Machine Learning for IoT

Homework 3

*****DUE DATE: 31 Jan (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 .zip file for each exercise, titled <HW3_exM_GroupN.zip>, where M is the exercise number and N is the group ID, containing the code (more details provided later in the text). The code must use only the packages that get installed with *requirements.txt* and *rpi_requirements.txt*.
2. One-page pdf report, titled <GroupN_Homework3.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.

Exercise 1: Model Registry

when you generate new models you need a place where to store them and a safe infrastructure to model them.

A Model Registry is a repository to store and manage trained ML models. Model Registries generally comprise the following components: (i) object storage to hold model files (e.g., *tflite)* (ii) model monitoring to monitor the model predictions on new data (iii) a programmatic API that enables users to interact with the registry.

- On your Raspberry Pi, develop a Model Registry (*registry_service.py)* to manage models for temperature and humidity forecasting. The Model Registry must offer the following services through a RESTful API:

  1. Store a *tflite* model in a local subfolder called *models* located at the same path of the service script.

since we need to send data in the body we cannot use get, put PUT

REQUEST

PATH	/add
PARAMETERS	any
BODY	**model:** (base64 str) the *tflite* model **name:** (str) the model name

RESPONSE

CODE	200 (if successful) or 400 (otherwise)
BODY	any

2. List the names all the models stored in the *models* folder.

we just need to retrive information, we don't need to send info —> GET

REQUEST

PATH	/list
PARAMETERS	any
BODY	any

RESPONSE

CODE	200 (if successful) or 400 (otherwise)
BODY	**models:** (list of strings) list containing the names of the models stored in the *models* folder

Alert are usually asynchronous communications —> MQTT

+ "multiple remote clients": the easiest way to send message to multiple clients is to use Mqtt

3. Measure temperature and humidity values with the DHT-11 sensor every 1s. As soon as 6 samples are measured, make a prediction with the model indicated in the request every 1s. Compare the measured vs. the predicted value. If the prediction error (absolute error) of temperature (humidity) is greater than a temperature (humidity) threshold, send an alert to a single or multiple remote clients using the SenML+JSON format. **Identify and adopt the most suitable communication protocol (between REST and MQTT) to send the alert and motivate your choice in the report.**

if the threshold are not respected we have to send an ALERT

REQUEST

PATH	/predict
PARAMETERS	**model:** (str) the name of the model **tthres:** (float) the temperature threshold in °C **hthres:** (float) the humidity threshold in %
BODY	any

The request to predict should be done with REST or MQTT?

RESPONSE

CODE	200 (if successful) or 400 (otherwise)
BODY	any

For each service, identify the proper HTTP method (among GET, POST, PUT and DELETE) and motivate your choice in the report. Moreover, manage possible errors in invoking the web services (e.g., wrong paths or wrong number of parameters).

SYNOPSIS:
```
python registry_service.py
```

- On your notebook, develop a client application (*registry_client.py*) that:
  1. Add the MLP and CNN *tflite* models developed in LAB3-Ex1 to the Model Registry.
  2. List the models stored in the Model Registry and verify that their number is two.
  3. Invoke the registry to measure and predict temperature and humidity with the CNN model. Set tthres=0.1 and hthres=0.2.

SYNOPSIS:
```
python registry_client.py
```

- On your notebook, develop a client application (*monitoring_client.py*) that receives the alerts and prints them on the screen according to the following format:

  (*day/month/year hour:min:sec*) *Quantity* Alert: Predicted=*value* Actual=*value*

  SYNOPSIS:
  ```
  python monitoring_client.py
  ```

  EXAMPLE OUTPUT:

