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 5

Machine Learning on Audio Data STM Cube AI vs. Tensorflow Lite Micro

Michele Magno, PhD
Marco Giordano
Viviane Potocnik
Pietro Bonazzi

Wednesday 16th October, 2024

1 Introduction

In this exercise, you will learn how to use popular machine-learning toolchains with embedded libraries to implement deep neural networks on microcontrollers. These exercises will focus on convolutional neural networks and the Tensorflow[1] Keras[2] and PyTorch[3] toolchains. Still, the methods you will see can be extended even to recurrent neural networks and other toolchains.

We need to be very careful in porting networks to embedded hardware platforms. As we have seen throughout this course, performance on Microcontroller (MCU) and Digital Signal Processor (DSP) platforms highly depends on the efficient usage of their specialized instruction sets.

For STM devices, the STM Cube environment can be expanded with the Cube AI module, which offers one way of implementing neural networks. As machine learning on microcontrollers is a ”hot topic” with many parties launching new frameworks, it is important that you are able to make informed decisions by being able to profile different options.

For this exercise, we will take a look at using STM Cube AI and Tensorflow Lite Micro and see some practical obstacles when combining machine learning and data acquisition on a sensing platform.

2 Notation

Student Task (Sample Title): These boxes are for describing the actual task that you are supposed to complete.

Note: These note boxes are for special remarks

3 Preparation

For this lab, you will need STM32 CUBE IDE , TERA TERM or SCREEN as well as Anaconda, Python, SciPy.io, Keras, and Tensorflow $\geq$ 2.3.0. If you are using Windows, we recommend you install the Ubuntu Subsystem for Windows 10. For the lab you need the Hey Snips Dataset that you've downloaded beforehand from this link:

https://polybox.ethz.ch/index.php/s/KIGyY2lAWgXU7Xp/download

4 Post-Training Quantization

Post-Training Quantization is a method where the weights and/or activations of a previously trained network are quantized to use lower bit precision. This can both save storage space and accelerate execution with SIMD[4] instructions.

[1] https://www.tensorflow.org/
[2] https://keras.io/getting_started/
[3] https://pytorch.org/
[4] https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

Note that post-training quantization is not the same as quantization-aware training. While the post-training conversion process takes care to minimize the loss that is incurred by using lower bit precision, the network will use the same weights, just with a different value representation.

5 Tensorflow Lite

5.1 Tensorflow Lite Micro

Tensorflow Lite Micro[5] is an interpreter-based framework for executing Tensorflow Lite models on MCUs and DSPs. Tensorflow Lite is not tailormade for any Instruction Set Architecture (ISA), meaning you can use it with very little adjustments on different platforms, e.g. ARM, Arduino, RISC-V and many more. Compiling the framework for a given platform requires you to build the library from source and implement some integration functions for your platform. TF Lite Micro allows developers to implement custom, optimized kernels for the operators used in neural networks. To leverage these high-performance implementations, the library has to be built accordingly. **For the sake of this exercise, we provide you with a library that has been built correctly to use the CMSIS-NN kernels.** In the appendix of this exercise you can find an installation guide for Tensorflow Lite Micro.

To deploy a Tensorflow/Keras model to Tensorflow Lite Micro, there are very few steps to follow. First, the model needs to be converted into a Tensorflow Lite model. In the same step, post-training quantization may be applied to shrink the model size and leverage SIMD instructions on embedded platforms. After converting the model, a flatbuffer is extracted which encapsulates all of the model's information and parameters in a single 8 Bit array, which the interpreter can use to run the inference.

Tensorflow Lite Micro is called by initializing a model, a `micro_error_reporter`, a `micro_interpreter` and a `micro_ops_resolver`. The micro_error_reporter reports any errors that happen while parsing or executing the model. The micro_interpreter's job is to parse the flatbuffer and the micro_ops_resolver object calls the Operator API for each operation that the micro_interpreter returns.

