POLITECNICO DI TORINO

MACHINE LEARNING FOR IoT

HOMEWORK 1

GROUP 14,

BERNAUDO Jacopo,

ERFANMANESH Omid,

VASSALLO Maurizio

REPORT HOMEWORK 1

Ex4) Data Preparation: Sensor Fusion Dataset with TFRecord

The data types we have chosen are:

- **`Int64List:`** `for the datetime in POSIX format,`
- **`FloatList:`** `for temperature and humidity values; even if in the case of the DHT-11 sensors these values are integers (due to sensor precision), it is safer to assume that these values are floating-point. The floating-point variable required 3 bytes more than an integer variable,`
- **`ByteList:`** `for the audio file.`

The audio is read using the **tf.io.read_file()** method, this allows to have both the .wav header (44 bytes) and the audio data so that when reading the .tfrecord it is just needed to read the audio byte part to have the initial .wav file.

The performance (using a .csv with 5 entries. One audio file is 96044B) is:

```
Initial folder size 480405, final .tfrecord size 480785. Ratio 1.0008
```

The size of the .tfrecord is bigger than the initial size due to the overhead added by the creation of the tfrecord itself but this overhead is negligible especially for a large number of entries.

Another solution would be to use the **open()** method from the **wave** library. This function returns only the audio bytes, without the header. This allows to avoid storing on the tfrecord 44 bytes for each audio file; but when reading the tfrecord to recreate the .wav file you need some parameters like duration, frame rate and so on. This option could be useful when the audio files share all the same parameters, like in this case. Anyway we kept the first method for simplicity.
The result using open() function:

```
Initial folder size 480405, final .tfrecord size 480565. Ratio 1.0003
```

Ex5) Low-power Data Collection and Pre-processing

The results obtained running the script with 5 number of samples to collect are:

1.076
1.070
1.070
1.068
1.070
600000 476
...
1500000 72

We used the **BytesIO** class to avoid writing the recording on the disk so as to avoid losing time.
The board frequency is set to powersave mode before the recording and it is set to performance mode one cycle before the end of the recording. That's the position where we found it more suitable to change board frequency also because the changing of frequency requires some time, so the best time is during 'dead' times: after opening the stream and before stopping the stream.

What takes most of the time is the calculation of the STFT, while the other steps require much less time to be executed.

For storing the result we use the **savetxt()** from the numpy library, since it requires less time than the **print(content,file)** method. The result is similar:

- **print(content,file)** saves the content of the mfcc as it is: a sequence of arrays,
- **savetxt()** saves the mfcc values as sequences of numbers separated by a space. Each array is separated by a new line.

The results obtained are compliant with the requirements (latency below 80ms) and the cycles at low frequency are larger than the cycles at high frequency. There are some oscillations in the processing time but this is normal because there may be also other processes running or the time to change frequency may change in different iterations due to different state of the board (temperature, ...).