ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

Machine Learning on Microcontrollers

227-0155-00G

Lab 5

Image Classification on a Neural Network Accelerator Enabled MCU

Michele Magno, PhD
Cristian Cioflan
Marco Giordano
Viviane Potocnik
Victor Jung

Wednesday 9th November, 2022

1 Introduction

In this lab, we are going to discover the MAX78000, a chip from Analog Devices. The Microcontroller (MCU) has a peculiar architecture, featuring a Cortex-M4F and RISC-V dual-core configuration and a 64-core neural network accelerator. In particular, the neural network accelerator can be seen as a peripheral to offload the inference of neural network tasks, achieving higher computational and energy efficiency.

2 Notation

Student Task 1: Parts of the exercise that require you to complete a task will be explained in a shaded box like this.

Note: You find notes and remarks in boxes like this one.

3 Overview

In this section, a broad overview of the MCU and the tools used to train and port the neural network to the onboard accelerator are described.

Finding information about the devices and tools you are using is a vital skill in engineering. This chip and the board are well-documented on the official website. For the software, a piece of very good advice is to read the rich README file on GitHub.

Before proceeding with Section 4, it is recommended to install the required tools: here you can find the tools needed to train the neural nets, while here you can find the steps to flash the MCUs with the trained network.

Note: Very frequently you will be confronted with extremely long (thousands of pages!) documentation. Note that, while all of it is important, it is often useless to learn every detail of it. Learning how to retrieve information from long docs is a very useful skill in engineering. Search keywords and skim through documents to solve your problems.

Note: The links provided in this document point you towards the right piece of documentation. If you feel lost, click on the links in each sub-task to be directed to the right reference.

3.1 Hardware

The board we are going to work with is the MAX7800 Feather, a demo board developed by Maxim. It hosts the Maxim 78000 with a plethora of other sensors, among which is an RGB camera and a two-channel microphone (which can be used for an impressive project demonstrator).

3.2 Software

Analog Devices released a framework to train, quantize, and port neural networks to the accelerator. The software package is completely open-source and is hosted on GitHub at the following link: MaximIntegratedAI.

In particular two repositories are of special interest: ai8x-training and ai8x-synthesis, which we are going to see more in detail later in the exercise.

4 Training a neural network

Analog Devices' framework is built on top of PyTorch, which has been augmented with some custom operations to fit the constraints of the accelerator. The best way to train a neural network for the MAX78000 is to use the proposed training script, `train.py`. The parameters for the training can be passed to the script via the command line arguments, while data loaders and models are referenced from user-specified classes.

Two directories should be particularly taken into account: `models` and `datasets`. The former contains the network description files, in the latter there are the data loaders. Have a look at the files in those two directories to see their structures. Even though Analog Devices provides plenty of examples, in this lab we are going to create our own dataloder, as well as model file.

4.1 Task introduction

To showcase the training and deploying pipelines of the MAX78000 in this lab we are going to train and evaluate a meme classifier. The training dataset will be composed of only one image per class, which will be heavily augmented with the `transform` class of PyTorch. This task should be light enough to be trained on machines without a GPU, and at the same time provide a complete overview of the training and deploying process of a Convolutional Neural Network on the MAX78000. And ultimately, it is funny :)

4.2 Dataloader

The goal of this step is to write the code that will generate the training data. We start from only 4 images, which is not much to train a neural network. However, we can augment the images by trying to cover as many real-life cases as possible, to maximise the accuracy on the test set and ultimately during the final deployment.

You should copy the directory `memes` inside the `ai8x-training` directory, better if inside a `data` directory (`data/memes`). We do not care at this stage if the images are randomised or not, since the data loader will take care of it.

let us have a look at the `memes.py` file. After importing a few packages there is the main class, `MemesDataset(Dataset)`. Following the structure of PyTorch data loaders, three methods must be present to initialize, report the total length and prepare a new sample. The function `memes_get_datasets` returns an image and its label, based on an instance of the class `MemesDataset`. The data augmentation is carried out using the `transform` class of PyTorch, and the most important transformation, the

affine transformation, is already implemented. This covers already the most real-life cases of rotating the camera, translating it, cropping the image and zooming in and out from the target (scale).

Student Task 2: Familiarise yourself with the PyTorch Dataloader tutorial and get used to the Dataloader structure. Augment the Dataloader provided with other transforms.

The following piece of code is not proprietary to PyTorch and can thus be confusing. It is used by the Analog Devices toolchain to identify the network, the input and output and which function has to be called to load data.

```
"""
Dataset description
"""
datasets = [
    {
        'name': 'memes',
        'input': (3, 64, 64),
        'output': list(map(str, range(4))),
        'loader': memes_get_datasets,
    }
]
```

4.3 Network

The network description is reported in `memenet.py`, which should be placed inside the `model` directory. This file is somehow similar to the previous one, with a neural network class and a function instantiating it. The network class follows the PyTorch structure, with a constructor and the forward function. In the constructor, the different layers are declared, and the network is composed in the `fowrard` method. At the end of the file, there are again a few identification lines for the training script.