5.2 Generating the model

A TensorFlow Lite model is represented in a special efficient portable format known as FlatBuffers (identified by the .tflite file extension). This provides several advantages over TensorFlow's protocol buffer model format such as reduced size (small code footprint) and faster inference (data is directly accessed without an extra parsing/unpacking step) that enables TensorFlow Lite to execute efficiently on devices with limited compute and memory resources.

A TensorFlow Lite model can optionally include metadata that has human-readable model description and machine-readable data for automatic generation of pre- and post-processing pipelines during on-device inference. Refer to Add metadata for more details.

You can generate a TensorFlow Lite model by converting a TensorFlow model into a TensorFlow Lite model: During conversion, you can apply optimizations such as quantization to reduce model size and latency with minimal or no loss in accuracy.[6]

[5] https://arxiv.org/abs/2010.08678
[6] https://www.tensorflow.org/lite/guide

5.3 Examples

6 STM Cube AI

STM Cube AI is a software expansion for STM CUBE MX which is able to parse models created by different machine learning frameworks like Keras, Caffe, PyTorch, and some more and generate C Code. The tool is also able to compress models and validate the compressed model accuracy against the baseline full-precision version.

Since STM Cube AI does use its own backend and is not open-source, it can be difficult to judge the implementation quality. For pure runtime measurements, STM Cube AI provides tools that run inference and count the number of cycles.

In the following paragraphs, we walk through setting up a new project with STM Cube AI.

6.1 Setting up a new project with Cube AI

Once you have created a new project and set up the normal preferences, i.e. IDE, stack & heap sizes and so on, you need to add the Cube AI expansion to your software components.
You can do this by navigating to the `Pinout & Configuration` tab and selecting `Software Packs→Manage Software Packs`.In the tab `STMicroelectronics` install the latest version of the `X-CUBE-AI: Artificial Intelligence` software pack.

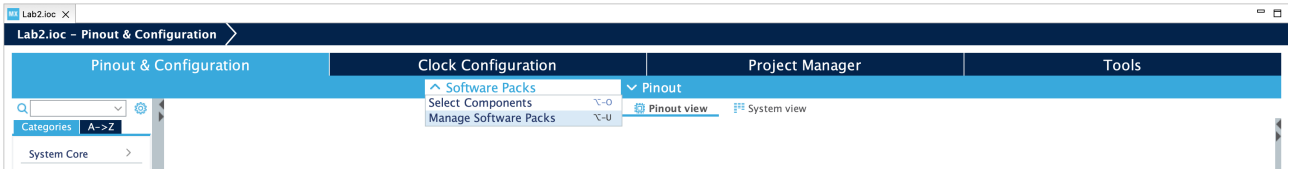

Figure 1: Software Packs

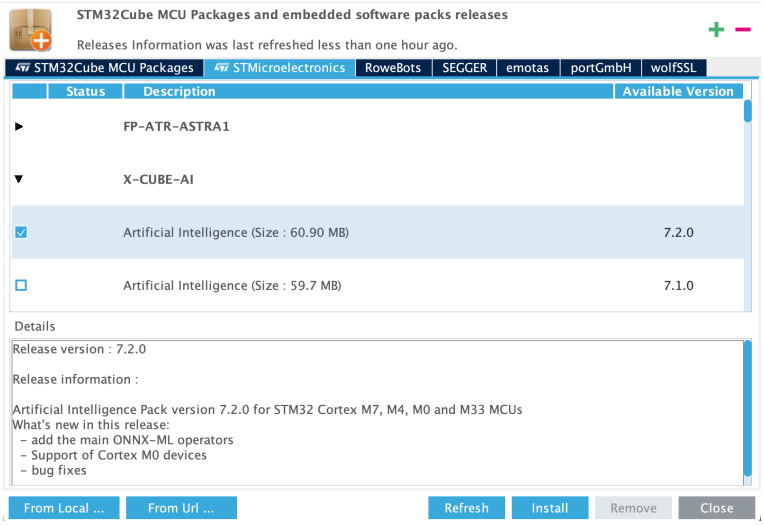

Figure 2: Cube AI Software Pack

Once the software is installed you need to activate it for your project. You can activate it under `Software Packs→Select Components`.

