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

227-0155-00L

Exercise 8

Image Classification on a Neural Network Accelerator Enabled MCU

Michele Magno, PhD
Marco Giordano
Potocnik Viviane
Pietro Bonazzi

Wednesday 20th November, 2024

1 Introduction

In this lab we are going to discover the MAX78000 chip from Analog Devices. The microcontroller has a peculiar architecture, featuring a Cortex-M4F and RiscV 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

This section will give a broad overview of the microcontroller and the tools used to train and port the neural network to the onboard accelerator.

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 very good advice is to read the rich README file on GitHub. Links that can come at hand are also the datasheet and user manual of the chip. Please note that you are not required to read hundreds of pages of documentation, but you shuold be able to search (ctrl+f, looking at index...) the information you need in the documentation provided you by the manufacturer.

Before proceeding with Section 4, it is recommended to install the required tools. Here are the links to the right section of the README file in the repo: neural network install, hardware install. The links provided in this exercise will lead you, when possible, directly to the right section in the documentation. Please read well the text content of the guides linked, as they provide instruction for different operating systems and configurations.

Student Task 2: Install the required tools to train and deploy neural networks with the MAX78000.

Note: The instructions call for you to install pyenv, as in previous Exercises we instead recommend to use conda to manage your virtual environments and using pip to install packages.

Note: For MacBook users with an M1 chip or a similar configuration, the installation process requires adjustments to get the appropriate Python version.

Create a conda environment configured for x86:

```
conda create -n my_x86_env -y
conda activate my_x86_env
conda config --env --set subdir osx-64
```

Install the necessary Python version:

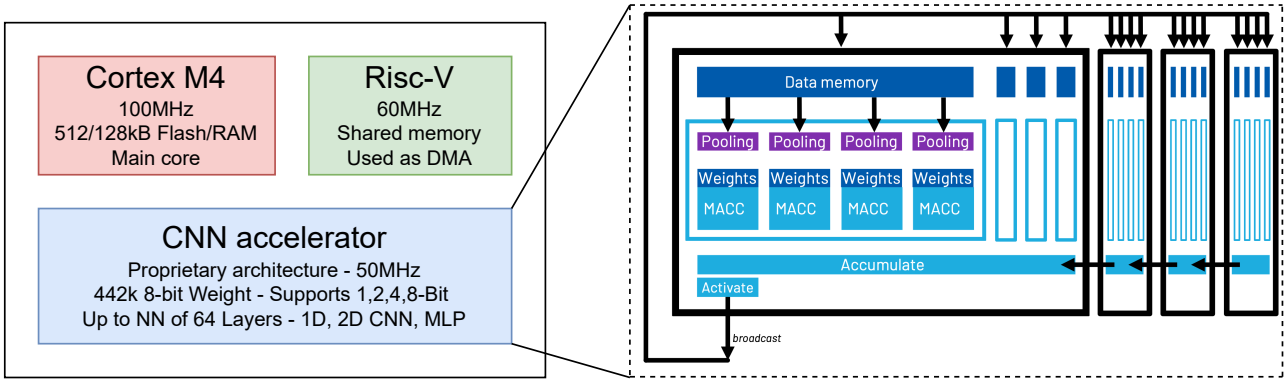

Figure 1: Overview of the differnet subsystem of the MCU and the internal structure of the NN accelerator.

```
conda install python=3.8.11
```

Note that conda is utilized for managing Python versions, replacing pyenv in this context. Therefore, any commands related to pyenv in the toolchain installation instructions can be ignored. It is only necessary that the created conda environment is active.

3.1 Hardware

This section is divided two subsections, the first describing in detail the internal components of the MCU, and the second giving information of the board you will use and the sensors it hosts.

3.1.1 MCU

The MCU hosts 3 macro subsystems, an ARM Cortex-M4 core, a Risc-V core and the neural network accelerator, as depicted in Figure 1. In particular, the Cortex-M4 is the main microcontroller of the chip, the one you will program in this exercise. It is in charge of talking with peripherals as SPI, I2C, and the CNN accelerator. It is the chip you actually program when loading the code on the MAX78000, and it will take care of copying input data to the accelerator and starting the inference sequence. The MAX78000 also includes a Risc-V MCU, which can be programmed as well (with an external programmer on the boards you are handed in). The main purpose of the Risc-V core is to act as smart direct memory access (DMA) between the Cortex M4 and the CNN accelerator, which is especially helpful when lots of data are exchanged between the main core and the accelerator.

