.iloc
]


def _visualize_detection_result(input_image, ground_truth):
  """Visualize the pose estimation result and write the output image to a file.

  The detected keypoints follow these color codes:
      * PoseNet: blue
      * MoveNet Lightning: red
      * MoveNet Thunder: yellow
      * Ground truth (from CSV): green
  Note: This test is meant to be run by a human who want to visually verify
  the pose estimation result.

  Args:
    input_image: Numpy array of shape (height, width, 3)
    ground_truth: Numpy array with absolute coordinates of the keypoints to be
      plotted.

  Returns:
    Input image with pose estimation results.
  """
  output_image = input_image.copy()

  # Draw detection result from Posenet (blue)
  person = _POSENET.detect(input_image)
  output_image = utils.visualize(output_image, [person], (255, 0, 0))

  # Draw detection result from Movenet Lightning (red)
  person = _MOVENET_LIGHTNING.detect(input_image, reset_crop_region=True)
  output_image = utils.visualize(output_image, [person], (0, 0, 255))

  # Draw detection result from Movenet Thunder (yellow)
  person = _MOVENET_THUNDER.detect(input_image, reset_crop_region=True)
  output_image = utils.visualize(output_image, [person], (0, 255, 255))

  # Create a fake score column to convert ground truth to "Person" type
  ground_truth[:, :2] = ground_truth[:, 1::-1]
  score = np.ones((17, 1), dtype=float)
  ground_truth = np.append(ground_truth, score, axis=1)
  person = person_from_keypoints_with_scores(ground_truth, 1, 1)

  # Draw ground truth detection result (green)
  output_image = utils.visualize(output_image, [person], (0, 255, 0))

  return output_image


def _create_ground_truth_csv(input_images, ground_truth_csv_path):
  """Create ground truth CSV file from the given input images.

  Args:
    input_images: An array of input RGB images (height, width, 3).
    ground_truth_csv_path: path to the output CSV.
  """
  # Create column name for CSV file
  column_names = []
  for body_part in BodyPart:
    column_names.append(body_part.name + '_x')
    column_names.append(body_part.name + '_y')
```

```
  # Create ground truth data by feeding the test images through MoveNet
  # Thunder 3 times to leverage the cropping logic and improve accuracy.
  keypoints_data = []
  for input_image in input_images:
    person = _MOVENET_THUNDER.detect(input_image, reset_crop_region=True)
    for _ in range(3):
      person = _MOVENET_THUNDER.detect(input_image, reset_crop_region=False)

    kpts = []
    keypoints = person.keypoints
    for idx in range(len(keypoints)):
      kpts.extend((keypoints[idx].coordinate.x, keypoints[idx].coordinate.y))

    # Store kpts into keypoints_data
    keypoints_data.append(kpts)

  # Write ground truth CSV file
  keypoints_df = pd.DataFrame(keypoints_data, columns=column_names)
  keypoints_df.to_csv(ground_truth_csv_path, index=False)


def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--ground_truth_csv_output',
      help='Path to generate ground truth CSV file. (Optional)',
      required=False)
  args = parser.parse_args()

  # Create ground truth CSV if the ground_truth_csv parameter is set
  if args.ground_truth_csv_output:
    _create_ground_truth_csv(_TEST_IMAGES, args.ground_truth_csv_output)
    logging.info('Created ground truth keypoint CSV: %s',
                 args.ground_truth_csv_output)

  # Visualize detection result of the test images
  for index in range(len(_TEST_IMAGES)):
    test_image_path = _TEST_IMAGE_PATHS[index]
    test_image = _TEST_IMAGES[index]
    keypoint_truth = _KEYPOINTS_TRUTH_LIST[index]
    visualized_image = _visualize_detection_result(test_image, keypoint_truth)
    cv2.imshow(test_image_path, visualized_image)

  cv2.waitKey(0)
  cv2.destroyAllWindows()


if __name__ == '__main__':
  main()
```

1.2.4 Segmentation

Go to the directory and use our script to install a couple Python packages, and download the `Deeplabv3` model:

```
cd examples/lite/examples/image_segmentation/raspberry_pi
```

Run the script to install the required dependencies and download the TFLite models.

```
sh setup.sh
```

1.2.4.1 Run the example

```
python3 segment.py
```

- You can optionally specify the `model` parameter to set the **TensorFlow Lite** model to be used:
  - The default value is **`deeplabv3.tflite`**
  - Image segmentation models from TensorFlow Hub **with metadata** are supported.
- You can optionally specify the **`displayMode`** parameter to change how the segmentation result is displayed:
  - Use values: **`overlay, side-by-side`**.
  - The default value is **`overlay`**.
- Example usage:

```
python3 main.py
  --model somemodel.tflite
  --displayMode side-by-side
```

1.2.4.2 Complete code

```
"""Main script to run image segmentation."""

import argparse
import sys
import time
from typing import List

import cv2
import numpy as np
from tflite_support.task import core
from tflite_support.task import processor
from tflite_support.task import vision
import utils

# Visualization parameters
_FPS_AVERAGE_FRAME_COUNT = 10
_FPS_LEFT_MARGIN = 24  # pixels
_LEGEND_TEXT_COLOR = (0, 0, 255)  # red
_LEGEND_BACKGROUND_COLOR = (255, 255, 255)  # white
_LEGEND_FONT_SIZE = 1
_LEGEND_FONT_THICKNESS = 1
_LEGEND_ROW_SIZE = 20  # pixels
_LEGEND_RECT_SIZE = 16  # pixels
_LABEL_MARGIN = 10
_OVERLAY_ALPHA = 0.5
_PADDING_WIDTH_FOR_LEGEND = 150  # pixels


def run(model: str, display_mode: str, num_threads: int, enable_edgetpu: bool,
        camera_id: int, width: int, height: int) -> None:
  """Continuously run inference on images acquired from the camera.

  Args:
    model: Name of the TFLite image segmentation model.
```

```
    display_mode: Name of mode to display image segmentation.
    num_threads: Number of CPU threads to run the model.
    enable_edgetpu: Whether to run the model on EdgeTPU.
    camera_id: The camera id to be passed to OpenCV.
    width: The width of the frame captured from the camera.
    height: The height of the frame captured from the camera.
  """

  # Initialize the image segmentation model.
  base_options = core.BaseOptions(
      file_name=model, use_coral=enable_edgetpu, num_threads=num_threads)
  segmentation_options = processor.SegmentationOptions(
      output_type=processor.SegmentationOptions.OutputType.CATEGORY_MASK)
  options = vision.ImageSegmenterOptions(
      base_options=base_options, segmentation_options=segmentation_options)

  segmenter = vision.ImageSegmenter.create_from_options(options)

  # Variables to calculate FPS
  counter, fps = 0, 0
  start_time = time.time()

  # Start capturing video input from the camera
  cap = cv2.VideoCapture(camera_id)
  cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
  cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

  # Continuously capture images from the camera and run inference.
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      sys.exit(
          'ERROR: Unable to read from webcam. Please verify your webcam settings.'
      )

    counter += 1
    image = cv2.flip(image, 1)

    # Convert the image from BGR to RGB as required by the TFLite model.
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Create TensorImage from the RGB image
    tensor_image = vision.TensorImage.create_from_array(rgb_image)
    # Segment with each frame from camera.
    segmentation_result = segmenter.segment(tensor_image)

    # Convert the segmentation result into an image.
    seg_map_img, found_colored_labels = utils.segmentation_map_to_image(
        segmentation_result)

    # Resize the segmentation mask to be the same shape as input image.
    seg_map_img = cv2.resize(
        seg_map_img,
        dsize=(image.shape[1], image.shape[0]),
        interpolation=cv2.INTER_NEAREST)

    # Visualize segmentation result on image.
    overlay = visualize(image, seg_map_img, display_mode, fps,
                        found_colored_labels)

    # Calculate the FPS
    if counter % _FPS_AVERAGE_FRAME_COUNT == 0:
      end_time = time.time()
      fps = _FPS_AVERAGE_FRAME_COUNT / (end_time - start_time)
      start_time = time.time()

    # Stop the program if the ESC key is pressed.
    if cv2.waitKey(1) == 27:
      break
    cv2.imshow('image_segmentation', overlay)

  cap.release()
  cv2.destroyAllWindows()


def visualize(
```

```
    input_image: np.ndarray, segmentation_map_image: np.ndarray,
    display_mode: str, fps: float,
    colored_labels: List[processor.Segmentation.ColoredLabel]) -> np.ndarray:
  """Visualize segmentation result on image.

  Args:
    input_image: The [height, width, 3] RGB input image.
    segmentation_map_image: The [height, width, 3] RGB segmentation map image.
    display_mode: How the segmentation map should be shown. 'overlay' or
      'side-by-side'.
    fps: Value of fps.
    colored_labels: List of colored labels found in the segmentation result.

  Returns:
    Input image overlaid with segmentation result.
  """
  # Show the input image and the segmentation map image.
  if display_mode == 'overlay':
    # Overlay mode.
    overlay = cv2.addWeighted(input_image, _OVERLAY_ALPHA,
                              segmentation_map_image, _OVERLAY_ALPHA, 0)
  elif display_mode == 'side-by-side':
    # Side by side mode.
    overlay = cv2.hconcat([input_image, segmentation_map_image])
  else:
    sys.exit(f'ERROR: Unsupported display mode: {display_mode}.')

  # Show the FPS
  fps_text = 'FPS = ' + str(int(fps))
  text_location = (_FPS_LEFT_MARGIN, _LEGEND_ROW_SIZE)
  cv2.putText(overlay, fps_text, text_location, cv2.FONT_HERSHEY_PLAIN,
              _LEGEND_FONT_SIZE, _LEGEND_TEXT_COLOR, _LEGEND_FONT_THICKNESS)

  # Initialize the origin coordinates of the label.
  legend_x = overlay.shape[1] + _LABEL_MARGIN
  legend_y = overlay.shape[0] // _LEGEND_ROW_SIZE + _LABEL_MARGIN

  # Expand the frame to show the label.
  overlay = cv2.copyMakeBorder(overlay, 0, 0, 0, _PADDING_WIDTH_FOR_LEGEND,
                               cv2.BORDER_CONSTANT, None,
                               _LEGEND_BACKGROUND_COLOR)

  # Show the label on right-side frame.
  for colored_label in colored_labels:
    rect_color = (colored_label.r, colored_label.g, colored_label.b)
    start_point = (legend_x, legend_y)
    end_point = (legend_x + _LEGEND_RECT_SIZE, legend_y + _LEGEND_RECT_SIZE)
    cv2.rectangle(overlay, start_point, end_point, rect_color,
                  -_LEGEND_FONT_THICKNESS)

    label_location = legend_x + _LEGEND_RECT_SIZE + _LABEL_MARGIN, legend_y + _LABEL_MARGIN
    cv2.putText(overlay, colored_label.class_name, label_location,
                cv2.FONT_HERSHEY_PLAIN, _LEGEND_FONT_SIZE, _LEGEND_TEXT_COLOR,
                _LEGEND_FONT_THICKNESS)
    legend_y += (_LEGEND_RECT_SIZE + _LABEL_MARGIN)

  return overlay


def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--model',
      help='Name of image segmentation model.',
      required=False,
      default='deeplabv3.tflite')
  parser.add_argument(
      '--displayMode',
      help='Mode to display image segmentation.',
      required=False,
      default='overlay')
  parser.add_argument(
      '--numThreads',
      help='Number of CPU threads to run the model.',
      required=False,
```

```
      default=4)
  parser.add_argument(
      '--enableEdgeTPU',
      help='Whether to run the model on EdgeTPU.',
      action='store_true',
      required=False,
      default=False)
  parser.add_argument(
      '--cameraId', help='Id of camera.', required=False, default=0)
  parser.add_argument(
      '--frameWidth',
      help='Width of frame to capture from camera.',
      required=False,
      default=640)
  parser.add_argument(
      '--frameHeight',
      help='Height of frame to capture from camera.',
      required=False,
      default=480)
  args = parser.parse_args()

  run(args.model, args.displayMode, int(args.numThreads),
      bool(args.enableEdgeTPU), int(args.cameraId), args.frameWidth,
      args.frameHeight)


if __name__ == '__main__':
  main()
```

1.2.5 Video classification

First go to the following directory:

```
cd examples/lite/examples/video_classification/raspberry_pi
```

Then run the script required dependencies and download the **TFLite** models.

