Group 2 — Homework I

Alessio Vacca
Politecnico di Torino
s288240@studenti.polito.it

Luca Benfenati
Politecnico di Torino
s286582@studenti.polito.it

Nicola Scarano
Politecnico di Torino
s287908@studenti.polito.it

***Abstract*—The following report addresses two different exercises related to the storing, the pre-processing and the feature extraction of data which take place prior to the machine learning pipeline. These early stages are crucial to the development of an efficient inference model that aims to be deployed on IoT edge devices.**

I. Exercise I

For the first exercise we are asked to build a TFRecord Dataset containing temperature and humidity measurements at different datetimes. Other than giving the chance to normalize measurement values, we need to address the choice of the data type for each field in order to minimize the storage requirements maintaining the original quality of data. In particular, as fields we have datetime, which needs to be stored as POSIX timestamp, temperature, which is an integer included in $[0, 50]$ °C and humidity, an integer in $[20, 90]\%$ RH, as we can see from the data-sheet [1].

To build our TFRecord we select the following datatypes:

- tf.train.Int64List for the POSIX timestamp,
- tf.train.Int64List for temperature and humidity when normalization is not considered,
- tf.train.FloatList for temperature and humidity when normalization is required.

For what concerns the POSIX timestamp, we notice that, even if tf.train.FloatList would have required less storage memory, the constraint for which we need to keep the same quality of the original data was not met. When considering measurements collected every second (i.e. frequency 1), we observed that tf.train.FloatList conversion could not represent the original value with the same precision. Indeed, employing this type, the original value of the POSIX timestamp is wrongly approximated in the tf.train.Feature, thus resulting in a loss of quality of data. This does not happen if we use either tf.train.Int64List and tf.train.BytesList. However, in order to minimize the storage requirements, we exploit tf.train.Int64List to store this information.

For what concerns temperature and humidity, the choice of the data type is conditioned by whether the normalization is required or not: if normalization is not needed values are stored with tf.train.Int64List to spare some space, whilst normalized values are store through tf.train.FloatList to necessarily address floating point values. The results of the toy example provided in the text is reported in table I

TABLE I
SIZE OF OUTPUT TFRECORD

TFRecord	232 Bytes
Normalized TFRecord	256 Bytes

Assuming that normalization is required by the training and inference pipelines, the best solution is probably to leave the normalization step after the TFRecord setup, including it into the model pipeline. This allows us to exploit the tf.train.Int64List data type, thus saving up space and speeding up transmissions: smaller files will result in faster data movements among edges and cloud. This become crucial especially when dealing with large datasets, where the gain in terms of bytes between normalized and non-normalized TFRecords gets bigger. If we deploy the normalization step in the model, we can leverage keras layers which already provide this preprocessing computation, as for example a batch normalization layer. Although these considerations, there is no general rule to establish which is the best suited solution, and both alternatives should be taken into account.

II. Exercise II

The second exercise of the homework considers a dataset of 2000 Yes and No recordings. We are asked to compute for each sample the Mel-frequency cepstral coefficients (MFCCs) in two different ways: one with the parameters provided in the text (which we will call $MFCC_{slow}$) and another one ($MFCC_{fast}$) where we modify the pipeline so to decrease the execution time and guarantee a certain value of the Signal to Noise Ratio (SNR), satisfying the given constraints.

First of all, we build a pipeline to compute the MFCCs testing the standard configuration: we are able to compute the MFCC for all the 2000 samples in $26.22s$.

In the second part of the exercise we change the pipeline adding a re-sampling step to reduce the required computation time. Moreover, we test multiple configurations of the parameters below in order to satisfy the constraints. The parameters taken into consideration are:

- re-sampling ratio,
- number of Mel bins,
- lower frequency,
- upper frequency.

Starting from the standard configuration, we try to find the best one exploiting an exploratory approach combined with technical knowledge of the field, thus optimizing the overall performances.

TABLE II
PARAMETERS

	rate	mel bins	low freq	up freq
MFCC slow	16000 Hz	40	20 Hz	4000 Hz
MFCC fast	4000 Hz	32	20 Hz	2000 Hz

TABLE III
RESULTS

MFCC slow	26.22 ms
MFCC fast	17.43 ms
SNR	11.44 dB

Given that the original signals have sampling rate 16kHz, we downsample the rate to 8kHz and then 4kHz (i.e. respectively with a re-sampling ratio of $1/2$ and $1/4$). As expected, the overall execution time decreases. However, the first part of our pipeline, which includes reading, now increases in time with respect to the original since the re-sampling phase is now required. On the other hand, the time for the STFT computation decreases as fewer number of samples are extracted from the file: indeed, we notice that the STFT has smaller size. The MFCC computation time, instead, remains approximately the same: the only operation which is faster is the computation of the weight matrix used to re-weight linearly the spectrogram, but since it is computed only once for all the samples the gain in time is marginal.

For what concerns the number of Mel bins, there is no general procedure to find the most suitable value, but it depends from case to case. We experience that, starting from the standard value of 40 Mel bins, an increase of such value corresponds to an increase in the execution time and a decrease in the SNR value. The best values is reported in the table II.

Lower and upper frequencies are, respectively, the lower and the upper bound to be included in the Mel-frequency cepstrum. To find the most suited interval of frequencies we fine-tune these two bounds. Decreasing the lower frequency from the default value results in a more noisy spectrum, thus worsening the SNR, while increasing it results in a gain in time and a reduction in the SNR. This behavior is probably due to the fact that lower frequencies are noisier, instead, considering an higher lower bound implies leaving out some important information. For what concerns the upper frequency, its value depends on the sampling rate. In particular, it must satisfy the constraint given by the Nyquist theorem, for which the upper frequency must be less than half of the sampling rate (i.e. $f_{UB} \leq \frac{1}{2} f_{sampling}$). Therefore, considering for example a re-sampling ratio equal to $1/4$, the upper frequency cannot be higher than $4000/2\ Hz = 2000\ Hz$. So, if the increment is bounded by this theorem, the decrease results in a loss of information, as in the previous case, thus worsening the SNR.

The best results that satisfy all the given constraints are reported in the table III.

We highlight the fact that we have not modified frame length, frame step and the number of the cepstral coefficients, since changing these parameters results in a different shape of the MFFCs tensors, thus failing to be compliant with one of the constraint.

REFERENCES

[1] "DHT11 Humidity & Temperature Sensor datasheet", Mouser Electronics