The CNN accelerator can be seen as a peripheral on the MAX78000. It can perform a limited amount of operations and it is specialised in convolutional neural network inference. An overview of the internal components is provided in the right part of Figure 1. It provides three main RAM memories, one for weights (”Weights”), one for biases and one for activations (”Data memory”), where the intermediate results of inference are stored. It has to be noted that these RAM memories can be read only by the processor they are assigned to (this will be particularly important during synthesis), but the write operation can span all the addresses. Moreover, the inference is performed with a finite state machine (FSM) implemented in silicon, which order of operations and start/stop conditions can be set up via writing some register (from the Cortex M4 core, for example). Luckily, the weight-memory mapping and the FSM setup is done automatically by a range of tools provided by Analog Devices.

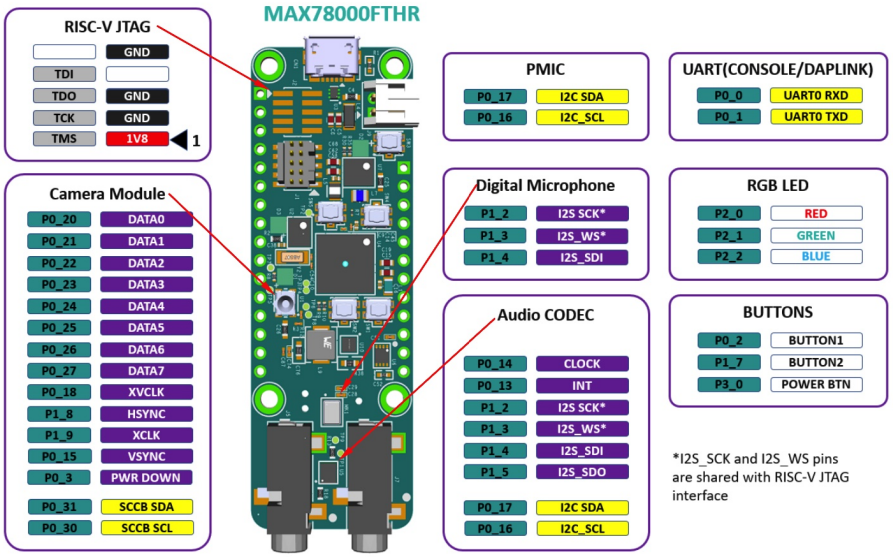

Figure 2: Overview of the differnet sensors and IC present on the Feather board.

3.1.2 Board

The board we are going to work with in this exercise is the MAX78000 Feather, a demo board developed by Analog Devices to showcase the capabilities of their chip. The board hosts the MAX78000, an onboard debugger, accessory electronics as a power management IC (PMIC), LEDs and buttons. Moreover, it features a camera module and an analog CODEC IC, connected to an onboard microphone. An explicative list of the onboard sensors is showed in Figure 4 with a plethora of other sensors, among which an RGB camera and a two-channel microphone, which can be exploited for impressive demos ;)

3.2 Software

Analog Devices released a toolchain to train 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 repository are of special interest: ai8x-training and ai8x-synthesis, which we are going to use later in the exercise.

Figure 3 reports the development flow of a typical MAX78000 project. It starts developing the model in Pytorch using the ai8x custom functions and the dataloader for dataset of your choice. The model is then trained, with the option for quantisation-aware training, checkpoints are saved from which the model can be evaluated. If QAT has not been selected, the model has to be quantised, and re-evaluated to check that the quantisation process does not lead to an important loss in accuracy. The checkpoint for the quantised model can then moved to the Synthesis tools, where it is matched with a Model Description File (more about that later) and a sample input is generated from the Input Data Sample provided during synthesis (this sample will be the one converted in C code for the onboard evaluation). The output of the Synthetiser is then C code, which takes care of setting up the network in the accelerator and can be modified to read data from the PC or sensors and feed them into the network. Ultimately, the inference is run and the end result is provided.