```
sh setup.sh
```

1.2.5.1 Run the example

```
python3 classify.py
```

- You can optionally specify the model parameter to set the TensorFlow Lite model to be used:
  - The default value is **`movinet_a0_int8.tflite`**
  - This sample currently uses **`MoviNet-A0`**, but it supports all TensorFlow Lite MoviNet video classification models that are available on TensorFlow Hub. You can use the larger variant of **`MoviNet`** if you need higher accuracy.
- You can optionally specify the **`maxResults`** parameter to limit the list of classification results:
  - Supported value: A positive integer.
  - Default value: 3.
- Example usage:

  ```
  python3 classify.py --model movinet_a0_int8.tflite --maxResults 5
  ```

1.2.5.2 Complete code

```
"""Main script to run video classification."""

import argparse
import sys
import time

import cv2

from video_classifier import VideoClassifier
from video_classifier import VideoClassifierOptions

# Visualization parameters
_ROW_SIZE = 20  # pixels
_LEFT_MARGIN = 24  # pixels
_TEXT_COLOR = (0, 0, 255)  # red  - you may change it to black !
_FONT_SIZE = 1
_FONT_THICKNESS = 1
_MODEL_FPS = 5  # Ensure the input images are fed to the model at this fps.
_MODEL_FPS_ERROR_RANGE = 0.1  # Acceptable error range in fps.


def run(model: str, label: str, max_results: int, num_threads: int,
        camera_id: int, width: int, height: int) -> None:
  """Continuously run inference on images acquired from the camera.

  Args:
      model: Name of the TFLite video classification model.
      label: Name of the video classification label.
      max_results: Max of classification results.
      num_threads: Number of CPU threads to run the model.
      camera_id: The camera id to be passed to OpenCV.
      width: The width of the frame captured from the camera.
      height: The height of the frame captured from the camera.
  """
  # Initialize the video classification model
  options = VideoClassifierOptions(
      num_threads=num_threads, max_results=max_results)
  classifier = VideoClassifier(model, label, options)

  # Variables to calculate FPS
  counter, fps, last_inference_start_time, time_per_infer = 0, 0, 0, 0
```

```
  categories = []

  # Start capturing video input from the camera
  cap = cv2.VideoCapture(camera_id)
  cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
  cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

  # Continuously capture images from the camera and run inference
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      sys.exit(
          'ERROR: Unable to read from webcam. Please verify your webcam settings.'
      )
    counter += 1

    # Mirror the image
    image = cv2.flip(image, 1)

    # Ensure that frames are feed to the model at {_MODEL_FPS} frames per second
    # as required in the model specs.
    current_frame_start_time = time.time()
    diff = current_frame_start_time - last_inference_start_time
    if diff * _MODEL_FPS >= (1 - _MODEL_FPS_ERROR_RANGE):
      # Store the time when inference starts.
      last_inference_start_time = current_frame_start_time

      # Calculate the inference FPS
      fps = 1.0 / diff

      # Convert the frame to RGB as required by the TFLite model.
      frame_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

      # Feed the frame to the video classification model.
      categories = classifier.classify(frame_rgb)

      # Calculate time required per inference.
      time_per_infer = time.time() - current_frame_start_time

    # Notes: Frames that aren't fed to the model are still displayed to make the
    # video look smooth. We'll show classification results from the latest
    # classification run on the screen.
    # Show the FPS .
    fps_text = 'Current FPS = {0:.1f}. Expect: {1}'.format(fps, _MODEL_FPS)
    text_location = (_LEFT_MARGIN, _ROW_SIZE)
    cv2.putText(image, fps_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                _FONT_SIZE, _TEXT_COLOR, _FONT_THICKNESS)

    # Show the time per inference.
    time_per_infer_text = 'Time per inference: {0}ms'.format(
        int(time_per_infer * 1000))
    text_location = (_LEFT_MARGIN, _ROW_SIZE * 2)
    cv2.putText(image, time_per_infer_text, text_location,
                cv2.FONT_HERSHEY_PLAIN, _FONT_SIZE, _TEXT_COLOR,
                _FONT_THICKNESS)

    # Show classification results on the image.
    for idx, category in enumerate(categories):
      class_name = category.label
      probability = round(category.score, 2)
      result_text = class_name + ' (' + str(probability) + ')'
      # Skip the first 2 lines occupied by the fps and time per inference.
      text_location = (_LEFT_MARGIN, (idx + 3) * _ROW_SIZE)
      cv2.putText(image, result_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                  _FONT_SIZE, _TEXT_COLOR, _FONT_THICKNESS)

    # Stop the program if the ESC key is pressed.
    if cv2.waitKey(1) == 27:
      break
    cv2.imshow('video_classification', image)

  cap.release()
  cv2.destroyAllWindows()


def main():
```

```
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--model',
      help='Name of video classification model.',
      required=False,
      default='movinet_a0_int8.tflite')
  parser.add_argument(
      '--label',
      help='Name of video classification label.',
      required=False,
      default='kinetics600_label_map.txt')
  parser.add_argument(
      '--maxResults',
      help='Max of classification results.',
      required=False,
      default=3)
  parser.add_argument(
      '--numThreads',
      help='Number of CPU threads to run the model.',
      required=False,
      default=4)
  parser.add_argument(
      '--cameraId', help='Id of camera.', required=False, default=0)
  parser.add_argument(
      '--frameWidth',
      help='Width of frame to capture from camera.',
      required=False,
      default=640)
  parser.add_argument(
      '--frameHeight',
      help='Height of frame to capture from camera.',
      required=False,
      default=480)
  args = parser.parse_args()

  run(args.model, args.label, int(args.maxResults), int(args.numThreads),
      int(args.cameraId), args.frameWidth, args.frameHeight)


if __name__ == '__main__':
  main()
```

To do:

Test the above code with default arguments, you may change the color of the text on the image.

Change the values of: **`--maxResults`**, **`--numThreads`**, **`--frameWidth`**, **`--frameHeight`**

1.2.6 Sound classification

First go to the following directory:

```
cd examples/lite/examples/sound_classification/raspberry_pi
```

Then run the script required dependencies and download the **TFLite** models.

```
sh setup.sh
```

Insert the **audio card** in your **GPIO** pins.

1.2.6.1 Install `Voicecard` software

Make sure you've got the **WaveShare HAT** or Voice Bonnet installed, and I2C support installed as well!
Note that the audio card may already be installed with your OS.

When you run

```
sudo i2cdetect -y 1
```

you should see an entry under **1A**, indicating the hardware sees the audio card. The number may also appear as UU if you already installed software

```
cd ~
sudo apt-get install -y git
git clone https://github.com/HinTak/seeed-voicecard
cd seeed-voicecard
git checkout v5.9
sudo ./install.sh
```

Reboot with

```
sudo reboot
```

and on reboot run

```
sudo aplay -l
```

To list all sound cards, you should see it at the bottom
If your card number differs from the above image, take note of your number.
You can use `alsamixer` to adjust the volume, dont forget to select the card with **F6**

1.2.6.1 Python Libraries

The Microphone and Voice Card are installed as Linux level devices, so using them in is done as you would with any system level audio device. If you would like to make use of audio in Python, you can use the PyAudio library. To install pyaudio and its dependencies, run the following code:

```
sudo pip3 install pyaudio
sudo apt-get install libportaudio2
```

1.2.6.2 Python Usage

Here is a basic test script to enumerate the devices and record for 10 seconds. When prompted, choose the device called **seeed-2mic-voicecard**.

Copy and paste the following code into a file called **audiotest.py.**

```
import pyaudio
import wave

FORMAT = pyaudio.paInt16
CHANNELS = 1           # Number of channels
BITRATE = 44100        # Audio Bitrate
CHUNK_SIZE = 512       # Chunk size to
RECORDING_LENGTH = 10  # Recording Length in seconds
WAVE_OUTPUT_FILENAME = "myrecording.wav"
audio = pyaudio.PyAudio()

info = audio.get_host_api_info_by_index(0)
numdevices = info.get('deviceCount')
for i in range(0, numdevices):
    if (audio.get_device_info_by_host_api_device_index(0, i).get('maxInputChannels')) > 0:
        print("Input Device id ", i, " - ", audio.get_device_info_by_host_api_device_index(0,
i).get('name'))

print("Which Input Device would you like to use?")
device_id = int(input()) # Choose a device
print("Recording using Input Device ID "+str(device_id))

stream = audio.open(
    format=FORMAT,
    channels=CHANNELS,
    rate=BITRATE,
    input=True,
    input_device_index = device_id,
    frames_per_buffer=CHUNK_SIZE
)

recording_frames = []

for i in range(int(BITRATE / CHUNK_SIZE * RECORDING_LENGTH)):
    data = stream.read(CHUNK_SIZE)
    recording_frames.append(data)

stream.stop_stream()
stream.close()
audio.terminate()

waveFile = wave.open(WAVE_OUTPUT_FILENAME, 'wb')
waveFile.setnchannels(CHANNELS)
waveFile.setsampwidth(audio.get_sample_size(FORMAT))
waveFile.setframerate(BITRATE)
waveFile.writeframes(b''.join(recording_frames))
waveFile.close()
```

Run the code with the following command:

```
sudo python3 audiotest.py
```

When finished, you can test playing back the file with the following command:

```
aplay recordedFile.wav
```

1.2.6.3 Run the example

```
python3 classify.py
```

- You can optionally specify the `model` parameter to set the TensorFlow Lite model to be used:
  - The default value is **`yamnet.tflite`**
- You can optionally specify the **`maxResults`** parameter to limit the list of classification results:
  - Supported value: A positive integer.
  - Default value: 5.

- Example usage:

```
python3 classify.py \
  --model yamnet.tflite \
  --maxResults 5
```

1.2.6.4 Complete code

```
"""Main scripts to run audio classification."""

import argparse
import time

from tflite_support.task import audio
from tflite_support.task import core
from tflite_support.task import processor
from utils import Plotter


def run(model: str, max_results: int, score_threshold: float,
        overlapping_factor: float, num_threads: int,
        enable_edgetpu: bool) -> None:
  """Continuously run inference on audio data acquired from the device.

