Group 5 - Report Homework 1

s277757 - Ruben Berteletti
s277771 - Riccardo Baldassa
s276525 - Paolo Fiorio Plà

November 2020

1 Exercise 4: Sensor Fusion Dataset with TFRecord

The aim of this exercise is to create a *TFRecord* dataset, starting from a given folder, selecting the proper datatypes in order to minimize the storage requirements without compromising the original quality of the data. The four fields contained in the *TFRecord* are: datetime (POSIX timestamp), temperature, humidity and audio. To obtain the best result in terms of memory storage and maintenance of original quality of the input data, we have chosen the *Int64List* datatype for datetime, temperature and humidity and the *BytesList* datatype for audio. The use of *BytesList* for the audio is clearly better than use *Int64List* or *FloatList* in terms of memory storage (the *TFRecord* size is respectively 385KB, 830KB and 770KB roughly for 5 samples). For the other three features, the use of *Int64List* or *FloatList* does not create a significant difference in term of size, but the usage of *FloatList* for datetime, leads to wrong representation in the *TFRecord*.

2 Exercise 5: Low-power Data Collection and Pre-processing

The aim of this exercise is to record a number of samples and pre-process them in an iterative way in order to obtain the MFCCS representation as serialized file; all the implementation choices are driven by the (hard) constraint to keep the pre-processing time under 80ms.
In general all the classes required - such as *PyAudio()* and *stream()* - are called only once before the loop and are kept in use for each sample. We applied the same reasoning for *the linear to mel weight matrix* required for the MFCSS representation, which is constant once computed the number of spectrogram bins as `spectrogram.shape[-1]`.
As second optimization we exploited the *BytesIO()* class for the buffer managing, that becomes useful during the recording, where instead of writing the *.wav* file after the recording, we kept it in memory. This philosophy (keeping the intermediate results in memory, without writing them on disk) has been adopted for the entire pipeline.
Our focus has been to reduce as much as possible the time elapsed in performance mode, but in any case with a setting that must achieve 1.080s for the entire process. To do so, we exploited the *Popen()* class in order to modify the clock frequency in two different moments, but both during the recording phase exploiting the 1s of acquisition, since switching frequency requires some milliseconds and then we avoid to affect the total execution time. The first instance has been positioned immediately after the stream starts, aimed to set for the acquisition the *powersave* mode, since we were bound by 1s of recording where no computations were needed; the second, instead, has been inserted during the stream loop in order to set the *performance* mode, in particular here the activation is triggered when the last i chunks are recording (where i is chosen with an ad-hoc variable called `performance_mode_trigger`). We decided for this variable approach since we noticed for different boards (actually even for the same model) different behaviours and then, to be sure to remain within 1.080s we suggest to trigger the performance mode while the recording for the second-last chunk starts for the old raspbian (Aug 2020 version) and when the last one starts for the new software (Oct 2020 version). What we have seen is that the bottleneck in this pipeline is the *STFT* that is then mandatory to run in performance.
On average we satisfied the constraint with 4.8s in powersafe and 0.6s in performance (for 5 samples).