Machine Learning for IoT

Homework 1

*****DUE DATE: 22 Nov (h23:59)*****

Submission Instructions:

Each group will send an e-mail to andrea.calimera@polito.it and valentino.peluso@polito.it (in cc) with subject <ML4IOT22 GroupN> (N is the group ID). Attached with the e-mail the following files:

1. One single .py file for each exercise, titled <HW1_exM_GroupN.py>, where M is the exercise number and N is the group ID, containing the Python code. The code must use only the packages that get installed with *rpi_requirments.txt* and *requirements.txt*.
2. One-page pdf report, titled <GroupN_Homework1.pdf>, organized in different sections (one for each exercise). Each section should motivate the main adopted design choices and discuss the outcome of the exercise.

Late messages, or messages not compliant with the above specs, will be automatically discarded.

HowTo 1: Writing TFRecord Datasets

Data movement represents one of the major bottlenecks in neural network training and inference. Indeed, standard machine learning pipelines encompasses the transfer of a massive amount of data from the main storage to the computational units, e.g., from the hard drive to the GPU memory. Fetching data from large storages is slow and energy costly. To mitigate this issue, raw data should be packed in serialized sequences of binary records. In this way, the data will reside in consecutive locations of memory, thus accelerating parallel fetching operations.

The TensorFlow Data API introduced the *TFRecord* format (see *tf.train.TFRecord*) to enable users to easily serialize structured data. Specifically, the user needs to specify:

- the structure of each record through the definition of a key-value based dictionary, called *Example* (see *tf.train.Example*);
- the value for each key of the record, i.e. the *Feature* (see *tf.train.Feature*);
- the data-type for each value (see *tf.train.BytesList*, *tf.train.FloatList*, *tf.train.Int64List*)

The following example builds a *TFRecord* Dataset composed by 10 records of random float and integer numbers:

```
filename = './numbers.tfrecord'
x = np.random.randint(0, 5, 10)
y = np.random.randn(10)

with tf.io.TFRecordWriter(filename) as writer:
   for i in range(10):
      x_feature = tf.train.Feature(int64_list=tf.train.Int64List(value=[x[i]]))
      y_feature = tf.train.Feature(float_list=tf.train.FloatList(value=[y[i]]))

      mapping = {'integer': x_feature,
                 'float'  : y_feature}

      example = tf.train.Example(features=tf.train.Features(feature=mapping))

      writer.write(example.SerializeToString())
```

For more information, check the official documentation and tutorial:
https://www.tensorflow.org/versions/r2.3/api_docs/python/tf/io/TFRecordWriter
https://www.tensorflow.org/tutorials/load_data/tfrecord

Exercise 1: Temperature and Humidity Dataset with TFRecord (3pt)

Write a Python script on your Raspberry to build a *TFRecord* Dataset containing temperature and humidity collected at different datetimes and stored as raw data in a csv file as shown in following example:

```
18/10/2021,09:45:34,21,65
18/10/2021,09:46:40,21,65
18/10/2021,09:47:45,21,65
18/10/2021,09:48:51,21,65
```

N.B.: The *csv* file is an input of the script. The script should contain only the code to build the *TFRecord* Dataset.

Each *Example* in the *TFRecord* must have three fields:

- datetime (stored as POSIX timestamp);
- temperature;
- humidity;

No assumptions must be done on the number of records contained in the dataset.

Add an optional *normalize* command-line argument to the script. If the *normalize* flag is enabled, the script should normalize temperature and humidity before storing the data. Apply min/max normalization to each sample.

For example, to normalize temperature (T):

$$T_{norm} = \frac{T - T_{min}}{T_{max} - T_{min}}$$

Depending on the use-case (with or without *normalize)*, for each field select the datatype that enables to minimize the storage requirements while keeping the original quality of the data. Motivate your choices (in the report).

Hint: Check the min./max. ranges and resolutions from the DHT-11 datasheet: https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf.

At the end of the script, print the size in bytes of the generated *TFRecord*.

SYNOPSIS:

```
python HW1_ex1_GroupN.py --input <INPUT FILE> --output <OUTPUT FILE> [--normalize]
```

EXAMPLE:

```
python HW1_ex1.1_Group1.py --input ./th.csv --output ./th.tfrecord
```

OUTPUT (on screen):

```
328B
```

Run the script with and without the *normalize* flag and comment the results (in the report).
Assuming that normalization is required by the training and inference pipelines, describe (in the report) the best solution to write, read, and process the *TFRecord* samples such that the storage requirements are minimized.

Exercise 2: Audio pre-processing optimization (3pt)

Download the Yes/No Dataset from the *Portale* to your Raspberry Pi. This dataset contains 1-second recordings of yes/no keywords sampled at 16kHz and Int16 resolution.

Write a Python script that iteratively reads the audio files from the Yes/No dataset and processes the MFCCs of each recording:

- For the STFT, set frame length=16ms, frame step=8ms.
- For the MFCC, set #Mel bins=40, lower frequency=20Hz, upper frequency=4kHz, #MFCCs=10.

Measure the average execution time of each iteration.
We denote the output of this pre-processing pipeline as $MFCC_{slow}$.

Add to the script a new pre-processing pipeline to extract the MFCCs ($MFCC_{fast}$) such that all the following constraints are met:

- the shape of the $MFCC_{fast}$ is equal to that of $MFCC_{slow}$
- the average execution time of the new pre-processing pipeline is <18ms
- the signal-to-noise ratio (SNR) between the $MFCC_{fast}$ and $MFCC_{slow}$ is >10.40dB (on average over all the recordings of the dataset).

$$SNR = 20\ \log_{10}\left(\frac{\|MFCC_{slow}\|}{\|MFCC_{slow} - MFCC_{fast} + 10^{-6}\|}\right)$$

Hint #1: Use the *numpy.linalg.norm* method to compute the ||·|| operation: https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html.

Hint #2: Add the following lines of code at the beginning of your script to guarantee stable measurements of the execution time:

```
from subprocess import Popen

Popen('sudo sh -c "echo performance >'
      '/sys/devices/system/cpu/cpufreq/policy0/scaling_governor"',
      shell=True).wait()
```

SYNOPSIS:

```
python HW1_ex2_GroupN.py
```

OUTPUT (on screen):

```
MFCC slow = 30 ms
MFCC fast = 5 ms
SNR = 11.56 dB
```

N.B.: The numbers reported in this example are not the expected ones!

In the report, summarize and motivate the parameters selected to meet the constraints of the fast pre-processing pipeline.