  Args:
    model: Name of the TFLite audio classification model.
    max_results: Maximum number of classification results to display.
    score_threshold: The score threshold of classification results.
    overlapping_factor: Target overlapping between adjacent inferences.
    num_threads: Number of CPU threads to run the model.
    enable_edgetpu: Whether to run the model on EdgeTPU.
  """

  if (overlapping_factor <= 0) or (overlapping_factor >= 1.0):
    raise ValueError('Overlapping factor must be between 0 and 1.')

  if (score_threshold < 0) or (score_threshold > 1.0):
    raise ValueError('Score threshold must be between (inclusive) 0 and 1.')

  # Initialize the audio classification model.
  base_options = core.BaseOptions(
      file_name=model, use_coral=enable_edgetpu, num_threads=num_threads)
  classification_options = processor.ClassificationOptions(
      max_results=max_results, score_threshold=score_threshold)
  options = audio.AudioClassifierOptions(
      base_options=base_options, classification_options=classification_options)
  classifier = audio.AudioClassifier.create_from_options(options)

  # Initialize the audio recorder and a tensor to store the audio input.
  audio_record = classifier.create_audio_record()
  tensor_audio = classifier.create_input_tensor_audio()

  # We'll try to run inference every interval_between_inference seconds.
  # This is usually half of the model's input length to create an overlapping
  # between incoming audio segments to improve classification accuracy.
  input_length_in_second = float(len(
      tensor_audio.buffer)) / tensor_audio.format.sample_rate
  interval_between_inference = input_length_in_second * (1 - overlapping_factor)
  pause_time = interval_between_inference * 0.1
  last_inference_time = time.time()

  # Initialize a plotter instance to display the classification results.
  plotter = Plotter()

  # Start audio recording in the background.
  audio_record.start_recording()

  # Loop until the user close the classification results plot.
  while True:
    # Wait until at least interval_between_inference seconds has passed since
    # the last inference.
    now = time.time()
    diff = now - last_inference_time
    if diff < interval_between_inference:
```

```
      time.sleep(pause_time)
      continue
    last_inference_time = now

    # Load the input audio and run classify.
    tensor_audio.load_from_audio_record(audio_record)
    result = classifier.classify(tensor_audio)

    # Plot the classification results.
    plotter.plot(result)


def main():
  parser = argparse.ArgumentParser(
      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
  parser.add_argument(
      '--model',
      help='Name of the audio classification model.',
      required=False,
      default='yamnet.tflite')
  parser.add_argument(
      '--maxResults',
      help='Maximum number of results to show.',
      required=False,
      default=5)
  parser.add_argument(
      '--overlappingFactor',
      help='Target overlapping between adjacent inferences. Value must be in (0, 1)',
      required=False,
      default=0.5)
  parser.add_argument(
      '--scoreThreshold',
      help='The score threshold of classification results.',
      required=False,
      default=0.0)
  parser.add_argument(
      '--numThreads',
      help='Number of CPU threads to run the model.',
      required=False,
      default=4)
  parser.add_argument(
      '--enableEdgeTPU',
      help='Whether to run the model on EdgeTPU.',
      action='store_true',
      required=False,
      default=False)
  args = parser.parse_args()

  run(args.model, int(args.maxResults), float(args.scoreThreshold),
      float(args.overlappingFactor), int(args.numThreads),
      bool(args.enableEdgeTPU))


if __name__ == '__main__':
  main()
```

Lab 2

TensorFlow Lite Speech Recognition

Simple audio recognition: Recognizing keywords on a Raspberry Pi using Machine Learning (I2S, TensorFlow Lite)

2.0 Introduction

2.0.1 Objective

Adapt the official TensorFlow simple audio recognition example to use live audio data from an I2S microphone on a Raspberry Pi.

In this lab we will show you how to build a basic speech recognition network that recognizes ten different words. It's important to know that real speech and audio recognition systems are much more complex, but like MNIST for images, it should give you a basic understanding of the techniques involved. Once you've completed this lab, you'll have a model that tries to classify a one second audio clip as "down", "go", "left", "no", "right", "stop", "up" and "yes".
(**https://github.com/mkvenkit/simple_audio_pi**)

2.0.2 The Method

Here's our plan:

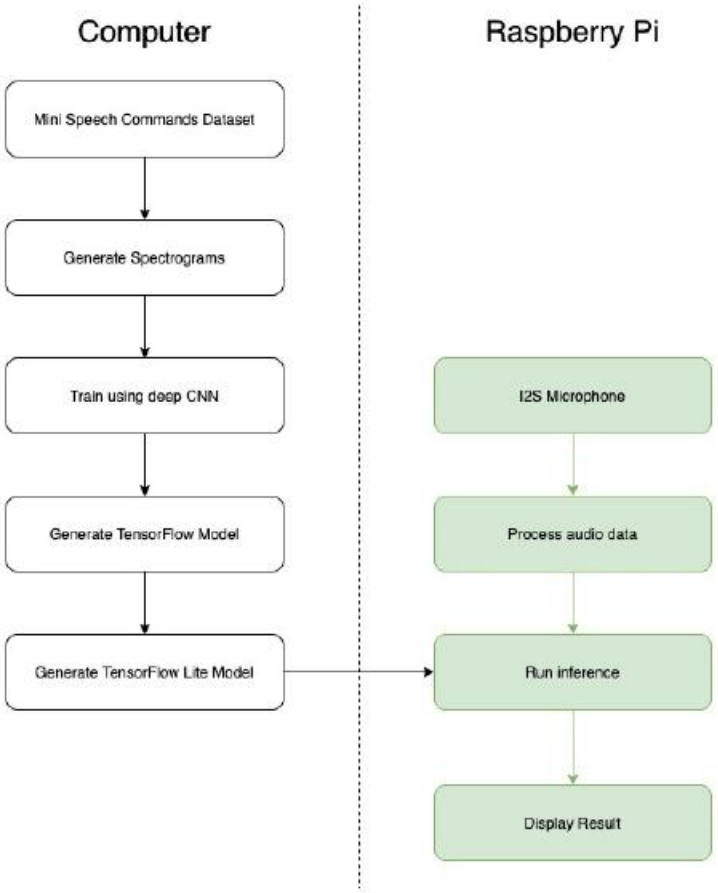

Fig 2.1 The method for training (TensorFlow) and execution (TensorFlow Lite) of the audio command model

2.0.3 Project Architecture

As outlined in the figure above, we will proceed as follows:

1. Train a model using a subset of the speech command data set from MNIST. The set of eight commands we train for are: “go”, “down”, “up”, “stop”, “yes”, “left”, “right”, and “no”. This was already being done by the original example, but since we need to run inference on it using TensorFlow Lite, we need to make some modifications. More on this later.
2. Convert the trained model to a TensorFlow Lite model.
3. Install the TensorFlow Lite interpreter on the Raspberry Pi.
4. Set up an I2S microphone on the Raspberry Pi to collect live audio data.
5. Scale and structure the audio data appropriately and run inference on it using the Lite model.
6. Display results on an OLED I2C display.

2.1 Training the Model

In this section we are going to study the steps required to train the **TensoFlow** model for audio command recognition. We will only discuss the most relevant parts of the **Jupyter** notebook presented in next section.

An important thing to be aware of during the training phase of your ML project is **the shape of your dataset tensors**. Getting the shapes wrong will give you a lot of misery when working with TensorFlow.

In our case, the input data consists of **8000 audio files in `.wav` format**. Each of them is sampled at **16000 Hz** and have a **length of less than or equal to 1 second**. So the first order of business is to **read these files**, **extract the audio data**, and **pad** them to make them all equal **to 1 second**. In this process, the data is also **normalized** to `[-1, 1]`. This is what the input data typically looks like after these manipulations:

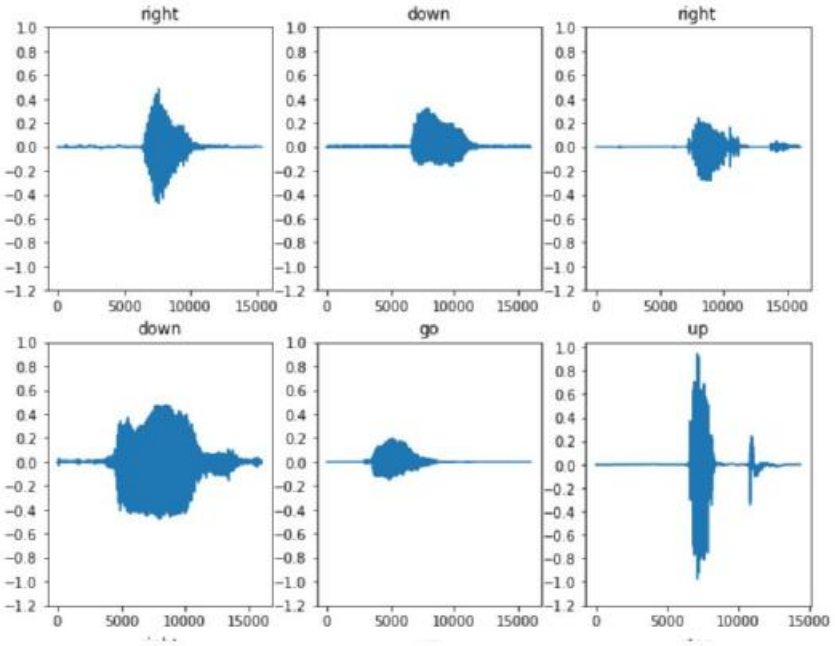

Fig 2.2 Audio data for the audio command model

Now **we’re not going** to feed this data directly into a model. We’re going to create **spectrograms** from it. Why? Because a spectrogram captures how the **signal frequencies changes** over time as the command is spoken.

This gives use features to train for, which will help in identifying the command. To compute the spectrograms, we will use **STFT** or the **short-time Fourier transform**. This is where our code deviates from the official example. See the snippet of code below:

