Name 1 :
Name 2 :

Lab 6 Embedded Artificial Intelligence on microcontroller

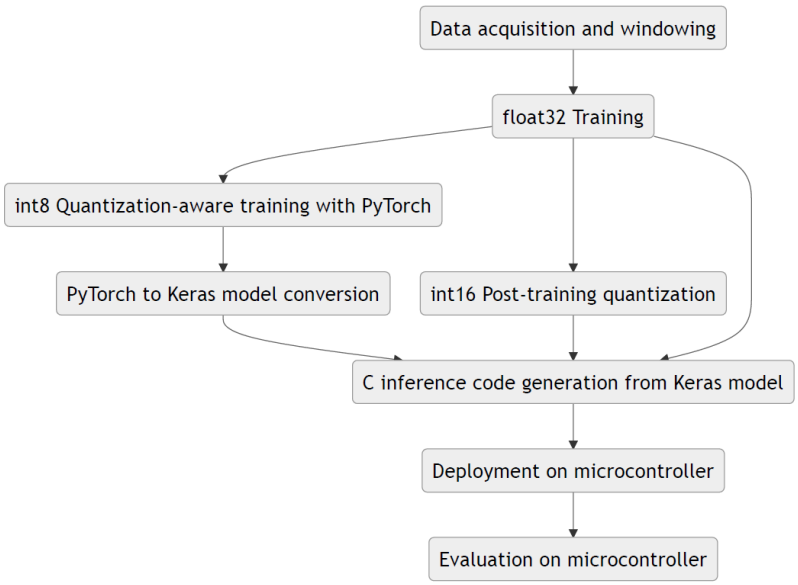
MicroAI - Automatic deployment of neural networks on target

Polytech Nice Sophia

During this lab, you will use the software tool MIcroAI developed at LEAT laboratory to automatically generate embedded implementation of convolutional neural networks.

The different steps of the generation are described in the following figure. Three types of inference are possible, but this lab will focus on the int 16 post-training quantization. This quantization reduces the number of bits used to encode a value from 32 to 16 after the network has been trained by converting the floating-point values to 16-bit integers with fixed-point representation for all the network weights and activations using the same scale factor.

More details about the two other types can be found on https://www.mdpi.com/1424-8220/21/9/2984



Part I. Manual generation of embedded neural networks

Part I.1 Fully connected layers

Work to be done: Complete the software code for Fully connected layers

File: *model.h*, function dense, line 232

In this first step, we provide you the bare C code used to generate embedded CNN. In order to better understand what are the challenges of embedded inference, you will first build manually your own simple neural network that will be composed of a single convolution layer and a single dense (fully-connected) layer. The C code is provided on moodle, but the dense function is not provided and you have to code it.

Name 1 :
Name 2 :

Take a time to read the code and understand its organization. You can especially observe that each layer is implemented as a function.

In a dense layer (empty dense function), the computation is realized in several steps:

i) a simple MAC (Multiply and ACcumulate) operation iterated over the units of the layer (FC_UNITS) and over the input samples (INPUT_SAMPLES). The weights are already stored in the 2D kernel array of dimensions [Units][Inputs].
ii) For each unit in the layer,
   a. a scale of the result of the MAC : function scale_number_t()
   b. the addition of the related bias
   c. the clamping to 16 bits of the result calculated onto 32 bits: function clamp_to_number_t()
   d. write the result in the output array

The result of a fixed-point multiplication contains twice the number of bits for the fractional part than its operands (when they use the same scale factor), this is the reason for scaling back the result at step ii) a. Adding fixed-point numbers does not require scaling back the result afterwards, but both operands must have the same scale factor, this is the reason for performing the addition only after step ii) a.

Part I.2 Build your embedded network

Work to be done: Complete the software code for the instantiation of layers

File: *model.h*, function cnn, line 322

In order to instantiate one specific network, you now have to complete the cnn function by calling the related function with the right buffer pointers as parameters (in our case 4 buffers: input, conv1d_output, flatten_output and dense_output).

