Machine Learning for IoT

Homework 1

*****DUE DATE: 17 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 <ML4IOT21 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.

Background 1: Sensor Fusion

Many prediction tasks can benefit from exploiting data originating from different sources. We refer to this kind of applications with *sensor fusion*. Some popular examples are: (i) scene classifications, which relies on image, audio, and temperature to classify the environment (indoor or outdoor); (ii) position and velocity estimation, which combines several inertial sensors (accelerometer, gyroscope, magnetometer, altimeter); (iii) activity recognition, which can complement measurements from inertial sensors with electrocardiography and electromyography data.
Inspired by humans that use the five basic senses to sense and process information, sensor fusion relies on the hypothesis that the analysis of multiple sources of data improves the prediction accuracy. In sensor fusion applications, the training pipeline is therefore fed with structured data. An efficient implementation requires to organize and prepare the data in order to create datasets compliant with the specifications of standard training frameworks.

HowTo 2: TFRecord Datasets

Data movement represents one of the major bottlenecks in neural network training. Indeed, standard training 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/api_docs/python/tf/io/TFRecordWriter
https://www.tensorflow.org/tutorials/load_data/tfrecord

Background 3: Power management with Dynamic Voltage Frequency Scaling

Dynamic Voltage Frequency Scaling (DVFS) is today a standard power management strategy for System-on-Chips (SoCs). It relies on the observation that the power consumption of a digital circuit has a quadratic dependance from voltage (V_{dd}) and a linear dependance from clock frequency (f_{clk}). Voltage Frequency (VF) scaling therefore enables a cubic reduction of power.

$$P \propto V_{\mathrm{dd}}^2 \cdot f_{clk}$$

To limit the power consumption, the SoC should run at maximum performance (VF_{max}) only for short periods, e.g. to meet tight timing constraints in critical and intensive tasks. Otherwise, the SoC can operate into a low-power state (VF_{min}) and save power without impact on the user experience.

HowTo 4: DVFS on the Raspberry Pi

On Unix-based operating systems (e.g. Raspian), the DVFS is managed by an operating system routine called governor. The standard governor can select two main policies: *performance* and *powersave*. In *performance* mode, the VF level is set to the maximum value VF_{max}=0.86V @ 1.5GHz, whereas in *powersave* mode, it is set to the minimum value VF_{min}=0.81V @ 600MHz.

To manually change the operating mode, modify the content of the *scaling_governor* system file:

```
# Set 1.5GHz
sudo sh -c "echo performance > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor"

# Set 600MHz
sudo sh -c "echo powersave > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor"
```

To check the current VF, read the *file* system file, that stores the operating clock frequency in Hertz:

```
cat /sys/devices/system/cpu/cpufreq/policy0/scaling_cur_freq
```

To monitor the operating time at different VF levels, the operating system provides performance monitors, that work as follows:

```
# Reset the monitors
sudo sh -c "echo 1 > /sys/devices/system/cpu/cpufreq/policy0/stats/reset"

# Read the monitors
cat /sys/devices/system/cpu/cpufreq/policy0/stats/time_in_state
```

Note that there exists intermediate VF levels and other governor policies, which will be not considered in this homework.

Exercise 4: Data Preparation: Sensor Fusion Dataset with TFRecord (3pt)

Write a Python script on your notebook to build a *TFRecord* Dataset containing temperature, humidity, and one-second audio samples (Int16 resolution at 48KHz), collected at different datetimes and stored as raw data in a single folder as shown in following example:

```
./raw_data
  samples.csv
  audio1.wav
  audio2.wav
  audio3.wav
  audio4.wav
```

Content of *samples.csv*:

```
    18/10/2020,09:45:34,21,65,audio1.wav
    18/10/2020,09:46:40,21,65,audio2.wav
    18/10/2020,09:47:45,21,65,audio3.wav
    18/10/2020,09:48:51,21,65,audio4.wav
```

N.B.: The *raw_data* folder 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 four fields:

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

Select the proper datatype depending on the field in order to minimize the storage requirements while keeping the original quality of the data. Motivate your choices.
No assumptions must be done on the number of records contained in the dataset.

SYNOPSIS:

```
python HW1_ex4_GroupN.py --input <INPUT DIR> --output <OUTPUT FILE>
```

EXAMPLE:

```
python HW1_ex4_Group1.py --input ./raw_data --output ./fusion.tfrecord
```

<u>Exercise 5:</u> Low-power Data Collection and Pre-processing (3pt)

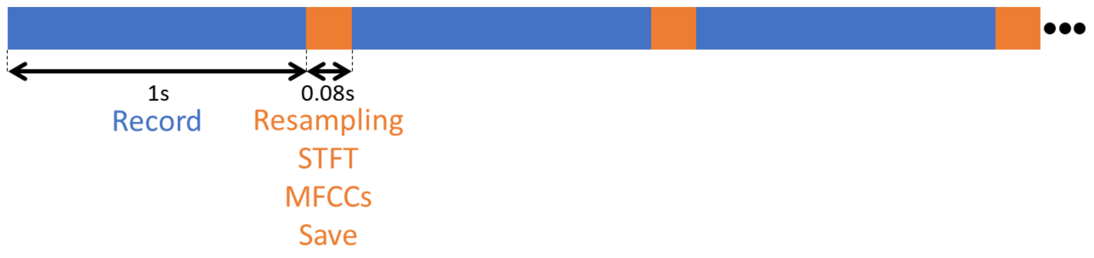
Write a Python script that iteratively samples 1-second audio signals, process the MFCCs, and store the output on disk. The script takes as input the number of samples to collect and the output path where to store the MFCCs.
Set the resampling, STFT, and MFCCs parameters to the same values as Lab 2 – Ex6.
The pre-processing latency must be <80ms (hard constraint). Identify which steps of the acquisition&pre-processing loop can be performed @VF_{min} such that the time @VF_{min} is maximized and the pre-processing latency is met.
The script should print the latency of each iteration of the loop and the total time spent at the different VF levels. Comment the results.

<u>Suggestion #1:</u> Avoid writing intermediate data on disk to reduce the processing time (check the *BytesIO* class from the *io* package).

<u>Suggestion #2:</u> Use the *Popen* class from the *subprocess* package to run DVFS commands in non-blocking mode.

SYNOPSIS:

```
python HW1_ex5_GroupN.py --num-samples <NUM SAMPLES> --output <OUTPUT>
```



EXAMPLE:

```
python HW1_ex5_Group1.py --num-samples 5 --output ./out
```

Output:

```
1.054
1.069
1.068
1.069
1.068
600000 465
750000 0
1000000 0
1500000 64
```

Content of the *out* folder:

```
./out
  mfccs0.bin
  mfccs1.bin
  mfccs2.bin
  mfccs3.bin
  mfccs4.bin
```