4 Training a neural network

The framework of choice is Pytorch, on top of which Analog Devices has developed 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`. It will take care of instantiating dataloaders and training loops for the designed neural network.

Two folders should be particularly taken into account: `models` and `datasets`. The former contains the network description files, in the latter there are the data loaders.

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 memes classifier. The training dataset will be composed on 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 CNN on the MAX78000. And ultimately yes, it is very 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 are not much to train a neural network. However, we can augment the images trying to cover as many real-life cases as possible, to maximise the accuracy on the test set and ultimately during the final deployment. The few lines of code of the file `generate_labels.py` create a label file iterating over the four labels of the classification task. Each line of the file contains the image and the label, and will be randomly augmented in the dataloader. Also, we do not care at this stage if the images are randomised or not, since, the dataloader will take care of it.

Let's have a look at the `memes.py` file. After importing few packages there is the main class, `MemesDataset(Dataset)`. Following the structure of Pytorch dataloaders, three methods must be present to initialize, report the total lenght 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 an 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 3: Read 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 of PyTorch and can thus be confusing. It is used by the Analog Devices toolchain to identify the network, the input and outs and which function call to load data.

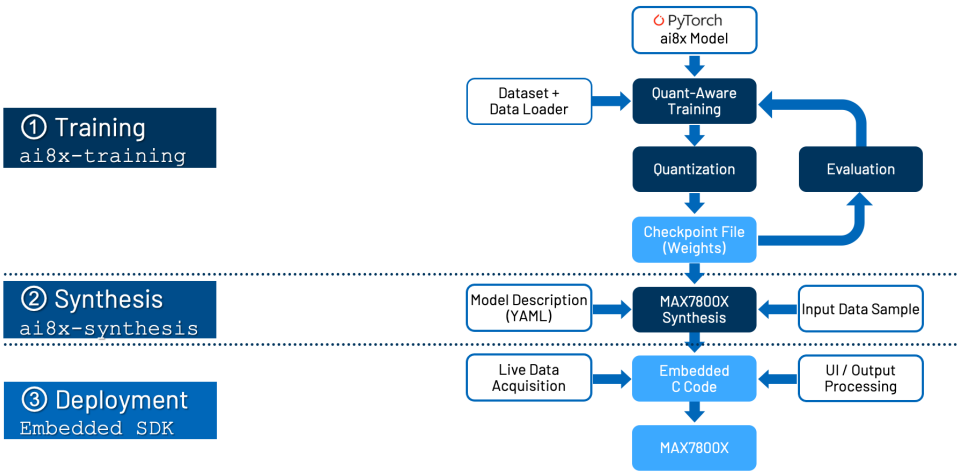

Figure 3: Overview of the software stack to train and deploy models to the MAX78000

```
"""
Dataset description
"""
datasets = [
    {
        'name': 'memes',
        'input': (3, 30, 40),
        'output': list(map(str, range(4))),
        'loader': memes_get_datasets,
    }
]
```

Deep learning is completely dependent on good quality data for training and inference. This is why it is always recommended, whenever possible, to visualise data. The last part of the file, currently commented out, creates an instance of the dataloader and visualise few augmented images. Do use plots to make sure the data you are passing to your model are actually meaningful.

4.3 Network

The network description is reported in `memenet.py`, which should be placed inside the `model` folder. This file is somehow similar to the previous, 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 `fowrard`. At the end of the file there are again few identification lines for the training script.

Student Task 4: A skeleton of a neural network is provided. Add some more layers, playing with the supported ones described in the README file.

4.4 Training script

To train the model it is best to use the ”train.py” script. All the possible arguments are listed in the github repo. Below you can find a sample command to begin the train, but do experiment with different configurations.

Student Task 5: Read the PyTorch Dataloader tutorial and get used to the Dataloader structure. Augment the Dataloader provided with other transforms.