In this exercice, you will build a network composed of 3 layers:

i) One 1D convolution
ii) One Flatten layer
iii) One Dense layer

Note that the hyper-parameters for each layer are specified by macros before the declaration of each function. So, the cnn function just has to call the layer function without specifying the size of input/output shapes, of the kernels…

Note that the weights of the network are already provided in the file, meaning that the training has been already made.

Part I.3 Evaluate your model on the board

Work to be done: complete the software code for the conversion of data

File: *S5Lab2.ino*, line 51

Now that your network is built, you can open the main file to be compiled with Arduino IDE: *S5Lab2.ino*. Note that this file includes the *model.h* file at line 2 (as C file), and call the cnn function at line 55.

Name 1 :
Name 2 :

The rest of the code read the test data from the serial link, provide the data to the CNN (inputs buffer), and calculate the predicted class from the output activity. The result is sent back on the serial interface.

The input data are received as characters (char *pbuf on line 43) and are first converted from char to float (finput array).

Then, the float data (finputs) have to be converted in fixed point (inputs) representation on 16 bits. The number of bits for the fractional part is specified by FIXED_POINT in the model.h file: here 9 bits.

With such a fixed representation the real value 2^{-9} becomes the integer 1, 2^{-8} the integer 2, 2^{-7} the integer 4…

So, for each input, your conversion code has to multiply the real data by 2^{9}. The floating point result of this multiplication has then to be converted in integer in the type long_number_t. Then this long integer value is clamped in our working representation (16 bits) with the clamp_to_number_t() function already used previously.

Warning: the data is received in *channels_last* format (TensorFlow/Keras convention), while the C inference code excepts the data in *channels_first* format (PyTorch convention). Therefore, the *finputs* array has dimensions [MODEL_INPUT_SAMPLES][MODEL_INPUT_CHANNELS] and the *inputs* array has dimensions [MODEL_INPUT_CHANNELS][MODEL_INPUT_SAMPLES]. In your code you must make sure you convert the value in the correct order.

Representation	**Number coding**	**sign**	**exponent**	**Mantisse**
Float (simple)	(-1)S.1,M.2E	S – 1 bits	E – 8 bits	M - 23 bits
Fixed point (16 bits)	QN,M N = 7 bits M = 9 bits	2 complement representation	-	-

Compile the code of the file S5Lab2.ino with ArduinoIDE and download the code on the board. Then execute on your computer the script evaluate.py (available on moodle). It reads the files containing the test set and send the values to the board and get the results back.

Example on windows: *> python evaluate.py x_test.csv y_test.csv COM4*

Part II Automatically generated embedded neural networks

Part II.1 Train your convolutional network

Work to be done: Reuse a pretrained convolutional network on UCI HAR

You can now reuse a previous CNN that attain good accuracy (>90%) on UCI HAR.

Part II.2 Generate a complete convolutional network

Work to be done: Use the provided MicroAI software to automatically generate the C code of your network

Download from moodle the notebook *UCA-HAR-SI4_MicroAI.ypynb* and adapt the code to your CNN model.

Name 1 :
Name 2 :

Run the notebook to i) remove the softmax layer (executed on your computer, not on the MCU), ii) install MicroAI and iii) call MicroAI to convert the code in C.

Note the parameters of the Converter constructor. They correspond to the quantization explained in Part I, but they can also be adapted to work on 8 bits during inference.

At this step, you should find in your working folder a subfolder 'output' containing one file per layer and a header file (*model.h*) containing all the code gathered in a single file (easier for compilation with ArduinoIDE).

Part II.3 Evaluate your network on the test set

Work to be done: Validate your network on the board with the data of the test set UCI HAR

Replace the previous *model.h* file (part I.3) by the new one and download the code on the board and run the python code evaluate.py on your computer in order to validate it on the MCU.

Make sure you find a similar accuracy on the board and during the validation of your keras model.

In the next session, you will reproduce the steps of part II onto your own data collected from the accelerometer of the board and make the prediction in real-time.