Machine Learning for IoT - Homework 3

Luca Campana
Student id: s290065

Gianvito Liturri
Student id: s290464

Alberto Morcavallo
Student id: s290086

Exercise 1: Model Registry

To address this task, we have implemented the three applications that accomplish their job in the following way:

- `registry_service.py`: this script sets up the needed services through HTTP methods. In particular, the first service needs to receive from the client the whole *tflite* model, so it has been developed through a POST method; on the contrary, the others, that only require a kind of data retrieval, have been implemented through the GET method. We need to make a further clarification: for the third service, the GET method has the only function of passing the name of the model to use plus the values of thresholds and making the sensing of data and forecasting start. Then, alerts are sent back using MQTT protocol, for which we create a publisher node that only sends these messages into the *alert* channel, in the required format. This choice has been done due to the fact that MQTT protocol is much better in handling short textual message exchanges, thanks to its speed of communication; moreover, the task structure is easier to be implemented with a communication of the asynchronous type. This also allows to potentially implement the delivery of the alerts to multiple clients in an easy way, since the nature of *publisher/subscriber* communication allows to have multiple instances that subscribe to the same channel.
- `registry_client.py`: this has been designed in order to execute the needed tasks in sequential order. Firstly, the two wanted models are read from the local storage, they gets encrypted as strings of bytes and sent to the server as the body of the POST request. Then, the client starts the second service and receives the list of models available in the server. After having checked that there are two available models, another GET request is sent to communicate the wanted model and threshold values and make the inference part start.
- `monitoring_client.py`: in this script, all we do is creating a subscriber node that senses what gets published on the *alert* topic. From each message, we take the needed quantities (date, hour, typology of the alert, true quantity, prediction), format them accordingly to the given example and print the result on screen.

One last specification to be made regards the fact that, since we implemented a MQTT connection, we must include the scripts `DoSomething.py` and `MyMQTT.py` on both sides of the communication, after having opportunely modified them to handle their outputs.

Exercise 2: Edge-Cloud Collaborative Inference

For this second task we had to develop a RESTful communication. We choose this type of communication for two main reasons: if we had made it through MQTT, we would have required one publisher and one subscriber node for each side of the communication; moreover, the fact of having asynchronous communication would have been harder to manage, dealing with the sequentiality of the process pipeline. In addition, we noticed that none of the constraints to obtain was influenced to the communication protocol, so we choose to reason independently from them. In order to respect the given constraints, the applications have been implemented in this way:

- `fast_client.py`: here, we manage the sequential analysis of audio tracks. Each of them gets read, preprocessed in a fast way, then submitted to the DSCNN. If the normalized prediction cannot reach the wanted value of confidence, we send the track to the server in SenML+JSON format, and wait for its response. At the end, we store the prediction, other than the time needed for the fast pipeline and the amount of data sent to the server. The communication has been implemented as a single POST method, summarized in the tables I, II.
  The fast preprocessing has been obtained by: resampling tracks to $8kHz$ and casting to $float32$, extracting windows with size 256 and stride 128, then computing MFCCs with 16 Mel bins. Finally, they get resized to the desired $[49, 10, 1]$ shape. The confidence threshold was set to 0.5.
- `slow_service.py`: this script, instead, reads the message coming from the client, then it decrypts the track and performs the preprocessing as described in the assignment. After that, the prediction gets computed and sent back to the client.

TABLE I. REQUEST

PATH	/prediction
PARAMETERS	None
BODY	**bn:** url, **bt:** timestamp, **record:** raw audio

TABLE II. RESPONSE

CODE	200 (if successful), 400 (otherwise)
BODY	predicted class

Through the described approach, we were able to get an overall accuracy of 91.489%, with a fast pipeline time of about $38ms$, and a total communication cost of 1.638 MB, allowing us to fulfill all the given constraints.