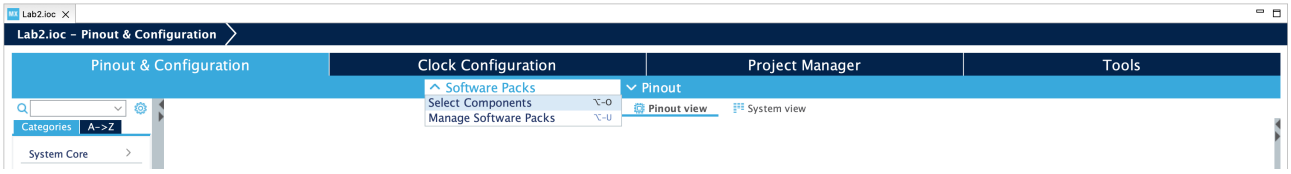

Figure 3: Select Components

Select the software packet and choose your Application type: `System Performance`, `Validation`, `Application Template`. `System Performance` and `Validation` can be used to profile your implementation, both in terms of cycles per inference and accuracy of the quantized network. The Application Template can be used to build your own application. You will get instructions in the following pages on which one to choose.

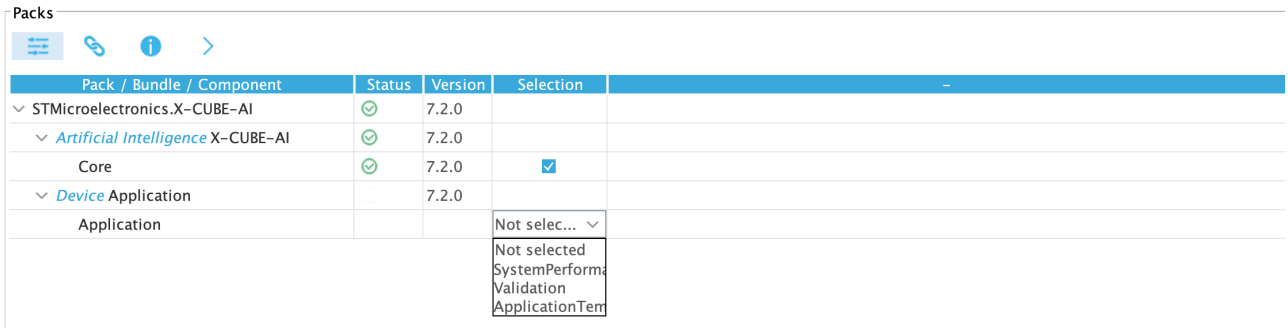

Figure 4: Choose the application type

Once you have selected the software packet, make sure that under `Software Packs→X-CUBE-AI`, both boxes are checked.