```
def stft(x):
    f, t, spec = signal.stft(x.numpy(), fs=16000, nperseg=255, noverlap = 124, nfft=256)
    return tf.convert_to_tensor(np.abs(spec))


def get_spectrogram(waveform):
```

```
# Padding for files with less than 16000 samples
zero_padding = tf.zeros([16000] - tf.shape(waveform), dtype=tf.float32)
# Concatenate audio with padding so that all audio clips will be of the
# same length
waveform = tf.cast(waveform, tf.float32)
equal_length = tf.concat([waveform, zero_padding], 0)
spectrogram = tf.py_function(func=stft, inp=[equal_length], Tout=tf.float32)
spectrogram.set_shape((129, 124))
return spectrogram
```

In the above code, **`get_spectrogram`** is a **function** that is mapped over a **TensorFlow DataSet** to compute the spectrogram from the normalized audio data. In the original code, the **STFT** was computed using the TensorFlow **`tf.signal.stft`** function.

But here's the problem – we won't have TensorFlow functions on the Raspberry Pi. What we'll have is our trusty **`Numpy`** and **`SciPy`** libraries. In addition, it turns out that **`tf.signal.stft`** and **`scipy.signal.stft`** output the data differently – it has to do with the scaling of the Fourier transfor.
We're using **TensorFlow 2.0** for this project which has eager execution turned on by default.

This means that in order to call a Python function from within a TensorFlow function you need to use **`tf.py_function`**. We use this construct to call our own function, **`sftf`** which in turn calls **`scipy.signal.stft`** to do the job.

Note that upon returning from this function, **we need explicitly set the shape of the tensor**, as we are going from Python back to TensorFlow. Note the use of **`np.abs`** – the **STFT** returns an **array of complex numbers**, and we take the **absolute values** of the result.

Also, about the spectrogram tensor shape. The official example had set up the **STFT** parameters such that the output has **shape `(124,129)`** – that is to say, **124 time steps**, with **each step** having **129 frequency bins** of values. Nothing special about these numbers, but for compatibility we'll stick to the same shape for our project.

Here's how a typical sample data and its spectrogram looks like:

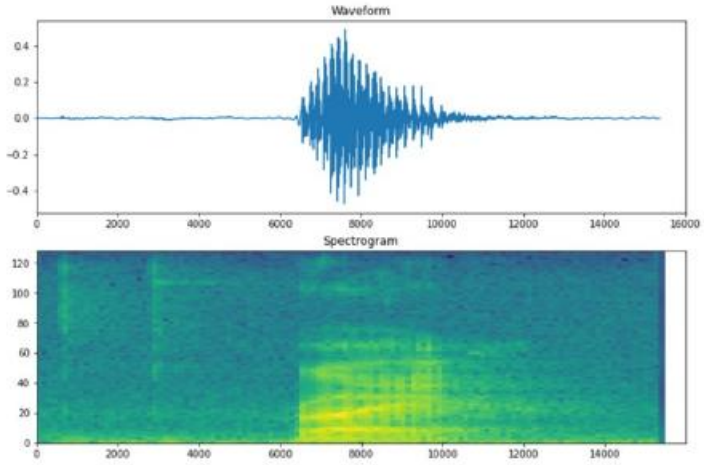

Fig 2.3 Audio spectogram produced by STFT from voice sample **shape `(124,129)`**.

You can see above how the evolution of the sound is captured much better in the spectrogram compared to the raw signal. This will be the input to our model.
Here's the neural network architecture used to train the data, as per the official example code:

The input passed through an initial scaling and normalization, and then through a couple of **convolution** layers, a **maxpool**, **dropout**, a fully connected dense layer, before the final dense layer that maps the input to the set of eight commands.

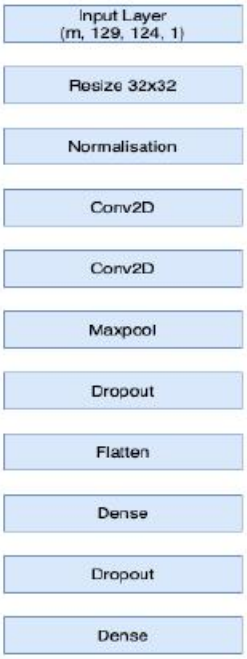

Fig 2.4 CNN model trained for audio spectrogram classification

The choice of the above architecture may seem a little arbitrary or complicated, but real-world neural networks are usually much deeper.

Once the model is compiled, it's trained as follows:

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

EPOCHS = 10
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS,
    callbacks=tf.keras.callbacks.EarlyStopping(verbose=1, patience=2),
)
```

For **10 epochs**, the training accuracy for this model is around **0.8389**. If you use **Google Colab** go up to 50 epochs. The next thing to do is **save** the **model** and **convert** it into `.tflite` model**.**

```
model.save('simple_audio_model_numpy.sav')

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model('simple_audio_model_numpy.sav') # path to the
SavedModel directory
tflite_model = converter.convert()

# Save the model.
with open('simple_audio_model_numpy.tflite', 'wb') as f:
  f.write(tflite_model)
```

Here's where **TensorFlow Lite** comes in. After saving the model, we convert the saved model to the Lite model. This is what we will use on the Raspberry Pi. There are many options like quantization, etc. when you convert your model to Lite, but we're just doing it in the simplest possible way here. You can read more about the TF Lite conversion here.

After this phase we can move on to the Raspberry Pi.

2.2 Our Jupyter notebook

We can do the actual training with **Jupyter Notebook** on **Google Colab**. You can find the notebook available at our **github** (`github.com/smartcomputerlab/EmbeddedAI`) .

Below is the content of the `simple_audio_command.ipynb` notebook.

2.2.1 Setup

Import necessary modules and dependencies. Note that you'll be using **`seaborn`** for visualization in this tutorial.

```
import os
import pathlib

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import models
from IPython import display

# Set the seed value for experiment reproducibility.
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)
```

2.2.2 Import the mini Speech Commands dataset

To save time with data loading, you will be working with a smaller version of the **Speech Commands dataset**.

The original dataset consists of over **105,000** audio files in the `.wav` format of people saying 35 different words. This data was collected by Google and released under a CC BY license.

Download and **extract** the `mini_speech_commands.zip` file containing the smaller Speech Commands datasets with `tf.keras.utils.get_file`:

```
DATASET_PATH = 'data/mini_speech_commands'

data_dir = pathlib.Path(DATASET_PATH)
if not data_dir.exists():
tf.keras.utils.get_file(
'mini_speech_commands.zip',
origin="http://storage.googleapis.com/download.tensorflow.org/data/mini_speech_commands.zip",
extract=True,
cache_dir='.', cache_subdir='data')
```

```
Downloading data from
http://storage.googleapis.com/download.tensorflow.org/data/mini_speech_commands.zip
182083584/182082353 [==============================] - 1s 0us/step
182091776/182082353 [==============================] - 1s 0us/step
```

The dataset's audio clips are stored in **eight folders** corresponding to each speech command: `no`, `yes`, `down`, `go`, `left`, `up`, `right`, and `stop`:

```
commands = np.array(tf.io.gfile.listdir(str(data_dir)))
commands = commands[commands != 'README.md']
print('Commands:', commands)
```

```
Commands: ['stop' 'go' 'no' 'up' 'down' 'right' 'left' 'yes']
```

Extract the audio clips into a list called `filenames`, and shuffle it:

```
filenames = tf.io.gfile.glob(str(data_dir) + '/*/*')
filenames = tf.random.shuffle(filenames)
num_samples = len(filenames)
print('Number of total examples:', num_samples)
print('Number of examples per label:',
len(tf.io.gfile.listdir(str(data_dir/commands[0]))))
```

```
print('Example file tensor:', filenames[0])

Number of total examples: 8000
Number of examples per label: 1000
Example file tensor: tf.Tensor(b'data/mini_speech_commands/right/cc6ee39b_nohash_3.wav', shape=(),
dtype=string)
```

Split **`filenames`** into **training**, **validation** and **test** sets using a **80:10:10** ratio, respectively:

```
train_files = filenames[:6400]
val_files = filenames[6400: 6400 + 800]
test_files = filenames[-800:]

print('Training set size', len(train_files))
print('Validation set size', len(val_files))
print('Test set size', len(test_files))

Training set size 6400
Validation set size 800
Test set size 800
```

2.2.3 Read the audio files and their labels

In this section you will **preprocess** the **dataset**, creating **decoded tensors** for the **waveforms** and the corresponding **labels**.
Note that:

- Each **`.wav`** file contains time-series data with a set **number of samples per second**.
- Each sample represents the amplitude of the audio signal at that specific time.
- In a **16-bit** system, like the **`.wav`** files in the **mini Speech Commands dataset**, the **amplitude** values range from **-32,768** to **32,767**.
- The sample rate for this dataset is 16kHz.

The shape of the tensor returned by **`tf.audio.decode_wav is [samples,channels]`**, where **`channels`** is **1** for **mono** or **2** for **stereo**. The **mini Speech Commands dataset** only **contains mono recordings**.

```
test_file = tf.io.read_file(DATASET_PATH+'/down/0a9f9af7_nohash_0.wav')
test_audio, _ = tf.audio.decode_wav(contents=test_file)
test_audio.shape

TensorShape([13654, 1])
```

Now, let's define a function that preprocesses the dataset's **raw `.wav`** audio files into **audio tensors**:

```
def decode_audio(audio_binary):
        # Decode WAV-encoded audio files to `float32` tensors, normalized
        # to the [-1.0, 1.0] range. Return `float32` audio and a sample rate.
        audio, _ = tf.audio.decode_wav(contents=audio_binary)
        # Since all the data is single channel (mono), drop the `channels`
        # axis from the array.
        return tf.squeeze(audio, axis=-1)
```

Define a function that **creates labels** using the **parent directories** for each file:

- Split the file paths into **`tf.RaggedTensors`** (tensors with ragged dimensions—with slices that may have different lengths).

```
def get_label(file_path):
        parts = tf.strings.split(
        input=file_path,
        sep=os.path.sep)
        # Note: You'll use indexing here instead of tuple unpacking to enable this
        # to work in a TensorFlow graph.
        return parts[-2]
```

Define another helper function—**`get_waveform_and_label`**—that puts it all together:

- The input is the `.wav` audio filename.
- The output is a **tuple** containing the **audio** and **label tensors** ready for **supervised learning**.

```
def get_waveform_and_label(file_path):
        label = get_label(file_path)
        audio_binary = tf.io.read_file(file_path)
        waveform = decode_audio(audio_binary)
        return waveform, label
```

Build the training set to extract the audio-label pairs:

- Create a `tf.data.Dataset` with `Dataset.from_tensor_slices` and `Dataset.map`, using `get_waveform_and_label` defined earlier.

You'll build the validation and test sets using a similar procedure later on.

```
AUTOTUNE = tf.data.AUTOTUNE

files_ds = tf.data.Dataset.from_tensor_slices(train_files)

waveform_ds = files_ds.map(
map_func=get_waveform_and_label,
num_parallel_calls=AUTOTUNE)
```

Let's plot a few audio waveforms:

```
rows = 3
cols = 3
n = rows * cols
fig, axes = plt.subplots(rows, cols, figsize=(10, 12))

for i, (audio, label) in enumerate(waveform_ds.take(n)):
        r = i // cols
        c = i % cols
        ax = axes[r][c]
        ax.plot(audio.numpy())
        ax.set_yticks(np.arange(-1.2, 1.2, 0.2))
        label = label.numpy().decode('utf-8')
        ax.set_title(label)

plt.show()
```

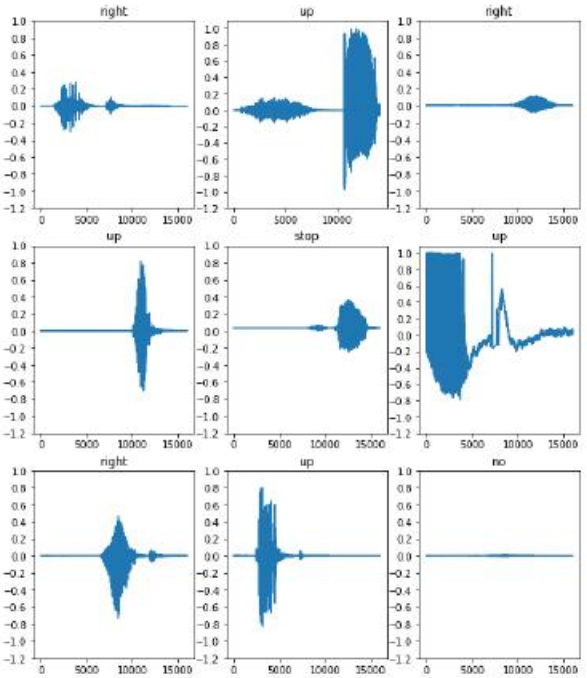

Fig 2.5 Plotting the audio samples with `audio.numpy()`

2.2.4 Convert waveforms to spectrograms

The **waveforms** in the dataset are represented in the **time domain**. Next, you'll transform the waveforms from the **time-domain** signals into the **time-frequency-domain** signals by computing the **short-time Fourier transform – STFT** to convert the **waveforms** to as **spectrograms**, which show **frequency changes over time** and can be represented as **2D images**. You will feed the **spectrogram images** into your neural network to train the model.

