ML4IOT: Homework 3

Tommaso Massaglia
s292988@studenti.polito.it

I. Exercise 1: Model Registry

The service has three aviable methods: **POST**, **GET**, and **PUT**.

A. POST

It receives in the body the model as a Base64 encoded and zlib compressed string and, after decoding and decompression, stores it in the models subfolder with the specified modelname. Compression is required to reduce the load in the network and storage space on the *rpi*. POST was chosen as we edit files.

POST body	
model	A Base64 encoded, zlib compressed model
modelname	the name of the model (CNN, MLP, ...)

B. GET

If the GET method is called, the service simply returns with code 200 a list of the aviable models in the models subfolder, we use GET as we need no body or write.

C. PUT

Whenever the PUT method is called, the server starts an MQTT service to send alerts whenever the predicted temperature or humidity difference with the actual one (using as input the last six measurements) is above a threshold (*tthres* and *hthres*). The alerts are sent under different channels depending on the event: */alerts/temp* for temperature, */alerts/humid* for humidity, and */errors* for measurements errors. After *duration* time is elapsed, the service is stopped and it returns to its RESTful state.

PUT body	
model	The name of the model (CNN, MLP, ...)
tthres	The temperature threshold
hthres	The humidity threshold
duration	How long to run the monitoring

All the fields have a default value, so the prediction can be started even with only the predict field in the body. PUT was chosen as we need to have a body of informations.

D. Clients

The *registry_client.py* client has multiple fields in the console: –request (forced to PUT, POST or GET), –modelpath and –modelname for the POST method fields, –predmodel, –tthres, –hthres, –duration for the PUT method fields. If the model specified in –modelpath is not compressed, the client handles compression before sending it to the service.

The *monitoring_client.py* is a simple MQTT subscriber that listens on the */alerts/#* channel; it runs for 100 seconds before closing itself.

II. Exercise 2: Edge-Cloud Collaborative Inference

A. Fast Client

The client runs on the edge, our raspberry pi, and has on it a preprocessing and predicting pipeline that allows for fast inference. The files are preprocessed: the audio file is padded to 16000 frames and then sampled to take 1 every three samples before in order to reduce the time, then, the 10 most relevant MFCCs are extracted.

parameters	
frame length	213
frame stride	106
# mel bins	20
mel frequency	5333
# mfccs	10
low freq	20
high freq	2666

After the file is preprocessed, the *kws_dscnn_True.tflite* model (which was trained with better parameters) is invoked and a prediction is made. On the prediction array, the *softmax* is computed to get the probability for each label; if the $top1$ probability is below 86%, the file is compressed using *zlib* to reduce as much as possible the communication cost, encoded with Base64 and sent to the *slow server*. Inference time is computed up to the compression step if present, and on average was $\approx 35ms$.

B. Slow Server

The slow server is a RESTful server with only the **PUT** method aviable, as we need a body of data; the method gets as input in the body the *audiofile* field, a Base64, zlib compressed numpy array representation of the audio.

Whenever a request is received the model does the same steps as the fast pipeline, but using different parameters:

parameters	
frame length	640
frame stride	320
# mel bins	20
mel frequency	8000
# mfccs	10
low freq	20
high freq	4000

It then returns with code 200 a message with only the label of the prediction as an integer, to reduce communication costs, that is then used by the fast client as the new prediction.

The *sys.getsizeof()* method was used to measure the size of the input and output requests bodies, with a total output of $\approx 1.6MB$