Report Homework 1

Machine Learning for IOT

Authors:

Antonio Dimitris Defonte s276033
Eleonora Carletti s280056
Filippo Cortese s273672

Exercise 4

In this exercise we built a *TFRecord* Dataset trying to minimize the storage requirements while keeping the original quality of the data. To accomplish this aim we had to decide the proper data-type for each feature that we wanted to store. For the datetime feature, after obtaining with the *time.mktime* method a float type datetime in posix format, we decided to cast it as an integer and consequently store it as an *Int64List* in the *TFRecord*. This was the theoretically more coherent choice and also in terms of size: in fact it was the one that in the end occupied less space without losing any information. For the temperature and humidity attributes we used the same reasoning of before and, since they were integer value, after casting them as integer variable we stored them using *int64list* data-type. Also in this case using other data-types resulted in bigger file sizes. In the end we had to decide how to store the wav audio file, so we took in account different methods. The first option was to decode the wav file through the *scipy.io.wavfile.read* method that return a numpy array with same data-type determined from the file (an *int16* in our case) and stores it in the tf record as *Int64List*. The second option was to read the wav file with *tf.io.read_file* method of tensorflow and then decode the raw audio with the *tf.audio.decode_wav* method that return a normalized *float32* tensor with the audio data and store it as a *FloatList* in the *TFRecord*. The last option, that in the end was the best as it can be seen in the table, was to simply read the wav file with the *tf.io.read_file* method that return the wav encoded audio as a string tensor and store it in the tf.record with a *BytesList* data-type. This method is able to significantly reduce the size of the TFRecord but requires a further step in the process of reading it : you have to apply the *tf.audio.decode_wav* method to get the audio samples content as a float tensor.

	tf.audio.decode_wav()	scipy.wavefile.read()	tf.io.read_file()
Size(Bytes)	960535	917687	480755

Exercise 5 (OS version: 5.4.72-v7l+ Thu Oct 22)

In this exercise, we had to sample iteratively one second audio and to preprocess it in order to save its mfcss, with a time threshold of *80 ms* for the preprocessing and saving. We exploited the *DVFS* power management strategy for executing low time operations at *Vfmin* and high time consuming operations at *Vfmax* (respectively 0.81V *a*t 600MHz and 0.86V *a*t 1.5GHz*).* At the beginning and at the end of the script, we switched to the power-save operating mode for consistency, to be sure that the *raspberry* started and finished the execution of the script in this mode.
We need to respect the threshold and at the same time minimize the time at *VFmax*. Since the policy transition is done in parallel with the command *Popen*, we estimated how much time was needed to switch between the 2 modes with the *check_call* (waits for termination, unlike *Popen*) function in a separate file. The time needed to transition between performance and powersave was around 25-35 ms, while the inverse was around 48-65ms.
So in order to respect the constraint we had to identify the correct place to put the commands for the voltage scaling. The first one is placed at the beginning of each sample iteration to execute the recording in powersave mode, and the second one that changes from powersave to performance is placed in the recording phase, since it is the only window where we have time to wait for the command to be executed.

To record the audio from the microphone we used a for loop, so choosing the right chunk size is fundamental since there is an iteration for each chunk of audio. We have chosen a number of chunks equal to 10 (chunk size equal to 4800) because, given that an iteration of this for loop lasts approximately 1000/10 ms, in the last one we had 100 ms to transition to performance mode. This has a bit of margin so we can stay under the 80 ms threshold and do the computational intensive operations in performance mode. We can achieve the same result if we put 20 chunks and switch to powersave in the second-last iteration, in order to have again around 100 ms to perform this operation.
In conclusion, the resampling, *STFT*, *MFCC* and saving were the most expensive operations (with the *STFT* on the lead, taking up to ⅔ of the total time for preprocessing) and were done in performance mode while it was feasible to record the audio in power-save policy, switching to the performance one in the last iteration (because as we mentioned changing policy is not instantaneous). All the constant initializations (like the matrix for *MFCCs*) were done before the sampling iterations.
This was accomplished using the previously mentioned OS version, but it can be easily adopted to the previous ones. Using the same reasoning, we would need to optimize the chunk size and at which iteration change policy, since changing policy would require more time.

We also managed to respect the threshold with other chunk sizes, but they will usually make the board stay more in the *VFmax* regime. In the table below there are some results on 5 samples of audio with different chunk sizes. We show the time (in 10ms) at the different frequencies, the average needed time (in ms) for recording and processing the samples and the number of the iteration at which we switch to performance mode. The time (ms) for 5 samples sampled with 10 chunks (green row) was: 1071.19; 1065.28; 1065.09; 1064.69; 1064.9. Usually in this setting the board stays at *VFmax* for 550-650 ms (total).

# chunks	time (10ms) at *VFmax*	time (10ms) at *VFmin*	average time (ms)	iteration for *Popen*
8	67	465	1066.17	8th
10	55	476	1066.23	10th
5	102	430	1066.25	5th
12	89	443	1067.04	11th