A **Fourier transform** (`tf.signal.fft`) converts a signal to its component frequencies, but **loses all time information**. In comparison, **STFT** (`tf.signal.stft`) splits the signal into **windows of time** and runs a **Fourier transform on each window**, preserving some time information, and returning a **2D tensor** that you can run **standard convolutions** on.

Create a **utility function** for converting **waveforms** to **spectrograms**:

- The waveforms need to be of the same length, so that when you convert them to spectrograms, the results have similar dimensions. This can be done by simply **zero-padding** the audio clips that are shorter than **one second** (using `tf.zeros`).
- When calling `tf.signal.stft`, choose the `frame_length` and `frame_step` parameters such that the generated spectrogram "image" is **almost square**.
- The **STFT produces an array of complex numbers** representing **magnitude and phase**. However, in this tutorial you'll **only use the magnitude**, which you can derive by applying `tf.abs` on the output of `tf.signal.stft`.

```
def get_spectrogram(waveform):
        # Zero-padding for an audio waveform with less than 16,000 samples.
        input_len = 16000
        waveform = waveform[:input_len]
        zero_padding = tf.zeros(
                [16000] - tf.shape(waveform),
                dtype=tf.float32)
        # Cast the waveform tensors' dtype to float32.
        waveform = tf.cast(waveform, dtype=tf.float32)
        # Concatenate the waveform with `zero_padding`, which ensures all audio
        # clips are of the same length.
        equal_length = tf.concat([waveform, zero_padding], 0)
        # Convert the waveform to a spectrogram via a STFT.
        spectrogram = tf.signal.stft(
                equal_length, frame_length=255, frame_step=128)
        # Obtain the magnitude of the STFT.
        spectrogram = tf.abs(spectrogram)
        # Add a `channels` dimension, so that the spectrogram can be used
        # as image-like input data with convolution layers (which expect
        # shape (`batch_size`, `height`, `width`, `channels`).
        spectrogram = spectrogram[..., tf.newaxis]
        return spectrogram
```

Next, start **exploring the data**. Print the **shapes** of one example's **tensorized waveform** and the corresponding **spectrogram**, and **play** the original audio:

```
for waveform, label in waveform_ds.take(1):
        label = label.numpy().decode('utf-8')
        spectrogram = get_spectrogram(waveform)

        print('Label:', label)
        print('Waveform shape:', waveform.shape)
        print('Spectrogram shape:', spectrogram.shape)
        print('Audio playback')
        display.display(display.Audio(waveform, rate=16000))
```

Now, define a function for displaying a spectrogram:

```
def plot_spectrogram(spectrogram, ax):
        if len(spectrogram.shape) > 2:
                assert len(spectrogram.shape) == 3
                spectrogram = np.squeeze(spectrogram, axis=-1)
        # Convert the frequencies to log scale and transpose, so that the time is
        # represented on the x-axis (columns).
        # Add an epsilon to avoid taking a log of zero.
        log_spec = np.log(spectrogram.T + np.finfo(float).eps)
```

```
        height = log_spec.shape[0]
        width = log_spec.shape[1]
        X = np.linspace(0, np.size(spectrogram), num=width, dtype=int)
        Y = range(height)
        ax.pcolormesh(X, Y, log_spec)
```

Plot the example's waveform over time and the corresponding spectrogram (frequencies over time):

```
fig, axes = plt.subplots(2, figsize=(12, 8))
timescale = np.arange(waveform.shape[0])
axes[0].plot(timescale, waveform.numpy())
axes[0].set_title('Waveform')
axes[0].set_xlim([0, 16000])

plot_spectrogram(spectrogram.numpy(), axes[1])
axes[1].set_title('Spectrogram')
plt.show()
```

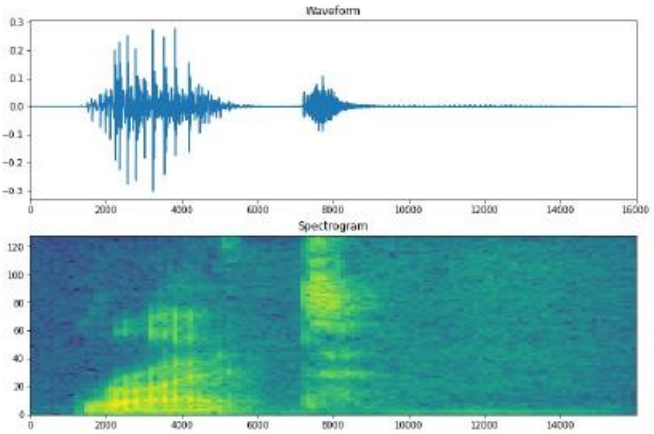

Fig 2.6 Plotting a “normalized” audio sample and the corresponding spectogram.

Now, define a function that transforms the waveform dataset into spectrograms and their corresponding **labels** as **integer IDs**:

```
def get_spectrogram_and_label_id(audio, label):
        spectrogram = get_spectrogram(audio)
        label_id = tf.math.argmax(label == commands)
        return spectrogram, label_id
```

Map **`get_spectrogram_and_label_id`** across the dataset's elements with **`Dataset.map`**:

```
spectrogram_ds = waveform_ds.map(
map_func=get_spectrogram_and_label_id,
num_parallel_calls=AUTOTUNE)
```

Examine the spectrograms for different examples of the dataset:

```
rows = 3
cols = 3
n = rows*cols
fig, axes = plt.subplots(rows, cols, figsize=(10, 10))

for i, (spectrogram, label_id) in
enumerate(spectrogram_ds.take(n)):
        r = i // cols
        c = i % cols
        ax = axes[r][c]
        plot_spectrogram(spectrogram.numpy(), ax)
        ax.set_title(commands[label_id.numpy()])
        ax.axis('off')

plt.show()
```

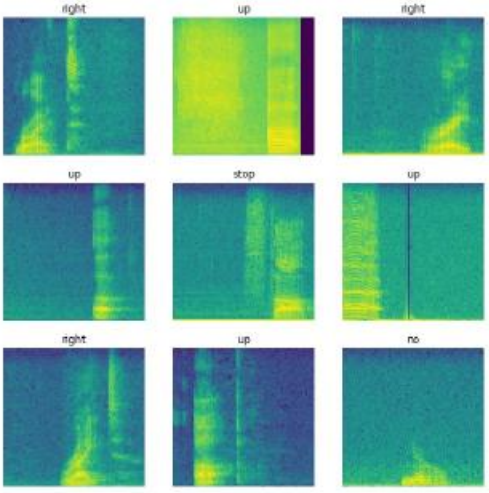

Fig 2.7 Plotting the set of the spectograms.

2.2.5 Build and train the model

Repeat the training set preprocessing on the validation and test sets:

```
def preprocess_dataset(files):
        files_ds = tf.data.Dataset.from_tensor_slices(files)
        output_ds = files_ds.map(
        map_func=get_waveform_and_label,
        num_parallel_calls=AUTOTUNE)
        output_ds = output_ds.map(
        map_func=get_spectrogram_and_label_id,
        num_parallel_calls=AUTOTUNE)
        return output_ds

train_ds = spectrogram_ds
val_ds = preprocess_dataset(val_files)
test_ds = preprocess_dataset(test_files)
```

Batch the training and validation sets for model training:

```
batch_size = 64
train_ds = train_ds.batch(batch_size)
val_ds = val_ds.batch(batch_size)
```

Add `Dataset.cache` and `Dataset.prefetch` operations to reduce read latency while training the model:

```
train_ds = train_ds.cache().prefetch(AUTOTUNE)
val_ds = val_ds.cache().prefetch(AUTOTUNE)
```

For the model, you'll use a **simple convolutional neural network** (CNN), since you have transformed the audio files into spectrogram images.

Your **`tf.keras.Sequential`** model will use the following Keras preprocessing layers:

- **`tf.keras.layers.Resizing`**: to downsample the input to enable the model to train faster.
- **`tf.keras.layers.Normalization`**: to normalize each pixel in the image based on its mean and standard deviation.

For the **`Normalization`** layer, its **`adapt`** method would first need to be called on the training data in order to compute **aggregate statistics** (that is, the **mean** and the **standard deviation**).

```
for spectrogram, _ in spectrogram_ds.take(1):
input_shape = spectrogram.shape
print('Input shape:', input_shape)
num_labels = len(commands)

# Instantiate the `tf.keras.layers.Normalization` layer.
norm_layer = layers.Normalization()
# Fit the state of the layer to the spectrograms
# with `Normalization.adapt`.
norm_layer.adapt(data=spectrogram_ds.map(map_func=lambda spec, label: spec))

model = models.Sequential([
        layers.Input(shape=input_shape),
        # Downsample the input.
        layers.Resizing(32, 32),
        # Normalize.
        norm_layer,
        layers.Conv2D(32, 3, activation='relu'),
        layers.Conv2D(64, 3, activation='relu'),
        layers.MaxPooling2D(),
        layers.Dropout(0.25),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(num_labels),
])

model.summary()

Input shape: (124, 129, 1)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resizing (Resizing)         (None, 32, 32, 1)         0

 normalization (Normalizatio  (None, 32, 32, 1)        3
 n)

 conv2d (Conv2D)             (None, 30, 30, 32)        320

 conv2d_1 (Conv2D)           (None, 28, 28, 64)        18496

 max_pooling2d (MaxPooling2D  (None, 14, 14, 64)       0
 )

 dropout (Dropout)           (None, 14, 14, 64)        0

 flatten (Flatten)           (None, 12544)             0

 dense (Dense)               (None, 128)               1605760

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 8)                 1032

=================================================================
Total params: 1,625,611
Trainable params: 1,625,608
Non-trainable params: 3
```

Configure the Keras model with the Adam optimizer and the cross-entropy loss:

