HW 3 Group 14

Scorca Francesco s288876 Dicataldo Michele s290091 Padovano Dario s291475

1. Model Registry

1.1. Registry service

We implemented a class for each service of the register (*add, list, predict*) using the HTTP methods:

- *add*: implemented through the "PUT" method, since we need to pass a body containing the *tflite* model and its name, to be saved in the folder *models* in the same path of the service script. The "PUT" method was preferred to the "POST" one, since we envisioned the case in which the folder contains a model with the same name (maybe an older version), and it is needed the possibility to update it. The method reads, decodes and saves the model in *models*.
- *list*: implemented through the "GET" method since no body needs to be passed to the service. The method lists the models contained in *models* and returns the body containing that list to the client.
- *predict*: implemented through the "GET" method since no body needs to be passed. The method is used to predict temperature and humidity every second[1] and send alerts to the *monitor* if the absolute error with the measured values is greater than a given threshold, specified with the name of the model (*CNN*) to use for the predictions. The alert is published using the MQTT protocol, since it implements a light near real-time asynchronous communication. Indeed, such task is by nature event-driven[2], and does not require a client invoking a server each time.

1.2. Registry client

The client sends the service requests to the server, passing the URL and the body[3] as parameters. As result it prints the *status code* of the request and the number of models saved in the *models* folder.

1.3. Monitoring client

The monitor implements a *subscriber* for the MQTT protocol which will receive the alerts from the *publisher* and print them on screen, comparing the predicted and the actual values.

[1] the time window is composed of 6 samples, so there are 6s without predictions

[2] the event is the crossing of the threshold

[3] only for the *add* service

2. Edge-Cloud Collaborative Inference

In order to develop a collaborative network for Keyword Spotting, it was necessary to write two different applications for the web service and the edge client, having different pre-processing pipelines parameters.

2.1. Web side

The script related to the web side involves a slower but more accurate inference pipeline. The communication with the edge device is implemented through the REST protocol since we are in the hypothesis of working with a single edge device, and in this way responsiveness and security are preserved too. Conversely, MQTT could be used in a scenario with a large numbers of devices and no need of a fast response from the edge.

2.2. Client side

In the client application there are two main elements:

- *fast inference pipeline*. The optimization consisted in a modification of the pre-processing parameters with which the network were trained[4], in particular it was changed the value of the *frame length* to 480 (30ms) and the number of Mel bins to 32. Previous works showed that such parameters are the only to focus on.
- *success checker*. A policy to decide if it is necessary to interrogate the web service to mitigate any uncertainty: in this case the *raw* audio file is sent to the cloud, fed to the slow inference pipeline, and the result is returned to the edge. Specifically, we apply a *SoftMax* to the outputs of the edge model and a sample is sent if the difference between the two highest values is greater than 0.28.

2.3. Results

The techniques exploited led to a collaborative accuracy of 91.2%, with a communication cost of 1.90MB and an

[4] Hence, being optimal, they will characterize the slow inference pipeline.

average total inference time of 37.08 ms, with confidence interval[5] [35.97ms, 38.19ms].

[5]Statistics computed on 10 runs, with con confidence level 95%.