Group 2 — Homework III

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

Exercise I - Model registry

We developed a model registry service deployed on our Raspberry Pi that, through a RESTful API, offers three different services, requested by two different client applications:

- *registry_client.py* sends two different requests to the service, first of all to add the MLP and CNN *tflite* models on the Model Registry, then list all the models stored in it and start the inference of temperature and humidity.
- *monitoring_client.py* receives the temperature and humidity alerts and prints them on the screen.

1) Add service)

For the **add** service, we implement the POST HTTP method since it is designed to send data to the the service from an HTTP client. Specifically, we can send the model that we want to add on the Raspberry Pi (which is where our service is run): indeed, the service accepts the data (i.e. a *tflite* model) enclosed into the body of the POST message, which is defined on our notebook (that works as client).

2) List service)

For the **list** service, we implement the GET HTTP method, since we are only asked to retrieve the models previously loaded in our Raspberry Pi. In this context, GET is the most appropriate one, since no data need to be sent from client to service.

3) Predict service)

For the **predict** service, as for the list one, we exploit the GET HTTP since, once again, no data are required to be sent: the only data necessary for the inference are retrieved by the DHT11 sensor on our Raspberry Pi. Once we started the inference service, we are asked to send an alert when the prediction errors exceed the thresholds. To do so, we decide to use the MQTT communication protocol, because it allows us to get an alert every second without blocking the execution of the service. On the other hand, if we had used REST, each time we encountered an alert, we would have been interrupting the service to send the message information to the *monitoring_client*. This happens because, before starting the inference, we need to fill a window with 6 samples: however, if you interrupt the service and restart it, you need to re-fill the window one more time, and you get your first alert after 6 seconds. With MQTT, we are able to keep the window updated and the service running without interrupting it.

Exercise II - Edge-Cloud Collaborative Inference

We developed an edge-cloud collaborative architecture which is able to achieve high accuracy prediction minimizing energy consumption. For this purpose, we exploit the MQTT protocol, since it allows high reliability of the communication, thanks to the *Quality of Service* (QoS). QoS can be chosen by developers based on the level of reliability desired: in our specific case, we chose $QoS = 2$, assuming a sensible application that cannot afford to loose data during the communication and compromising the total accuracy. Our application is composed by two hosts that both serve as publisher and subscriber at the same time: a client (*fast_client.py*) and a web server (*slow_service.py*). The client computes the prediction of the audio, measuring its confidence; then the audio whose predictions are classified as "not confident" are published on the corresponding topic on the broker. The server, subscribed to this topic, receives them and deploy a slower pre-processing to compute more accurate predictions. These predictions are then published on the topic "prediction": the client, subscribed to this, uses them to compute the final collaborative accuracy.

For the pre-processing pipeline, we set a fast inference one running on the edge, which is bounded by hardware and memory resources. In particular, we downsampled our signals with a factor of 2, achieving a lower latency both in the pre-processing step and during inference. With the fast pipeline we obtain a total inference time of 37.73ms. On the other hand, we consider the web server running on the cloud. We opt for a slow pipeline able to reach an higher accuracy but with an higher latency: we keep the sample rate equal to the one on which the model was trained. The parameters for the two proposed approaches are showed in the table I.

A *success checker policy* decides whether the prediction is confident or not. The amount of information sent from client to server depends on this choice: it is crucial to maintain both a limited communication cost and a decent collaborative accuracy. To construct such policy, we check the softmax output of the predictions, looking at the top-2 probabilities. We decide to consider our prediction confident if the probability of the most likely label is at least 20% more than the second one. If we had looked just at the top probability, we would have to rely on every prediction with a probability higher than 50% in order to satisfy the cost constraint, but this is not ideal. According to this, we send about 45 out of 800 raw signals from client to server, with a communication cost of 1.887MB and a collaborative accuracy of 91.875%.

TABLE I
DIFFERENT PRE-PROCESSING FOR THE EXERCISE 2

	Parameters							
	sample rate	mfcc	frame lenght	frame step	num mel bins	lower frequency	upper frequency	num coefficients
Fast pre-processing	8000 kHz	True	320	160	16	20	4000	10
Slow pre-processing	16000 kHz	True	640	320	40	20	4000	10

APPENDIX

Command use to compute the total latency with the fast pre-processing configuration:

```
python kws_latency.py --model kws_dscnn_True.tflite --rate 8000 --bins 16 --length 320
                      --stride 160 --mfcc
```