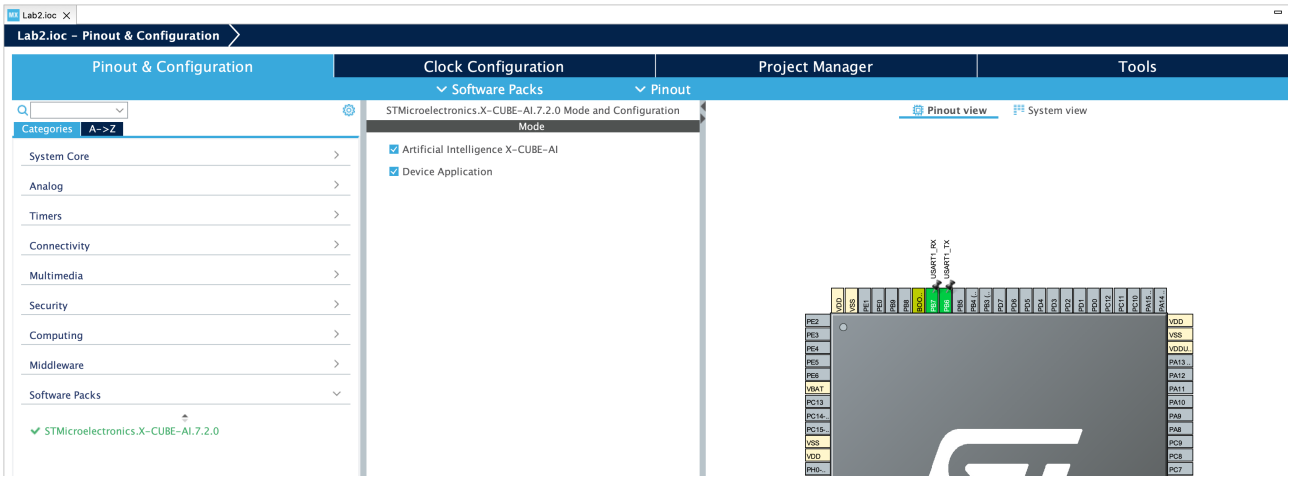

Figure 5: Cube AI Configuration window

If you have not already, set up the UART interface to connect to the USB-Port as you have learned in Lab 0. Then, under `Platform Settings`, set up the connection with Cube AI.