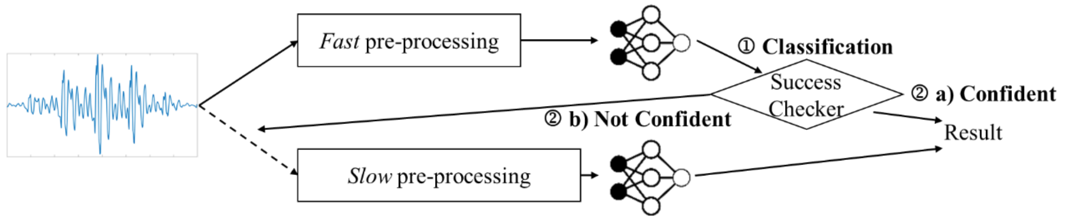
  ```
  (05/12/2022 19:15:01) Humidity Alert: Predicted=48.8% Actual=50.1%
  (05/12/2022 19:15:01) Temperature Alert: Predicted=22.0°C Actual=21.5°C
  (05/12/2022 19:17:07) Temperature Alert: Predicted=21.8°C Actual=21.4°C
  ```

- The final submission should contain the following files:
  1. The *registry_service.py* script.
  2. The *registry_client.py* script.
  3. The *monitoring_client.py* script.
  4. Any other .py file needed for the correct execution of the previous commands (if any).

  Organize the files in two different folders, named *notebook* and *rpi* respectively, depending on the target device where the commands should run.

<u>Exercise 2:</u> Edge-Cloud Collaborative Inference

Collaborative Inference is a computational paradigm where the inference workload is distributed across interconnected devices to improve the prediction accuracy. Different devices host different inference pipelines, with complexity and accuracy proportional to their hardware resources. Collaborative Inference frameworks generally comprise the following components: (i) a *Fast* inference pipeline running on edge (ii) a *Slow* inference pipeline running on the cloud (iii) a "success checker" policy to determine whether the *Fast* inference was "confident" about its prediction or not; if not, run the *Slow* inference to get the final prediction.



Develop a Collaborative Inference framework for Keyword Spotting composed of a *Slow* inference pipeline running on your notebook and a *Fast* inference pipeline running on your Raspberry Pi. Both pipelines run the same model (same architecture and same weights) deployed in *tflite* format, but with different pre-processing. Use the *kws_dscnn.tflite* model on the Portale (folder Homework3).

The *kws_dscnn.tflite* model has been trained on the mini speech command dataset with sampling frequency=16kHz, resolution=16-bit, frame length=40ms, frame step=20ms, #Mel bins=40, lower frequency=20Hz, upper frequency=4kHz, #MFCCs=10, and the label ordering indicated in the *labels.txt* file on the Portale.

The overall system must be composed by:

- A web service (named *slow_service.py*) that receives a **raw** audio signal (sampling frequency is 16 kHz) in SenML+JSON format, runs inference using the *Slow* pipeline and returns the output label (in JSON format).

  SYNOPSIS:
  ```
  python slow_service.py
  ```

- A client application (*fast_client.py)* that sequentially reads an audio signal from the mini speech command test-set (the signals are listed in *test_files.txt* from the *Portale*), runs inference using the *Fast* pipeline and, if needed, invokes the web service to get the prediction from the *Slow* pipeline. Define and implement the "success checker" policy that decides whether to accept the prediction of the *Fast* pipeline or invoke the web service.
  SUGGESTION: To implement the "success checker" policy, monitor the SoftMax of the *Fast* inference output.

  The client must print on the screen the accuracy and the communication cost measured on the mini speech command test-set. The communication cost is the sum among the sizes of the SenML+JSON strings sent to the web service.

  SYNOPSIS:
  ```
  python fast_client.py
  ```

  EXAMPLE OUTPUT:
  ```
  Accuracy: 91.125%
  Communication Cost: 1.983 MB
  ```

  Identify and adopt the most suitable protocol (between REST and MQTT) to implement the communication between the two applications. Motivate your choice in the report.

- Design the *Slow* and *Fast* pre-processing pipelines and the "success checker" policy such that the following constraints (measured on the **test-set**) are met:
  - ✓ Collaborative Accuracy > 91%
  - ✓ *Fast* Total Inference Time < 40 ms
  - ✓ Communication cost < 2.0 MB

- The final submission should contain the following files:
  1. The *slow_service.py* script.
  2. The *fast_client.py* script.
  3. Any other .py file needed for the correct execution of the previous commands (if any).

  Organize the files in two different folders, named *notebook* and *rpi* respectively, depending on the target device where the commands should run.

- In the report, describe the *Slow/Fast* pre-processing pipelines and the "success checker" policy adopted.