Student Task 3: A skeleton of a neural network is provided. Add some more layers, playing with the supported ones described in this section of the README file.

4.4 Neural network profiling

Now that you have played a bit with the tools, let us implement the following network.

```
conv1 =>  kernel = 32, kernel_size = 3, padding=1
mp1 => kernel=2
conv2 =>  kernel = 32, kernel_size = 3, padding=1
mp2 => kernel=2
conv3 =>  kernel = 24, kernel_size = 3, padding=1
mp3 => kernel=2
conv4 =>  kernel = 24, kernel_size = 3, padding=1
conv5 =>  kernel = 24, kernel_size = 3, padding=1
mp4 => kernel=2
flatten
fc1 => in=dim_mp4_x*dim_mp4_y*24, out=32, bias=True
```

```
fc2 => in=32, out=16, bias=True
fc3 => in=16, out=4, bias=True
```

4.5 Quantization-Aware Training and quantization precision

As you have seen in the previous labs, the available memory and storage are hardware-imposed limitations on the complexity of the network and, thus, on the attainable accuracy. By reducing the representation precision (i.e., decreasing the bit-width) of the network's parameters, input(s), and intermediate activations, said limitations can be mitigated without incurring significant accuracy losses. The term quantization thus refers to the discretization of the continuous real-valued, while minimizing the number of bits required to represent the numbers and also to maximizing the computational accuracy. The process of quantization often involves a change in the data type, from a floating point (FP) representation to an integer (INT) one.

Considering the process required to quantize a neural network, we distinguish two types of quantization. Post-Training Quantization (PTQ) refers to the process of quantizing a pretrained neural network by simply discretizing the FP32 parameters to the desired precision; calibration data is often used to determine the quantization range, thus maximizing the information encoded in the available bit-width. Quantization-Aware Training (QAT) extends the scope of PTQ, as a (pretrained) neural network is firstly fake-quantized (i.e., the FP32 data type is only allowed to encode values within the desired quantization range) and retrained, with the aim of reaching a similar accuracy to that of the full-precision network. Only afterwards, a true-quantized network is generated, obtaining a byte-level (i.e., INT8) or sub-byte-level quantized neural network.

The Maxim framework allows you to both operate PTQ and QAT. This setting can be specified with a `yaml` file inside the `policies` directory. Two parameters are of particular interest: `start_epoch` that signals from when the weights should be considered quantized and `weight_bits` indicating to what precision the weights should be quantized.

The toolchain also allows you to schedule a certain learning rate as the network trains: you can pass a `yaml` file with the `--compress` argument to the training script. As the training of this network is purposefully kept short, we will not use this feature, but example files can be found in the `policies` directory.

4.6 Training script

To train the model it is best to use the `train.py` script. All the possible arguments are listed in the GitHub repository. To begin the training with the current configuration the following command must be run:

```
python train.py --lr 0.001 --optimizer adam --epochs 5 \
--batch-size 32 --deterministic --compress policies/schedule.yaml\
--qat-policy policies/qat_policy_memenet.yaml --model memenet\
--dataset memes --confusion --param-hist --pr-curves\
--embedding --device MAX78000 "$@"
```

The more basic arguments regulate the epochs, batch size, optimizer as well as learning rate.

Student Task 4: Train the script using the `train.py` and the provided arguments.
What is the accuracy?

Are you overfitting?

5 Quantization and evaluation

Once the neural network has been trained, the next step is to quantize it. Again, the process is supported by a script offered in the toolchain: the `quantize.py` script inside the `ai8x-synthesis` directory. The first argument of the script is the trained network, which gets saved by the training script automatically in the `log` directory inside `ai8x-training` (you can copy it in the `trained` directory inside the `ai8x-synthesis` for convenience). The second argument is the path where the script will save the quantized version of the network. The `--device` argument indicates our target MCU, and the `-v` flag enhances the verbosity of the script.

```
python quantize.py trained/mememet_trained.tar \
trained/mememet_trained-q.pth.tar --device MAX78000 -v "$@"
```

5.1 Evaluation

Inside the `ai8x-training` directory an evaluation function is present to check the accuracy loss due to quantization. Running the following command is possible to compute the overall accuracy as well as plot a confusion matrix with the different classes:

```
python train.py --model mememet --dataset memes --confusion --evaluate \
--exp-load-weights-from ../ai8x-synthesis/trained/mememet_trained-q.pth.tar \
-8 --save-sample 0 --device MAX78000 "$@"
```

The flag `--save-sample` will extract a sample from the test dataset and save it in pickle format. This sample can be given to the synthesis tool to generate a header file containing the selected sample and can be used to test, in a supervised fashion, the neural network running on the MCU.