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

Train the model over **10 (50) epochs** for demonstration purposes:

```
EPOCHS = 10
history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=EPOCHS,
        callbacks=tf.keras.callbacks.EarlyStopping(verbose=1, patience=2),
)
```

```
Epoch 1/10
100/100 [==============================] - 26s 147ms/step - loss: 1.7292 - accuracy: 0.3767 - val_loss: 1.2869 - val_accuracy: 0.5775
Epoch 2/10
100/100 [==============================] - 1s 7ms/step - loss: 1.1682 - accuracy: 0.5834 - val_loss: 0.9513 - val_accuracy: 0.6725
Epoch 3/10
100/100 [==============================] - 1s 7ms/step - loss: 0.9229 - accuracy: 0.6711 - val_loss: 0.7656 - val_accuracy: 0.7412
Epoch 4/10
100/100 [==============================] - 1s 7ms/step - loss: 0.7578 - accuracy: 0.7303 - val_loss: 0.6540 - val_accuracy: 0.7638
Epoch 5/10
100/100 [==============================] - 1s 7ms/step - loss: 0.6410 - accuracy: 0.7686 - val_loss: 0.6024 - val_accuracy: 0.8050
Epoch 6/10
100/100 [==============================] - 1s 7ms/step - loss: 0.5660 - accuracy: 0.7972 - val_loss: 0.5883 - val_accuracy: 0.8037
Epoch 7/10
100/100 [==============================] - 1s 7ms/step - loss: 0.5160 - accuracy: 0.8150 - val_loss: 0.5786 - val_accuracy: 0.8163
Epoch 8/10
100/100 [==============================] - 1s 7ms/step - loss: 0.4740 - accuracy: 0.8344 - val_loss: 0.5409 - val_accuracy: 0.8350
Epoch 9/10
100/100 [==============================] - 1s 7ms/step - loss: 0.4082 - accuracy: 0.8567 - val_loss: 0.5278 - val_accuracy: 0.8438
Epoch 10/10
100/100 [==============================] - 1s 7ms/step - loss: 0.3880 - accuracy: 0.8602 - val_loss: 0.5036 - val_accuracy: 0.8487
```

Let's plot the training and validation loss curves to check how your model has improved during training:

```
metrics = history.history
plt.plot(history.epoch, metrics['loss'], metrics['val_loss'])
plt.legend(['loss', 'val_loss'])
```

```
plt.show()
```

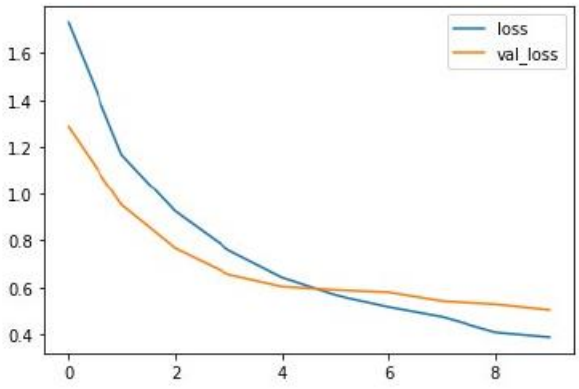

Fig 2.8 Plotting the loss and validation loss values for 10 epochs

2.2.6 Evaluate the model performance

Run the model on the test set and check the model's performance:

```
test_audio = []
test_labels = []

for audio, label in test_ds:
        test_audio.append(audio.numpy())
        test_labels.append(label.numpy())

test_audio = np.array(test_audio)
test_labels = np.array(test_labels)

y_pred = np.argmax(model.predict(test_audio), axis=1)
y_true = test_labels

test_acc = sum(y_pred == y_true) / len(y_true)
print(f'Test set accuracy: {test_acc:.0%}')
```

Test set accuracy: 82%

2.2.6.1 Display a confusion matrix

Use a **confusion matrix** to check how well the model did classifying each of the commands in the test set:

```
confusion_mtx = tf.math.confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_mtx,
        xticklabels=commands,
        yticklabels=commands,
        annot=True, fmt='g')
plt.xlabel('Prediction')
plt.ylabel('Label')
plt.show()
```

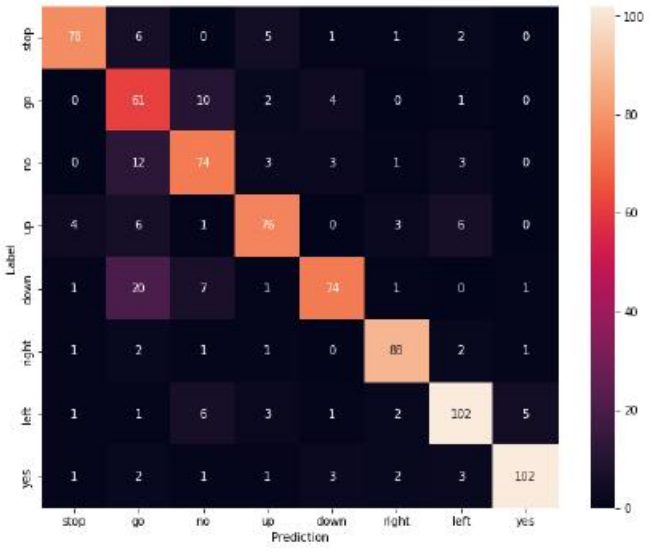

Fig 2.9 Plotting the **confusion matrix**

2.2.7 Run inference on an audio file

Finally, verify the model's prediction output using an input audio file of someone saying "no". How well does your model perform?

```
sample_file = data_dir/'no/01bb6a2a_nohash_0.wav'

sample_ds = preprocess_dataset([str(sample_file)])

for spectrogram, label in sample_ds.batch(1):
        prediction = model(spectrogram)
        plt.bar(commands, tf.nn.softmax(prediction[0]))
        plt.title(f'Predictions for "{commands[label[0]]}"')
        plt.show()
```

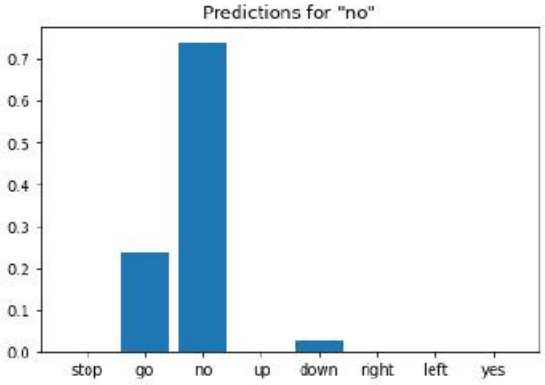

Fig 2.10 Plotting the inference prediction values for given audio sample

As the output suggests, your model should have recognized the audio command as "**no**".

2.2.8 Saving the model in `.tflite` format

The next thing to do is save the model and convert it to **`simple_audio_model_numpy.tflite`** model.

```
model.save('simple_audio_model_numpy.sav')

# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model('simple_audio_model_numpy.sav') # path to the
SavedModel directory
tflite_model = converter.convert()

# Save the model.
with open('simple_audio_model_numpy.tflite', 'wb') as f:
f.write(tflite_model)


INFO:tensorflow:Assets written to: simple_audio_model_numpy.sav/assets
WARNING:absl:Buffer deduplication procedure will be skipped when flatbuffer library is not properly
loaded
```

Here's where **TensorFlow Lite** comes in. After saving the model, we convert the saved model to the Lite model. This is what we will use on the Raspberry Pi. There are many options like quantization, etc. when you convert your model to Lite, but we're just doing it in the simplest possible way here.
You can read more about the **TF Lite conversion** here.

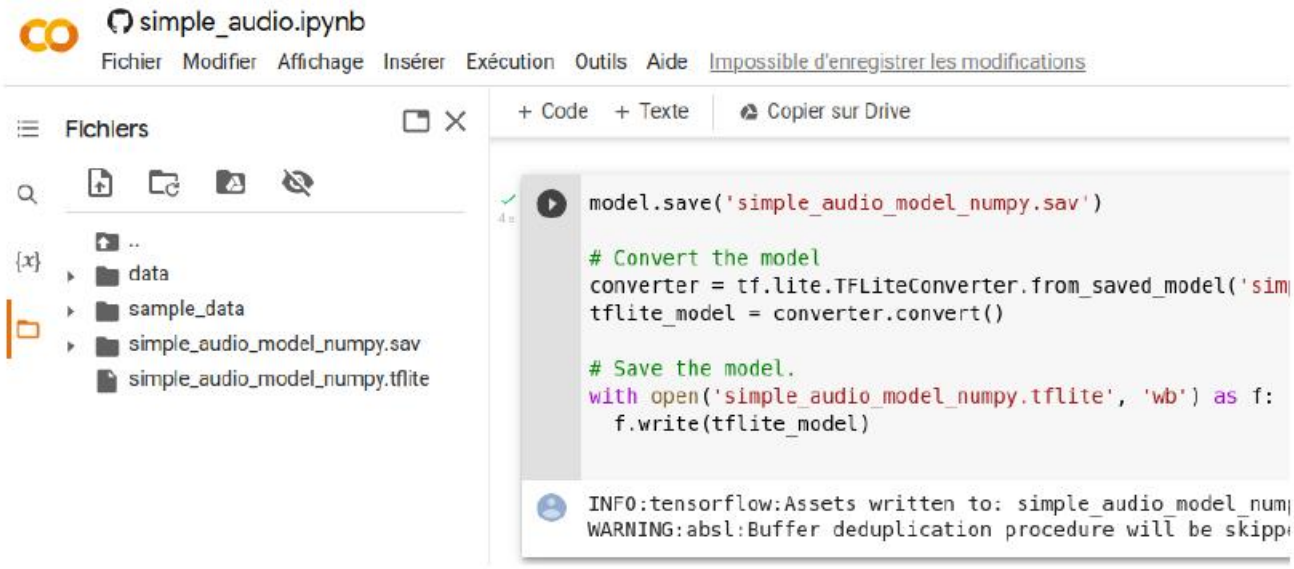

Fig 2.11 Saving and converting the **TensorFlow** model to **TensorFlow Lite** model in your **Google Colab** environment

Download **`simple_audio_model_numpy.tflite`** to your Raspberry Pi.

Now it's time to move on to the Raspberry Pi.

2.3 Inference using TensorFlow Lite on Raspberry Pi

Your Raspberry Pi board contains prepared OS with all necessary **ML libraries** including **TensorFlow**, **TensorFlow Lite** with **TensorFlow Lite interpreter** etc.
For the OLED display (optional) you will also need to install the Adafruit_Python_SSD1306 module.

Transfer the **TF Lite** model (**`simple_audio_model_numpy.tflite`**) to the **Pi** and **test it** out as follows:

```
$ python3
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from tflite_runtime.interpreter import Interpreter
>>> interpreter = Interpreter('simple_audio_model_numpy.tflite')
>>> interpreter.allocate_tensors()
>>> interpreter.get_input_details()
[{'name': 'input_3', 'index': 0, 'shape': array([  1, 129, 124,   1]), 'shape_signature':
array([  1, 129, 124,   1]), 'dtype': <class 'numpy.float32'>, 'quantization': (0.0, 0),
'quantization_parameters': {'scales': array([], dtype=float32), 'zero_points': array([],
dtype=int32), 'quantized_dimension': 0}, 'sparsity_parameters': {}}]
>>> interpreter.get_output_details()
[{'name': 'Identity', 'index': 17, 'shape': array([1, 8]), 'shape_signature': array([1, 8]),
'dtype': <class 'numpy.float32'>, 'quantization': (0.0, 0), 'quantization_parameters': {'scales':
array([], dtype=float32), 'zero_points': array([], dtype=int32), 'quantized_dimension': 0},
'sparsity_parameters': {}}]
>>>
```

So you can see that the input expected by the model is a tensor of **`shape (1, 129, 124,1)`** – the **spectrogram** which we used to train the model – and the expected output is **`(1, 8)`** – the eight categories of commands.