Student Task 6: Start the training with the following command:

```
python train.py --lr 0.001 --optimizer adam --epochs 50 --batch-size 40 \
--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.

The toolchain supports different learning rate schedules and quantization aware training via YAML files. The first is specified via the `--compress` argument, while the second is specified via the `--qat-policy`. It is possible to set different learning rate schedules, as well as the epoch where to start QaT and different bits of quantisation for different layers.

5 Porting the neural network

5.1 Quantisation

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` folder. The first argument of the script is the trained network, which get saved by the training script automatically in the `log` folder inside `ai8x-training`. The second argument is the path where the script will save the quantized version of the network. The `--device` argument indicates our target microcontroller, and the `-v` flag enhance the verbosity of the script.

Student Task 7: Copy the network (`qat_best.pth.tar`) from the `ai8xtraining/logs` folder to `ai8xsynthesis/trained/`.
Quantise the neural network with the following arguments.

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

5.2 Evaluation

Inside the ”ai8x-training” folder 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 plotting a confusion matrix with the different classes:

Student Task 8: Evaluate the performance of your quantised network:

```
python train.py --model memenet --dataset memes --confusion --evaluate \
--save-sample 10 \
 --exp-load-weights-from ../ai8x-synthesis/trained/memenet_trained-q.pth.tar \
-8 --device MAX78000 "$@"
```

An important flag is ”–save-sample”. When it is used the script won’t evaluate the model on the test dataset, but will extract a sample from the test dataset and save it in pickle format. This sample can then be given to the synthesis tool to generate an header file containing the selected sample, and can be used to test the neural network against a known class as a check when the network is running on the microcontroller.

6 Synthesis

The last step of the pipeline is to generate the C files that will then be uploaded on the microcontroller.

Before we can do this, we need to update the network config file to match the implemented network.

6.1 Config file

The config file is a YAML file that contains a description of the network with some lower-level bindings for the architecture. A very good explanation for this file is given in the official documentation and in the kickstart guide.

Student Task 9: Read the documentation and and complete the YAML file proposed.

6.2 Generate C files

Once again, a script assist us in this operation: the ”ai8xize.py”. A sample command can be found below:

Student Task 10: Copy `sample_memes.npy` to the `ai8xsynthesis/tests` and synthesise the neural network:

```
python ai8xize.py --test-dir synthed_net --prefix memenet --checkpoint-file \
trained/memenet_trained-q.pth.tar --config-file networks/memenet.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.

7 Microcontroller flashing

Once all the files are created with the synthesis script, we can proceed to load them to the microcontroller.

Student Task 11: Make sure the SDK is installed: flash the microcontroller.

Let's 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 changing it is most likely better to re-generate them and not modify them by hand.
- `main.c`: file containing the `mail()` function.
- `Makefile`: file containing the instruction on how to compile the program.
- `sampledata.h`: file containing a sample from the test dataset 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.c`: Header file containing the actual kernels.

There are two ways to compile and load the program on the microcontroller. The first way is to use the GUI, which is downloadable on the official page, the second way is to compile the files using the Makefile and loading the binaries with GDB. If you have followed the installation steps proposed at the begin of the exercise, you should have at least one of the two ways set up.

Using the GUI is be pretty straightforward and, being based on Eclipse as CubeMX, should be a familiar environment. For all the command line adventurers, I can recommend the following commands inside the GDB shell:

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

To run the GDB server there is a pre-packaged OpenOCD distribution within the ai8x-synthesis repository, as explained in the README. You might have to install some dependancy in order to run OpenOCD, below the most-likely needed:

Mac:

```
brew install libusb libftdi hidapi
```

Ubuntu:

```
sudo apt install -y libusb-0.1-4 libhidapi-hidraw0
```

Student Task 12: Compile, load and run the generated code on the microcontroller. Open a serial monitor, you should be prompted the result of the test inference, as well as an inference time measurement.

8 Optional: Interfacing with the camera

Congratulations! You managed to reach this point in the exercise :D

Here we will attempt to ”close the loop” by feeding fresh data collected by onboard sensors to our deployed machine learning model. You already got an introduction on the board, and here you can find more documentation, including the camera schematics. We will not dwelve too much with hardware tho, since Analog Devices did the heavy lifting by writing the drivers for us.