Student Task 5: Quantize and evaluate your model as described.
What is the evaluation accuracy for the non-quantized model?
And for the quantized model?

6 Synthesis

The last step of the pipeline is to generate the C files that will then be uploaded to the MCU. Once again, a script assists us in this operation: the `ai8xize.py` inside the `ai8x-synthesis` repository. The standard position for the quantized neural network is inside the `./trained/` directory, you should then copy the sample input generated with the evaluation command into `./tests/`.

An important missing file is the the YAML description file inside the `./networks/` directory, a configuration file that contains a description of the network with lower-level bindings for the architecture. A very good explanation for this file is given in the official documentation. In short, the synthesis tool should know from which memory location will the input be loaded, as well as its format, where to store the output, and what processor should be active.

A sample command to generate C code can be found below:

```
python ai8xize.py --test-dir synthed_net --prefix mememet \
--checkpoint-file trained/mememet_trained-q.pth.tar \
--config-file networks/mememet.yaml --sample-input tests/sample_memes.npy \
--softmax --device MAX78000 --compact-data --mexpress --timer 0 \
--display-checkpoint --verbose  --overwrite "$@"
```

All the flags are described thoroughly in the README.

Student Task 6: Complete the provided YAML file and generate the C files for MemeNet.

7 Microcontroller flashing

Once all the files are created with the synthesis script, we can proceed to load them to the MCU. Again, the official guide provides the required steps to flash the MCU.

Let us briefly analyse the different parts of the automatically generated code:

- `cnn.c/cnn.h`: files containing the neural network architecture and loading/unloading operations. If there are architecture/training changes it is most likely better to re-generate them and not modify them by hand.
- `main.c`: file containing the `main()` function.
- `Makefile`: file containing the program compilation rules.
- `sampledata.h`: file containing the sample data generated with the evaluate command, it is used to test the network.
- `sampleoutput.h`: file containing a known answer from the neural network when fed the data above. It is used as a self-test to check if the porting to the hardware has been done correctly.
- `softmax.c`: C implementation of the softmax function.
- `weights.h`: header file containing the actual kernels.

There are two ways to compile and load the program on the MCU. The first way is to use a GUI program, which is downloadable on Analog Devices's official page, and the second way is to compile the files using the Makefile and loading the binaries with GNU Debugger (GDB). If you have followed the installation steps proposed at the beginning of the exercise, you should have at least one of the two ways set up.

Using the GUI is pretty straightforward and, being based on Eclipse (as CubeIDE), it should be a familiar environment. For all the command line adventurers, compiling, flashing and debugging follows the classic GNU Compiler Collection (GCC)- GDB pipeline.

Note: Loading the program using command line tools is not needed to program the MAX78000. However, knowing what is happening under the hood when a debug button is pressed on any IDE is extremely valuable.

First, you define the toolchain path in the Makefile and compile the code. Then you have to launch a debug server, in our case OpenOCD, which will communicate with the onboard hardware debugger present on the MAX78000 Feather Board. There is a pre-packaged OpenOCD distribution within the `ai8x-synthesis` repository, as explained in the README. You might have to install some dependencies in order to run OpenOCD; below are the most likely needed ones:

```
libusb
libftdi
hidapi
```

Then you have to run the GDB client, which in the ARM toolchain is `arm-none-eabi-gdb`, connect to the server (localhost, port 3333), indicate the output file, flash the file, set a breakpoint, reset the application and continue to the set breakpoint. The needed set of commands is reported below:

```
set pagination off
target remote :3333
file ./build/memenet.elf
load
layout src
focus next
b main
monitor reset
c
```

Student Task 7: Compile, load and run the generated code on the MCU. Open a serial monitor, you should be prompted with the result of the test inference, as well as an inference time measurement.

7.1 Network evaluation

Evaluation of a system and reporting are two of the most important phases of any project. In this task, we are going to evaluate some of the most important metrics for a TinyML system: the number of weights, the latency and the inference throughput, measured in MAC/Cycle.

Student Task 8: Compute the three metrics mentioned above. How many kB, ms and MAC/-Cycle does the network account?
Model size: _____
Inference time: _____
Inference throughput: _____

Note: The latency is displayed if you add the `--timer` flag to the synthesis tool and the total count of MACs can be seen in one of the automatically generated files.

$\mathcal{E}$ **Congratulations! You have reached the end of the exercise. If you are unsure of your results, discuss with an assistant.** $\mathcal{E}$