Now let's look at how to get the audio input for inference.

2.3.1 Audio from an I2S Microphone on WM8960 audio HAT

We will use **`pyaudio`** to grab audio data from the microphone. In this lab, we are using **WM8960 audio HAT**.

- **WM8960 codec** uses I2S digital audio for both input and output
- On/Off privacy switch to deactivate audio so you *know* it can't be recording.
- Two analog microphone inputs (left and right)
- Two 1W speaker outputs
- 3.5mm stereo headphone or line-out audio
- Plugs into any Raspberry Pi with 2x20 headers

With the extra space on the PCB we also added some bonus extras!

- Push button - Use to change modes, activate the voice assistant, anything you like!
- Three DotStar RGB LEDs - add LED feedback or make a rainbow light show
- STEMMA QT connector - plug in any of our I2C sensors, OLEDs, or accessories.
- 3 Pin JST STEMMA connector - for larger accessories, like NeoPixels, a relay or servo

2.3.1.1 Install software for WM8960 audio HAT

Make sure you've got the **WM8960 audio HAT** installed, and I2C support installed as well!
When you run :

```
sudo i2cdetect -y 1
```

you should see an entry under **1A**, indicating the hardware sees the audio card. The number may also appear as **UU** if you already installed software.

Attention:
Skip the following steps if the driver software is installed.

```
cd ~
sudo apt-get install -y git
git clone https://github.com/HinTak/seeed-voicecard
cd seeed-voicecard
git checkout v5.9
sudo ./install.sh
```

Reboot with

```
sudo reboot
```

and on reboot run

```
sudo aplay -l
```

To list all sound cards, you should see it at the bottom. If your card number differs from the above image, take note of your number.
You can use **`alsamixer`** to adjust the volume, don't forget to select the card with **F6**

2.3.1.2 Python Libraries and usage

The **Microphone and Voice Card** are installed as Linux level devices, so using them in is done as you would with any system level audio device. If you would like to make use of audio in Python, you can use the **`pyaudio`** library. To install **`pyaudio`** and its dependencies, run the following code:

```
sudo pip3 install pyaudio
sudo apt-get install libportaudio2
```

Here is a basic test script to **enumerate the devices and record for 10 seconds**. When prompted, choose the device called **`seeed-2mic-voicecard`**.
Copy and paste the following code into a file called **`audiotest.py`**.

```
import pyaudio
import wave

FORMAT = pyaudio.paInt16
CHANNELS = 1           # Number of channels
BITRATE = 44100        # Audio Bitrate
CHUNK_SIZE = 512       # Chunk size to
RECORDING_LENGTH = 10  # Recording Length in seconds
WAVE_OUTPUT_FILENAME = "myrecording.wav"
audio = pyaudio.PyAudio()

info = audio.get_host_api_info_by_index(0)
numdevices = info.get('deviceCount')
for i in range(0, numdevices):
    if (audio.get_device_info_by_host_api_device_index(0, i).get('maxInputChannels')) > 0:
        print("Input Device id ", i, " - ", audio.get_device_info_by_host_api_device_index(0,
i).get('name'))

print("Which Input Device would you like to use?")
device_id = int(input()) # Choose a device
print("Recording using Input Device ID "+str(device_id))

stream = audio.open(
    format=FORMAT,
    channels=CHANNELS,
    rate=BITRATE,
    input=True,
    input_device_index = device_id,
    frames_per_buffer=CHUNK_SIZE
)

recording_frames = []

for i in range(int(BITRATE / CHUNK_SIZE * RECORDING_LENGTH)):
    data = stream.read(CHUNK_SIZE)
    recording_frames.append(data)
```

```
stream.stop_stream()
stream.close()
audio.terminate()

waveFile = wave.open(WAVE_OUTPUT_FILENAME, 'wb')
waveFile.setnchannels(CHANNELS)
waveFile.setsampwidth(audio.get_sample_size(FORMAT))
waveFile.setframerate(BITRATE)
waveFile.writeframes(b''.join(recording_frames))
waveFile.close()
```

Run the code with the following command:

```
sudo python3 audiotest.py
```

When finished, you can test playing back the file with the following command:

```
aplay recordedFile.wav
```

pyaudio works by opening the audio stream and reading in chunks of data. The input stream is configures to read in 32 bit 2-channel data at 16000 Hz. Above, we are discarding the first few seconds of data. I did this because I found that when the stream starts up, there is loud click every time – maybe something to do with audio initialisation. The data frames are then combined together and written to a WAV file. Let's try it out.

```
$python3 audiotest.py --nsec 3 --output hello.wav
```

This is what the **.wav** file looks like in Audacity.

These experiments give us a sense of how to process the audio stream before we send it as input to our **TensorFlow Lite interpreter**.

2.3.2 Processing audio for Inference

As you saw during the training phase of our model, we need to compute the spectrogram from the audio data before we do any inference.
To be consistent with what we did during training, we first need to ensure that the audio data is in the following format:

One second long, single channel, 32-bit values sampled at 16000 Hz, normalised to [-1.0, 1.0].

Another thing we need is the ability to **slice** the audio data. Since we'll have an **"always on" kind of listening system**, we need to listen for say 3 seconds, and take a best guess of where the relevant 1 second audio is within that input data before we try inference on it.

Take a look at the Jupyter notebook **slice_audio.ipynb** which tests out these ideas.

```
from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt
from IPython import display

VERBOSE_DEBUG = True

def print_info(waveform):
        # audio data
        if VERBOSE_DEBUG:
        print("waveform:", waveform.shape, waveform.dtype, type(waveform))
        print(waveform[:5])


def show_audio(wavfile_name):
        # get audio data
        rate, waveform0 = wavfile.read(wavfile_name)
        print_info(waveform0)
        # if stereo, pick the left channel
        waveform = None
        if len(waveform0.shape) == 2:
                print("Stereo detected. Picking one channel.")
                waveform = waveform0.T[1]
```

```
else:
        waveform = waveform0

# normalise audio
wabs = np.abs(waveform)
wmax = np.max(wabs)
waveform = waveform / wmax
display.display(display.Audio(waveform, rate = 16000))
print("signal max: %f RMS: %f abs: %f " % (np.max(waveform),
        np.sqrt(np.mean(waveform**2)),
        np.mean(np.abs(waveform))))
max_index = np.argmax(waveform)
print("max_index = ", max_index)
fig, axes = plt.subplots(4, figsize=(10, 8))
timescale = np.arange(waveform0.shape[0])
axes[0].plot(timescale, waveform0)
timescale = np.arange(waveform.shape[0])
axes[1].plot(timescale, waveform)

# scale and center
waveform = 2.0*(waveform - np.min(waveform))/np.ptp(waveform) - 1
timescale = np.arange(waveform.shape[0])
axes[2].plot(timescale, waveform)
timescale = np.arange(16000)
start_index = max(0, max_index-8000)
end_index = min(max_index+8000, waveform.shape[0])
axes[3].plot(timescale, waveform[start_index:end_index])
plt.show()
```

The first thing we do it is to pick the left channel (in my case) from the 2-channel data. Notice the *.T* or transpose of the data – that's because of the shape of the incoming data:

```
[[         0 -122814464]
 [         0 -122912768]…]
```

While working with data, it's a good idea to print out the shape of your numpy arrays at various stages to verify that your assumptions are correct.

The data is then normalized by dividing the values by the maximum of the absolute values. But this will still not center the data as we need. So, we scale and center it using the **peak-to-peak** (`np.ptp`) values of the data.

Next, we need to pick the relevant 1 second of data from the input. For this. we first find the index of the **maximum data amplitude** using *np.argmax*. We then extract a **one second clip centered around this value**.

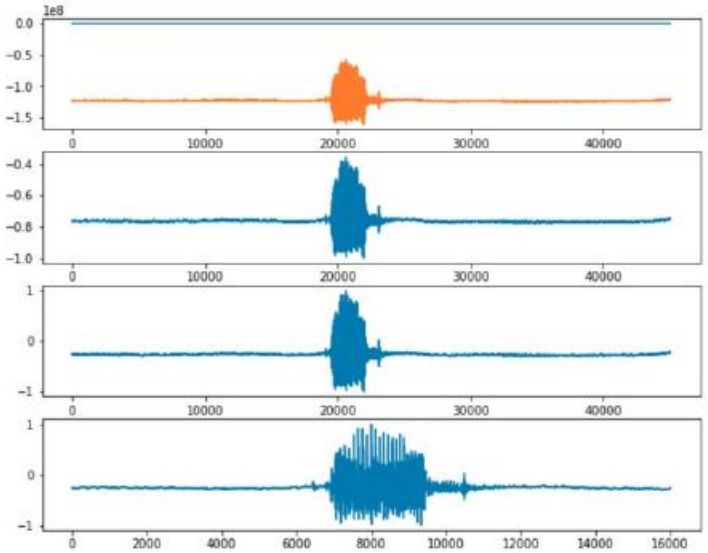

Fig 2.11 The plots of a 3 second audio data as it goes through the above stages:

Note that if the maximum amplitude is close to the start or end of the clip, we will end up with an extracted clip of less than one second. So we need to pad the data with zeros as follows:

```
waveform_padded = np.zeros((16000,))
waveform_padded[:waveform.shape[0]] = waveform
```

One more thing about the audio data. We want to **skip inference** if there is not much action going on. For this. we can make use of the **peak-to-peak** data as follows:

```
PTP = np.ptp(waveform)
print("peak-to-peak: %.4f. Adjust as needed." % (PTP,))

# return None if too silent
if PTP < 0.5:
    return []
```

Now we're ready to look at the spectrogram. We can safely use **`scipy.signal.stft`**, since we used the same function during training of the data.

```
def get_spectrogram(waveform):
    waveform_padded = process_audio_data(waveform)
    if not len(waveform_padded):
        return []
    # compute spectrogram
    f, t, Zxx = signal.stft(waveform_padded, fs=16000, nperseg=255,
        noverlap = 124, nfft=256)
    # Output is complex, so take abs value
    spectrogram = np.abs(Zxx)

    if VERBOSE_DEBUG:
        print("spectrogram:", spectrogram.shape, type(spectrogram))
        print(spectrogram[0, 0])

    return spectrogram
```

2.3.3 OLED Display (optional)

Since want to be a bit fancy on the output, we're going to hook up an 128 x 64 pixel I2C OLED display to the Raspberry Pi. It's connected to the Pi as follows:

OLED Display	Raspberry Pi
VCC	3.3 V
GND	GND
SDA	GPIO 2 (Header pin 3)
SCL	GPIO 3 (Header pin 5)

For displaying text, there is a helper class called **`display_ssd1306.py`** in the repository which makes use of the **Adafruit_SSD1306** library.

2.3.4 Putting it all Together

Now we have all the pieces required to run inference on the incoming audio data on the Pi. We know how to extract the audio data, process it, scale it correctly and compute the spectrogram. Next we need to use the TensorFlow Lite mode and run inference on it using the input data.