Assuming you correctly installed the *ai8x-synthesis* package, and that also the *SDK* has been pulled already from GitHub, we can go ahead and check out the *SDK folder*. The most important folders inside are the *Examples* folder and the *Libraries* folder. The first one containes a lot of nice examples that Analog Devices wrote and tested for us, the second contains the actual drivers, startup files etc.

We are interested in the camera interface, so let's go to the folder *sdk/Examples/MAX78000/CameraIF*. Here you can see the standard content of an example for this *SDK*. In particular:

- *.c: files containing the source code. At the beginning of the '*main.c* usually there are some defines to specify different behaviour of the firmware.
- *.h: header files.
- Makefile: the makefile to build the firmware. I highly reccomend going through it at least once to better understand where the source of the drivers are located and which are th compilation option selected.
- pc_utility: this folder containes a small python script

In the source code for this lab you will find a version of this example simplified and stripped down of much code. In particular, the python ”front-end” has been rewritten and thoroughly commented. The python script now implements a simple handshake algorithm and puts a lot of attention on how to handle the bitstream coming from the camera.

Student Task 13: The goal of this task is to analyse the data analysis in python and reproduce it in C code.

- Compile and flash the code given in the lab documents.
- Open *img_reader.py* and understand the flow of the code.
- Change the com port and run *img_reader.py* once to see if it is actually performing as it should.
- Isolate the pre-processing algorithm, contained in the function `parse_image_rgb565`.
- Write the same algorithm in C. Data are retrieved from the camera in the function `camera_get_image()`. Go and show your pointers' skills!

Once the data is correctly parsed inside an array in C, it's time to include the neural network files generated in the previous point and feed the data to the neural network.

Student Task 14: The goal of this task is to analyse the data analysis in python and reproduce it in C code.

- Take the files generated before while synthesising the neural network and add them to the current project. Does the Makefile compile also these files?
- Include the corresponding headers in the main.c file.
- Copy the array obtained in the previous Task in the neural network accelerator memory.
- Compile and run inference!

And now you should be done! Congratulations for having a system that capture an input, pre-processess it, runs an inference on it and display the end result of it. That's the full pipeline of a TinyML system :)

9 Optional: NAS

Neural architecture search (NAS) is a technique used to automatically design neural network architectures that can outperform human-designed architectures in terms of accuracy and efficiency. The One for All (OFA) NAS was first introduced in 2020 with the paper: Once-for-All: Train One Network and Specialize it for Efficient Deployment and it is a popular and promising technique that has gained significant attention in the deep learning community. The OFA NAS involves training a fairly large neural network from which an evolutionary algorithm extract smaller networks which are optimised for devices with different computational resources.

9.1 NAS details

OFA is a type of Neural Architecture Search (NAS) method that allows for the direct deployment of a trained model to different hardware, without the need for retraining. This is achieved by training a supernet which contains all possible sub-networks that can be generated within a given search space of architectures. The sub-networks in the Once-for-All network share weights, which reduces the computational cost of training all the possible architectures individually.

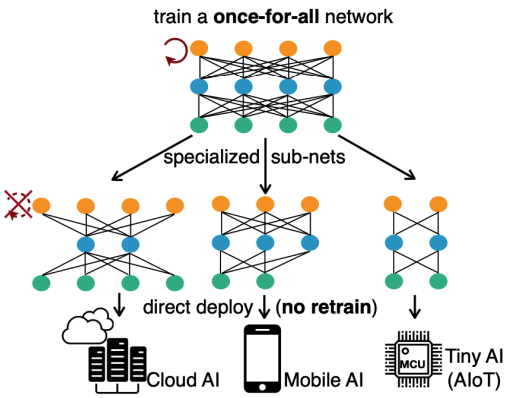

Figure 4: Overview of the OFA NAS.

Once the supernet is trained, only a portion of it is deployed on a target device, depending on its computational constraints. This is accomplished by selecting a sub-network from the supernet that meets the hardware constraints and then fine-tuning it. The fine-tuning process is faster than training a new architecture from scratch, as the sub-network shares the weight with the supernet which has been already partially trained.

