Lab II : Embedded Artificial Intelligence on microcontroller – Build your CNN for MCU

Polytech Nice Sophia

During this lab, we will code your own CNN using Tensorflow Notebooks. Then, you will use the software tool MIcroAI developed at LEAT laboratory to automatically generate embedded implementation of convolutional neural networks.

Part I: Tensorflow and Keras

I.1 Train a network on MNIST Dataset (Image classification)

In this part, your goal is to define a Convolutional Neural Network reaching **between 98,5 and 99%** of good recognition (accuracy) on the MNIST Dataset. The accuracy of your model must be confirmed by an average **of 3 learning** and the validation **on 250 inferences** (Obviously, the statistical studies would need more trials, but it is here a first introduction to CNN exploration).

To reach better accuracy you can change the following hyper-parameters of your CNN in the Build model and Train model parts of TF script from Lab 1:

- The number of **filters** in each convolution (*Conv2D*) layer
- The size of the **kernels** in each convolution layer (by default set to 3x3)
- The number of **Convolution layers**
- The use of ***MaxPool2D*** layers between *Conv2D* layers
- The number of ***Dense* layers**
- The number of **neurons** in each dense layer (except the output layer)
- The **activation function** in each layer
- The number of **epochs** of learning (how many times the network will learn the entire dataset)

When you get the expected accuracy, fill the following table as your result.

Layer	Output shape	Number of parameters	Kernel
Input	(28, 28, 1)		
Conv2D	(None, 26, 26, 32)	320	3x3
MaxPooling2D	(None, 13, 13, 32)	0	
Conv2D	(None, 11, 11, 64)	18,496	3x3
MaxPooling2D	(None, 5, 5, 64)	0	
Flatten	(None, 1600)	0	
Dense	(None, 128)	204,928	
Dense	(None, 64)	8,256	
Dense	(None, 10)	650	
Total trainable parameters	232,650		
Number of Epochs	3	15	15

Final model	Accuracy on test	Loss on test	Accuracy on validation

Learn 1	**94.23%**		**0.1768**		**95.2%**
Learn 2	**98.32%**		**0.0526**		**98.32%**
Learn 3	**99.25%**		**0.0325**		**99.25%**
Results	**Average** **97.2(6)%**	**Std deviation** **2.1805554847841**	**Average** **0.0873**	**Std deviation** **0.063815828757449**	

I.2 Train a network on UCI HAR (time series classification)

In this part, your goal is to define a Convolutional Neural Network reaching **at least 90%** of good recognition (accuracy) on the **UCI HAR** Dataset. The accuracy of your model must be confirmed by an average **of 3 learning** and the validation **on 100 inferences**!
(Obviously, the statistical studies would need more trials, but it is here a first introduction to CNN exploration)
First, get the new notebook from https://lms.univ-cotedazur.fr and test it. Note that data are time series. Consequently, the shape of data is 1D, just as the convolution kernels.
First, **modify the script in order to plot the accuracy and loss** during the training according to epochs. It will help you to interpret the good or bad behavior of your tuning.
You can download the dataset at:

```
https://archive.ics.uci.edu/ml/machine-learning-databases/00240/UCI%20HAR%20Dataset.zip
```

The **test dataset** has been divided in
two parts: the all test set composed of 2793 vectors, and a subpart of it composed only of 250 vectors for on-board validation.
To reach better accuracy you can change the following hyper-parameters of your CNN in the Build model and Train model parts of notebook:

- The **Learning rate**
- The number of **epochs** of learning (how many time the network will learn the entire dataset)
- The number of **filters** in each convolution (*Conv1D*) layer
- The size of the **kernels** in each convolution layer
- The number of **Convolution layers**
- The use of ***MaxPool1D*** layers between *Conv1D* layers
- The number of ***Dense*** **layers**
- The number of **neurons** in each dense layer (except the output layer)

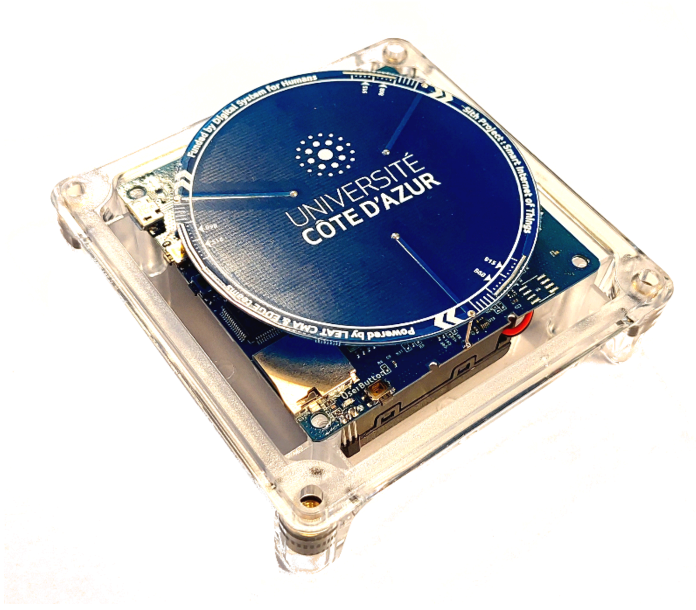
When you get the expected accuracy, fill the following table as your result.

