Machine Learning for IoT – HW2

Davide Napolitano, Edoardo Lardizzone – Group20

The scope of this first exercise is to create two different models that are able to classify an audio of the KWS dataset, by applying the *"Big/Little Inference"* paradigm. The main target is to construct a little model which runs on the Raspberry Pi client with a maximum dimension of 20kB and a maximum latency time of 40ms and a big model which runs on the Notebook service. The little model has to predict the label of a preprocessed audio and if the prediction is not good, the client app has to send the raw audio to the big service on the Notebook in which it is preprocessed and then classified by means of the big model, and then the obtained prediction is sent back to the raspberry. The total accuracy has to be higher than 93% and the communication cost of the data sent to the notebook must be smaller than 4.5MB.
The little model is a DSCNN with four conv. layer with respectively 48, 40, 40 and 40 padded (*"same"*) filters (also depthwise similarly padded), MP(2,1)* and a quantization (int8) before saving it as tflite, while the big model is also a DSCNN with five convo with padded (*"same"*) filters of 512, 512, 512, 256 and 256, MP(2,1)*, Dropout(0.2), Dense(16) and quantization (int8) before making the tflite (other parameters are the same as in the model presented in lab3, except in the little model with the first conv kernel = [5, 5]). After a train of 25 epochs, the accuracy of the little model is 90.625% with a size on the tflite of 20288B while the big reaches 94.625%. The outputs of these networks are the fully connected layers with 8 units that we transform in vectors of probabilities via the softmax operation. The succes checker, which is constructed in the little client to understand when to send the audio to the big service or not, is built on this output vector: whenever the difference between the highest probability and the second highest one is smaller than a threshold, the audio is sent. We found two different values useful for two different qualities: if the difference is smaller than 0.5, 94 of the 800 audios are sent to the big service with a total accuracy equal to 93.125% and a communication cost of 3.78 MB, instead with a threshold of 0.6, 110 audios are sent reaching an accuracy of 93.875% and a communication cost of 4.39 MB, in this way it is possible to decide if prefer the accuracy or the communication cost, with every value in the middle both constraints are respected.
In order to do the exercise we implement a REST protocol of communication, which is a protocol based on synchrony of the client and the service, since the communication is made directly with a request/response approach. Because the communications are made on one single channel (Raspberry Pi ↔ Notebook) we thought that this was the best protocol to use, in fact a server that distributes data is not indispensable. When the threshold is triggered the *little_client* reads the file of the test dataset via *tf.io.read_file()* which output is transformed in a numpy array and then encoded/decoded in a string via the base64 *.encode()* and the *.decode()* methods and JSON serialized, the body of the message also contains the IP address of the raspberry and the timestamp to be compliant with the senML+JSON format. On the other side the *big_service* receives the audio file with the PUT function, since we transfer big quantities of data (GET not suitable) and we don't need to create new entities (POST), and preprocesses it after a decoding, the prediction are made on the mfccs of the audio using a padding to 16000** while the little service is made with a resample to 8000*** in order to satisfy the constraint on the latency (which is 39.2±0.55ms).

The scope of the second exercise is to create N models (N>1) able to predict the same audio of ex1, by applying the *"Cooperative Inference"* paradigm. On the Raspberry Pi we have to use a *cooperative_client* that iteratively sends a preprocessed audio** from the test set to the models on the notebook and, after receiving their output, it applies a cooperative policy to assign a label to the audio. The final accuracy on the test set has to be >94%. In order to do this we create three different models which are three DSCNN with these configurations (convs and depthwise padded as in ex1):

	Model Description	Accuracy Model	Final Accuracy
2.tflite	conv:128-64-64; MP(2,1)*; dense:32; dropout:0.2; first conv kernel: [5,5]	92%	-
1.tflite	conv:128 x4; MP(2,1)*; dense:32; dropout:0.3; first conv kernel: [5,5]	92.375%	1+2: 93.5%
3.tflite	conv:256-256-128-128; MP(2,1)*; first conv kernel: [3,3]	93.125%	1+2+3: 94.5%

For this exercise we use a MQTT protocol which is really useful since in this way the client has to send the audio only one time to the message broker. As suggested, we consider all the clients online, however an asynchronous approach would be more handy if we took into consideration a real case in which we don't know if the *inference_clients* are online or not. For what concern QOS, we set it in all the clients equal to 0 in order to speed up the transmissions.The three different inference clients take the audio with the topic they subscribed to (the Raspberry Pi) from the server and they send back the prediction (the output vector of probabilities obtained by applying a softmax to the last Dense layer of the DSCNNs). In our case the management of the incoming messages on the *cooperative_client* is done through a function called *"message_handler"* that is associated with the *myOnMessageReceived* of the *myMQTT* class. This function takes as input the topic and the message, the topic is used to make an association between *inference_client* and index, while the topic embed the output vector. Each time a message arrives, the content is stored on the *pred* inner list associated to the infecence client (inference1->idx=0, inference2->idx=1, inference3->idx=2 and *pred* is a list of list where the outer list has 3 inner list), when all the inner list are populated (at least one el inside each one), the first element of each of them is taken, added into an external variable *somma* and removed. From this *somma* variable we take the argmax since in our view this allows us to take the overall best prediction, a thing that can be done with a standard major voting of the best label of each model. Then we take the corresponding label and we check if it is correct, incrementing the *correct* counter. An important thing about our implementation is that, each time

we apply the policy on the predictions, they are then removed from *pred (.pop(0))*, so in the next controls will be taken always the three predictions of the following audio, in a nutshell we simulate a FIFO queue. An analogue management of the incoming messages is done on the *inference_client*, which uses a *message_handler* function to receive the audio, then it is decode(b64), the inference is made and the output vector (probs) is sent to the raspberry pi. About sending/receiving messages, the *cooperative_client* sends the audio (only the mfccs without the *tf* metadata in order to not increase network weight) encoded with base64.b64encode and decoded to string with .decode() as it is requested by the senML+JSON standard, instead the *inference_client* sends a list according to JSON format.
All the clients implement a loop of 800 elements (800 is the the test set size) with a sleep of 0.01s in order to be sure that there is enough time to do all the work, in addition in the *cooperative_client* is added a sleep of 1s after the loop in order to be sure that the policy is applied on all the 800 tracks. The final result, as reported in the table above, is obtained as *correct/counter* (*counter* indicates how many times is invoked the policy) an accuracy of 94.5%.

* MaxPooling2D applied after each block excepted the last one.
** (*sampling_rate*=16000, *lower_frequency*=20, *upper_frequency*=4000, *frame_lenght*=640, *frame_step*=320, *num_mel_bins*=40, *num_coefficients*=321) with the SignalGenerator class from the homonymous .py
*** (*sampling_rate*=8000, *lower_frequency*=20, *upper_frequency*=4000, *frame_lenght*=320, *frame_step*=160, *num_mel_bins*=40, *num_coefficients*=161) with the SignalGenerator class from the homonymous .py