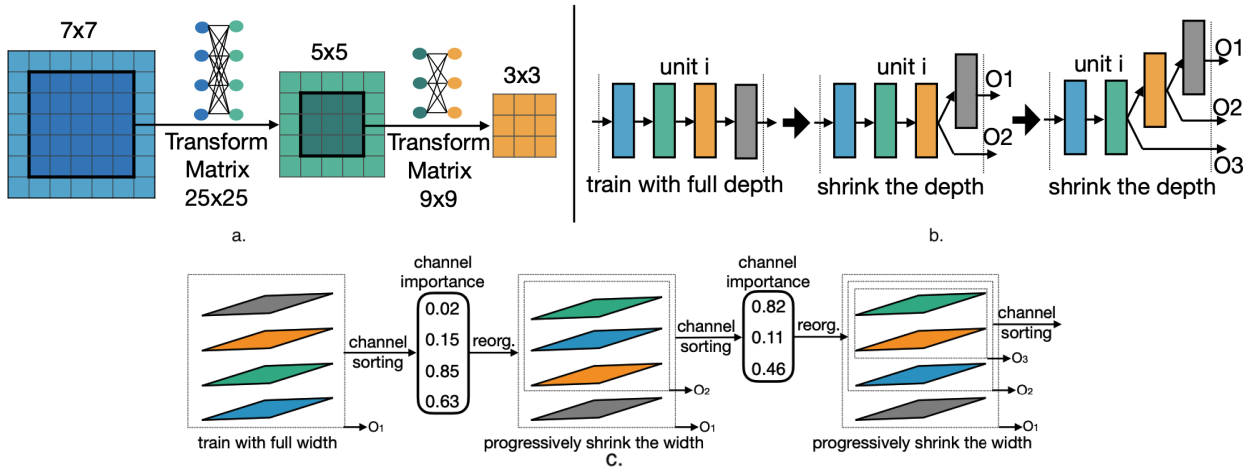

Figure 5: Model provided for the OFA NAS.

Training all the possible sub-nets can be computationally prohibitive, as there can be a large number of sub-networks within a given search space. To address this issue, the authors propose a progressive shrinking algorithm which involves training the largest network with the maximum kernel size, depth, and width, and then progressively training smaller sub-networks that share parameters with the supernet. The smaller sub-networks are initialized with the most important parameters learned from the larger network. This approach reduces the computational cost of training all possible architectures within the search space.

The progressive shrinking algorithm includes three stages: elastic kernel, elastic depth, and elastic width:

- Elastic Kernel: In the elastic kernel stage, the kernel size is chosen from a set of pre-defined options, while the depth and width are kept at their maximum values. This stage involves choosing the most appropriate kernel size that fits the hardware constraints. By allowing the kernel size to be selected from a set of options, the algorithm can select the most efficient kernel size that fits the target hardware. A visual representation of the elastic kernel phase is visualised in 5a. In the OFA implementation from Maxim, the hardware limitations for kernel sizes are taken into account, limiting the possible kernels to 3x3 and 1x1 for Conv2D and 5x1, 3x1 and 1x1 for Conv1D.
- Elastic Depth: In the elastic depth stage, layer depths are sampled. This stage allows the network to find the optimal depth of the model that fits the target hardware constraints. Reducing the depth of the network, the algorithm can find the most efficient architecture that satisfies the hardware constraints. This stage is particularly useful when the available memory on the target hardware is limited. A visual representation of the elastic kernel phase is visualised in 5b.
- Elastic Width: In the elastic width stage, sub-networks are sampled changing the width of the layers. For each layer the most important channels with the largest L1 norm are selected to ensure that only the most important channels are shared. The input channels of the following layers are sorted similarly to keep the supernet functional. This parameter is of particular use when the activation memory is scarce. A visual representation of the elastic kernel phase is visualised in 5c, it is important to notice how the layers (represented with different colors) get swapped as the algorithms proceeds.