Layer	Output shape	Number of parameters	Kernel
Input			
Dense	None, 16	8, 992	
Dense	None, 8	136	
Dense	None, 6	54	
Total trainable parameters	9,182		
Number of Epochs	20		
Final model	Accuracy on test	Loss on test	Accuracy on validation

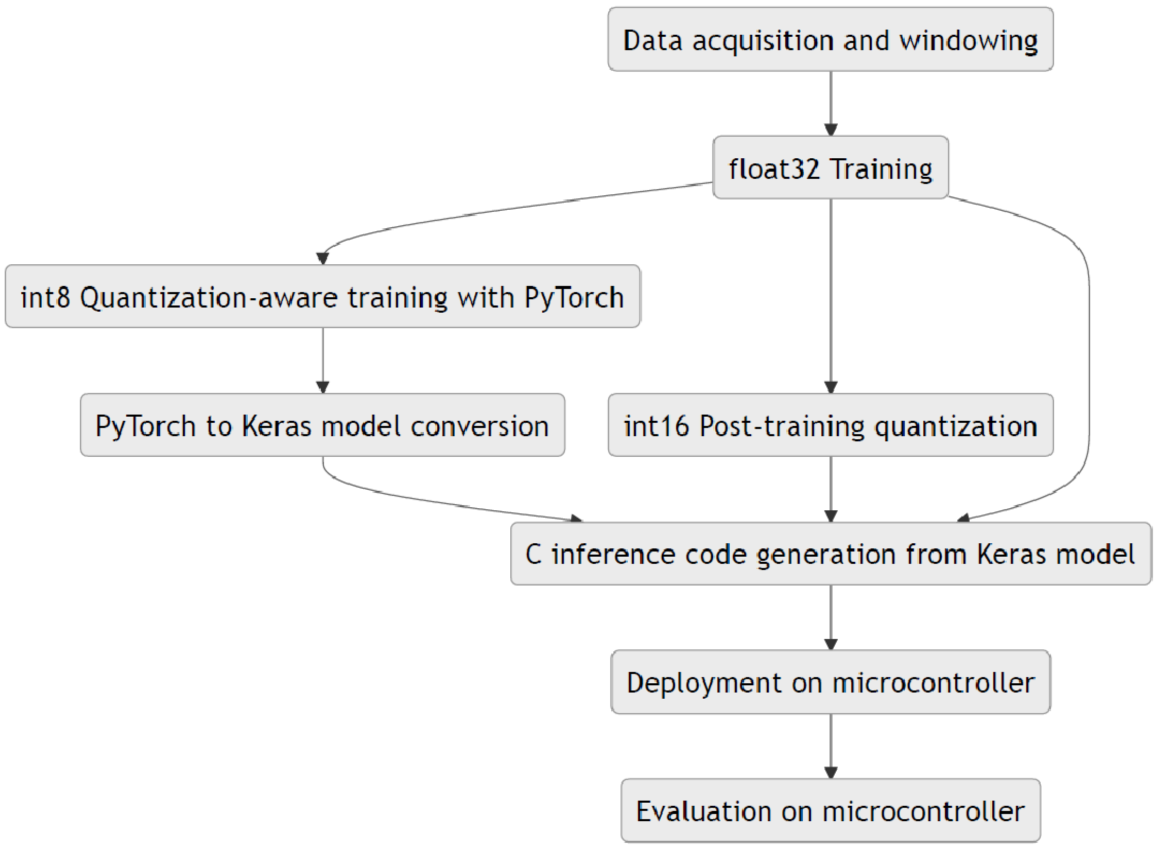
Learn 1	**0.9835**		**0.04762280359864235**		**0.9579**
Learn 2					
Learn 3					
Results	**Average 98.35%**	**Std deviation 98.35%**	**Average 0.04762280359864235**	**Std deviation 0.04762280359864235**	

Part II: MicroAI and Embedded CNN on Cortex M4

Reminder:
The target is a RFThings-AI Dev Kit board equipped with a STM32L476RGT6 Microcontroller. This MCU is based on the ARM Cortex M4 architecture and runs at a frequency of 80 MHz. The board provides 1 MB Flash and 128 KB SRAM.
Please refer to lab 1 if you have any issue with your environment.



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


Data acquisition and windowing
float32 Training
int8 Quantization-aware training with PyTorch
PyTorch to Keras model conversion
int16 Post-training quantization
C inference code generation from Keras model
Deployment on microcontroller
Evaluation on microcontroller

II.1: Manual generation of embedded neural networks

II.1.1 Fully connected layers

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

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

The code provided in this model.h file contains the parameters of a simple CNN learnt on the UCI HAR Dataset.

In this first step, we provide you the bare C code used to generate embedded CNN. In order to better understand what the challenges of embedded inference are, 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.

Take 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:

1) 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].
2) For each unit in the layer,
   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 II.1.2 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 exercise, you will build a network composed of 3 layers:

1) One 1D convolution
2) One Flatten layer
3) 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 II.1.3 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.

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 to 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*

You can find this two files, which has been extracted from the UCI HAR dataset, on moodle.

Part II.2 Automatically generated embedded neural networks

Part II.2.1 Train your convolutional network

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

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

Part II.2.2 Generate a complete convolutional network

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

Download Part2.2 from moodle, *place the UCIHAR dataset (download from* here) in the d1_notebook folder, open the Jupiter lab environment and adapt the code with your previously designed CNN model.

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.2.3 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 II.1.3) 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 and complete the following table.

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.