Machine learning for IoT
Homework 1 report, Group 11

1 Exercise 4

In this exercise we created a TFRecord dataset starting from a csv file, whose rows contain date, time,values of temperature and humidity and the name of a .wav audio file stored in the same folder as the csv file. The requirement is to minimize the storage requirements, while maintaining the original quality of the data. We used the wave api to read the wav files, creating a wave_read object whose method readframes returns the raw audio data in bytes format. Then we created the Features corresponding to our variables and defined the dictionary of features, which was serialized and inserted in the tfrecord dataset with the tf.train.Example Class.
The challenge of this exercise was choosing the data type for the audio data, so that it requires as little storage space as possible. We observed that by storing them as bytes objects the storage space used was less than the space used when storing them as a float list (using tf.audio.decode_wav). To verify the validity of our implementation we compared the dimension of the wav files with that of the output .tfrecord file. We observed that in the case of 4 audio files that weigh 377.008 bytes collectively, the tfrecord file weighs 377.260. The slight difference is explained by the fact that the structure of the tfrecord file occupies some storage space itself.

2 Exercise 5

This exercise simulates a real time data processing task, that in our case is the recording of 1 second audio sample followed by a set of preprocessing operations in order to extract features from each audio sample and save them in a bin format file. The main objective of the exercise is to remain within a delay threshold of 80 ms for the preprocessing phase while minimizing the power consumption through Voltage Frequency scaling. The code is organized in three different sections: a preliminary set of operations which includes the input arguments parsing and the declaration of all the elements that need to be specified only once within the whole process (useful variables, *pyaudio.PyAudio()* instance, *stream* instance with *start* parameter set to False, *linear_to_mel weight matrix*, *performance monitor* resetting; the main for loop cycle where audio sample is collected, and the preprocessing operations are carried out; a final set of operations which includes stream closing, pyAudio instance termination and results printing. The monitor reset operation is performed with *subprocess.call()* in order to be sure it is already executed when the main for cycle starts. The most computationally expensive phase is the preprocessing phase, mainly due to the computation of the *stft* and the dot product operation. In order to respect the time constraint and limit the consumption the preprocessing part must be executed in performance mode, which must be not too earlier effectively activated when this part starts. The performance mode activation time is about 90 ms, measured exploiting *subprocess.call()* and *time* library (in the final version textitsubprocess.Popen() is used to speed up this step). With a chunk sample rate of 48 KHz and a chunk size of 4800, the performance mode activation is inserted before the last chunk is read. The powersave mode is activated just before the first chunk reading, resulted to be the best moment in order to remain below the time threshold and actually record a 1 sec audio sample (activating it between the stream start and the chunks for-loop resulted in reading a shorter audio sample). The BytesIO class used for storing the audio sample does not seem to significantly improve the performances. Pay attention that the first iteration just after the Raspberry is turned on systematically takes much more time, while all the following runs give comparable results to those reported in the assignment document.