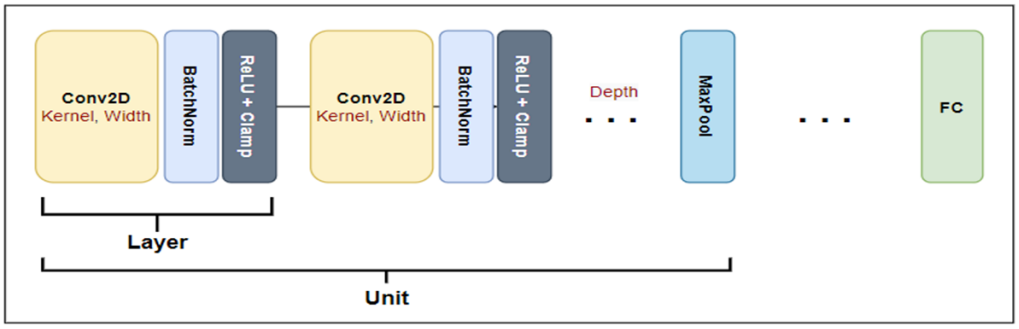

Figure 6: Model provided for the OFA NAS.

9.2 MAX78000 implementation

Analog Devices implemented a OFA NAS in their repository. The usage is described in the NAS section of the readme, below we will go more in detail with the implementation.

In order to enable the OFA NAS, custom models must be implemented on the `OnceForAllModel` model, which includes a series of functions to retrieve information about the starting model. Luckily, Analog devices provides two base models for 1D and 2D classification, which structure is reported in Figure 6. They are composed of a user definable series of *Layers*, composed of a Conv2D, BatchNorm and ReLu activation, that with a maxpool layer form a *Unit*. A model can have different units and every unit can have different layers. The set of units is then followed by a set of fully connected layers which form the classifier.

A sample model description file can be seen in `models/ai85nasnet-sequential.py`. Since the layer scheme proposed by Maxim is a good fit for our task, we can go ahead and implement our NAS definition borrowing the `OnceForAll2DSequentialModel`, of which you can find an example in hte function `ai85nasnet_sequential_cifar100`. It is advised to start from a network which is few times (3x-5x) larger than the target model, as the progressive shrink algorithm will take care of finding the best performing model from the supernetwork, given the hardware characteristics.

The workflow for NAS training with the Analog Devices framework can be not straightforward at a first glance. Let's review the steps needed to go from the supermodel to the optimised model:

- **Training the super-net**. This step is similar to training a normal model, but the training must inlcude the flags `--nas` and `--nas-policy`. In this stage is advisable to train a network which is few times bigger than the target network. A sample script implementing this stage is `train_cifar100_nas.sh`
- **Evolutionary search**: Once the supermodel is trained, the evolutionary search described above can start. A sample script which runs the evolutionary search is `search_cifar100_nas.sh`. An important parameter is `--num-out-archs`, which defines how many output models should the search output.
- **Quantisation**. The step before has delivered few well-performing models, now it is left to quantise them. These models can be either quantised with PQT as we have already seen, finetuned with QAT, or trained from scratch with QAT. A scirpt which retrains the models from scratch with QAT can be found in `train_cifar100_qat_nas_results.sh`

The documentation has a thorough explanation of each parameter. Some of them are not straightforward, in particular:

- Leveling: this enables the progressive sampling, meaning that the algorithm starts to sample from the options which are closer to the super model.
- Epochs (wrt leveling): it indicates the number of epoches spent at each level (meaning that the total number of epoches for that stage will become num_epochs*num of levels), if leveling is set to True. Otherwise it indicates the epoches spent in the stage.
- `Constraints`: probably the most important, it defines the constraint of the network in terms of weights and layer depth.

Student Task 15: Implement the NAS model and add the model definition at the bottom of the file. Use the following configuration:

- Use a total of 6 units
- Use the following decreasing depth: *3, 3, 2, 2, 2, 1*.
- User an increasing width from 32 to 128: *32, 32, 64, 64, 128, 128*
- Use 3x3 kernels

NAS like OFA, and in particular this implementation, can be used to design different models, targeting different deployment pareto points: memory consumption, speed of execution, energy consumption (quadrants of the accelerator can be turned off for incresead power saving). And ultimetly, can be used to search models for other platforms as well :)

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