Here's the inference code from simple_audio_nodisp.py from the repository.

```
from scipy.io import wavfile
from scipy import signal
import numpy as np
import argparse
import pyaudio
import wave
import time

from tflite_runtime.interpreter import Interpreter

#from display_ssd1306 import SSD1306_Display

VERBOSE_DEBUG = False

# get pyaudio input device
def getInputDevice(p):
    index = None
    nDevices = p.get_device_count()
    print('Found %d devices:' % nDevices)
    for i in range(nDevices):
        deviceInfo = p.get_device_info_by_index(i)
        #print(deviceInfo)
        devName = deviceInfo['name']
        print(devName)
        # look for the "input" keyword
        # choose the first such device as input
        # change this loop to modify this behavior
        # maybe you want "mic"?
        if not index:
            if 'input' in devName.lower():
                index = i
    # print out chosen device
    if index is not None:
        devName = p.get_device_info_by_index(index)["name"]
        #print("Input device chosen: %s" % devName)
    return 3   #index

get_live_input():
    CHUNK = 4096
    FORMAT = pyaudio.paInt32
    CHANNELS = 2
    RATE = 16000
    RECORD_SECONDS = 3
    WAVE_OUTPUT_FILENAME = "test.wav"
    NFRAMES = int((RATE * RECORD_SECONDS) / CHUNK)

    # initialize pyaudio
    p = pyaudio.PyAudio()
    getInputDevice(p)  # p=3

    print('opening stream...')
    stream = p.open(format = FORMAT,
                    channels = CHANNELS,
                    rate = RATE,
                    input = True,
                    frames_per_buffer = CHUNK,
                    input_device_index = 3)

    # discard first 1 second
    for i in range(0, NFRAMES):
        data = stream.read(CHUNK, exception_on_overflow = False)

    try:
        while True:
            print("Listening...")

            frames = []
```

```
            for i in range(0, NFRAMES):
                data = stream.read(CHUNK, exception_on_overflow = False)
                frames.append(data)

            # process data
            # 4096 * 3 frames * 2 channels * 4 bytes = 98304 bytes
            # CHUNK * NFRAMES * 2 * 4
            buffer = b''.join(frames)
            audio_data = np.frombuffer(buffer, dtype=np.int32)
            nbytes = CHUNK * NFRAMES
            # reshape for input
            audio_data = audio_data.reshape((nbytes, 2))
            # run inference on audio data
            run_inference(audio_data)
    except KeyboardInterrupt:
        print("exiting…")

   stream.stop_stream()
    stream.close()
    p.terminate()

def process_audio_data(waveform):
    """Process audio input.

    This function takes in raw audio data from a WAV file and does scaling
    and padding to 16000 length.

    """

    if VERBOSE_DEBUG:
        print("waveform:", waveform.shape, waveform.dtype, type(waveform))
        print(waveform[:5])

    # if stereo, pick the left channel
    if len(waveform.shape) == 2:
        print("Stereo detected. Picking one channel.")
        waveform = waveform.T[1]
    else:
        waveform = waveform

    if VERBOSE_DEBUG:
        print("After scaling:")
        print("waveform:", waveform.shape, waveform.dtype, type(waveform))
        print(waveform[:5])

    # normalise audio
    wabs = np.abs(waveform)
    wmax = np.max(wabs)
    waveform = waveform / wmax

    PTP = np.ptp(waveform)
    print("peak-to-peak: %.4f. Adjust as needed." % (PTP,))

    # return None if too silent
    if PTP < 0.5:
        return []

    if VERBOSE_DEBUG:
        print("After normalisation:")
        print("waveform:", waveform.shape, waveform.dtype, type(waveform))
        print(waveform[:5])

  # scale and center
    waveform = 2.0*(waveform - np.min(waveform))/PTP - 1

    # extract 16000 len (1 second) of data
    max_index = np.argmax(waveform)
    start_index = max(0, max_index-8000)
    end_index = min(max_index+8000, waveform.shape[0])
    waveform = waveform[start_index:end_index]

    # Padding for files with less than 16000 samples
    if VERBOSE_DEBUG:
        print("After padding:")

    waveform_padded = np.zeros((16000,))
```

```
    waveform_padded[:waveform.shape[0]] = waveform

    if VERBOSE_DEBUG:
        print("waveform_padded:", waveform_padded.shape, waveform_padded.dtype,
type(waveform_padded))
        print(waveform_padded[:5])

    return waveform_padded

def get_spectrogram(waveform):

    waveform_padded = process_audio_data(waveform)

    if not len(waveform_padded):
        return []

    # compute spectrogram
    f, t, Zxx = signal.stft(waveform_padded, fs=16000, nperseg=255,
        noverlap = 124, nfft=256)
    # Output is complex, so take abs value
    spectrogram = np.abs(Zxx)

    if VERBOSE_DEBUG:
        print("spectrogram:", spectrogram.shape, type(spectrogram))
        print(spectrogram[0, 0])

    return spectrogram


def run_inference(waveform):

    # get spectrogram data
    spectrogram = get_spectrogram(waveform)

    if not len(spectrogram):
        #disp.show_txt(0, 0, "Silent. Skip...", True)
        print("Too silent. Skipping...")
        #time.sleep(1)
        return

    spectrogram1= np.reshape(spectrogram, (-1, spectrogram.shape[0], spectrogram.shape[1], 1))

    if VERBOSE_DEBUG:
        print("spectrogram1: %s, %s, %s" % (type(spectrogram1), spectrogram1.dtype,
spectrogram1.shape))

    # load TF Lite model
    interpreter = Interpreter('simple_audio_model_numpy.tflite')
    interpreter.allocate_tensors()

    # Get input and output tensors.
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()

    #print(input_details)
    #print(output_details)

    input_shape = input_details[0]['shape']
    input_data = spectrogram1.astype(np.float32)
    interpreter.set_tensor(input_details[0]['index'], input_data)

    print("running inference...")
    interpreter.invoke()

    output_data = interpreter.get_tensor(output_details[0]['index'])
    yvals = output_data[0]
    commands = ['go', 'down', 'up', 'stop', 'yes', 'left', 'right', 'no']

    if VERBOSE_DEBUG:
        print(output_data[0])
    print(">>> " + commands[np.argmax(output_data[0])].upper())
    #time.sleep(1)

def main():
```

```
    # create parser
    descStr = """
    This program does ML inference on audio data.
    """
    parser = argparse.ArgumentParser(description=descStr)
    # add a mutually exclusive group of arguments
    group = parser.add_mutually_exclusive_group()

    # add expected arguments
    group .add_argument('--input', dest='wavfile_name', required=False)

    # parse args
    args = parser.parse_args()


    # test WAV file
    """
    if args.wavfile_name:
        wavfile_name = args.wavfile_name
        # get audio data
        rate, waveform = wavfile.read(wavfile_name)
        # run inference
        run_inference(waveform)
    else:
    """
    get_live_input()

    print("done.")

# main method
if __name__ == '__main__':
    main()
```

You can see in the code above how the interpreter is created from the Lite model. We then use the **`set_tensor()`** method to set the spectrogram data into it, and call **`invoke()`** which runs the inference. The **`get_tensor`** call gets us the output data, which is an array of 8 numbers.

We need to pick the one with the highest value, whose index we can get using **`np.argmax(output_data[0])`**. We then use this index to get the command name from a list of the 8 commands.

Here is a typical command line output from a run.

```
opening stream…
Listening…
Stereo detected. Picking one channel.
peak-to-peak: 0.4564. Adjust as needed.
Too silent. Skipping…
Listening…
Stereo detected. Picking one channel.
peak-to-peak: 1.4908. Adjust as needed.
running inference…
>>> STOP
Listening…
```

Above, inference was skipped when nothing was spoken during the 3 second collection interval. Then when "stop" was spoken, it was detected correctly.

2.4 Conclusion

In this lab, we trained an audio recognition model on our computer using TensorFlow, converted it to a TensorFlow Lite model, and used that to infer commands from a live audio stream on a Raspberry Pi.
The inference works quite well in practice, even though we used only a subset of the MNIST data for training, and the training was done only for 10 epochs.

We are at a point where low cost, low power embedded devices are powerful enough to have some intelligence built-in. This project illustrates the general approach for training a Machine Learning / Deep Learning model on a computer, and converting that model for use on a much more resource-constrained system.

It also shows the importance of choosing data formats and processing steps for training that take into account the available capabilities of the embedded device.

2.5 Downloads

All code for this lab can be downloaded from the github link below:

https://github.com/smartcomputerlab/EmbeddedAI

https://github.com/mkvenkit/simple_audio_pi

Table of Contents

Lab 1....1
Running TensorFlow Lite Models on Raspberry Pi....1
1.0 Introduction....1
1.1 Installing TensorFlow Lite to your Raspberry Pi....1
1.2 Download MobileNet....1
1.3 Classifying a Single Image....2
1.4 Conclusion....4
1.5 TensorFlow Lite example apps (pre-trained)....5
1.5.1 Classifying Objects (WebCam)....5
1.5.1.1 Install (setup) the example files....5
1.5.1.2 Run the example....5
1.5.1.3 Complete code:....5
To do:....7
1.2.2 Objects Detection....8
1.2.2.1 Run the example....8
1.2.2.2 Complete code....8
To do:....10
1.2.3 Pose estimation....11
1.2.3.1 Install the dependencies....11
1.2.3.2 Run the pose estimation sample....11
1.2.3.3 Run the pose classification sample....11
1.2.3.4 Complete code....11
1.2.3.5 Customization options....14
1.2.3.6 Visualize pose estimation result of test data....14
1.2.3.7 Complete code....14
1.2.4 Segmentation....17
1.2.4.1 Run the example....17
1.2.4.2 Complete code....17
1.2.5 Video classification....21
1.2.5.1 Run the example....21
1.2.5.2 Complete code....21
To do:....23
1.2.6 Sound classification....24
1.2.6.1 Install Voicecard software....24
1.2.6.1 Python Libraries....24
1.2.6.2 Python Usage....24
1.2.6.3 Run the example....25
1.2.6.4 Complete code....26
Lab 2....28
TensorFlow Lite Speech Recognition....28
Simple audio recognition: Recognizing keywords on a Raspberry Pi using Machine Learning (I2S, TensorFlow Lite)....28
2.0 Introduction....28
2.0.1 Objective....28
2.0.2 The Method....28
2.0.3 Project Architecture....29
2.1 Training the Model....29
2.2 Our Jupyter notebook....32
2.2.1 Setup....32
2.2.2 Import the mini Speech Commands dataset....32
2.2.3 Read the audio files and their labels....33
2.2.4 Convert waveforms to spectrograms....35
2.2.5 Build and train the model....37
2.2.6 Evaluate the model performance....39
2.2.6.1 Display a confusion matrix....39
2.2.7 Run inference on an audio file....40
2.2.8 Saving the model in .tflite format....40
2.3 Inference using TensorFlow Lite on Raspberry Pi....42
2.3.1 Audio from an I2S Microphone on WM8960 audio HAT....42

2.3.1.1 Install software for WM8960 audio HAT....42
2.3.1.2 Python Libraries and usage....43
2.3.2 Processing audio for Inference....44
2.3.3 OLED Display (optional)....46
2.3.4 Putting it all Together....47
2.4 Conclusion....51
2.5 Downloads....51