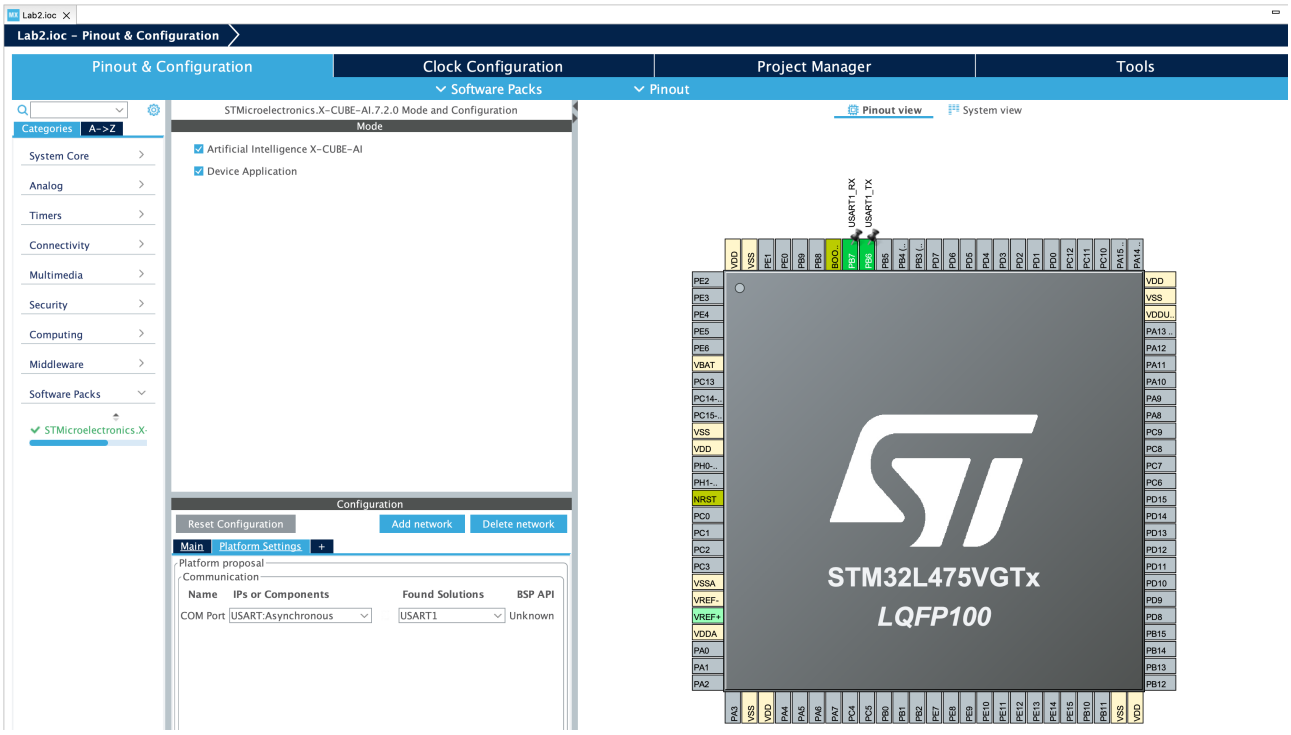

Figure 6: Setting up the communication with Cube AI

Now you are ready to load your model and work on the implementation.

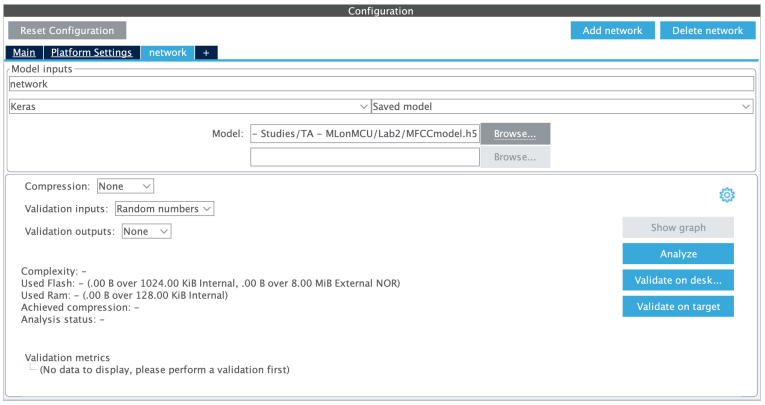

Figure 7: Load the model and set preferences

You can for example compress your model, validate it on your machine, analyze it and validate it on your MCU.

6.1.1 Some notes on compatibility

As explained before, STM Cube AI uses its own backend, therefore not every operation that you can use in Keras can be used with Cube AI. Make sure you check before you train your model. Also be careful to check the accuracy after compressing your network. For some layer types the results tend to be much worse than for others.

7 Audio recognition with Neural Networks

For this exercise we are going to use the ”Hey Snips” dataset from Couke et al. 2018 ”Efficient keyword spotting using dilated convolutions and gating”. The train dataset consists of utterances from 30 speakers, the test dataset consists of utterances from 10 speakers. The length of each utterance ranges between about 3 and 9 seconds, the data is labeled 1 if the sentence ”Hey Snips!” was uttered and 0 if not.

Note 1: The ”Hey Snips” dataset is record with a sampling rate of 16000 Hz and 16 Bit sampling depth. In microcontrollers you usually want to minimize both sampling rate and sampling depth. A possible idea for optimized MCU implementation could be to halve both sampling rate and sampling depth, since speech is mostly limited to the 0-4000 Hz frequency band.

The general difficulty with machine learning on audio is the temporal component. For images you can always assume to have the same input shape, i.e. the number of pixels and colors is constant. For audio you must assume that the features you would like to classify have variable length, as different speakers speak differently.

There are some strategies to go about this fundamental problem. One very successful approach is introducing temporal dependencies into neural networks, either by recurrent cells (LSTM, GRU or similar) or by embeddings.

Another slightly less successful (in the order of single percentage points) approach is based on convolutional networks. The idea is to segment one slice of a continuous input signal into (overlapping) segments, do some preprocessing, and stack the resulting processed segments over one-another to end up with a two-dimensional frame. On these frames the same principles are used as for ”real” images when it comes to machine learning.

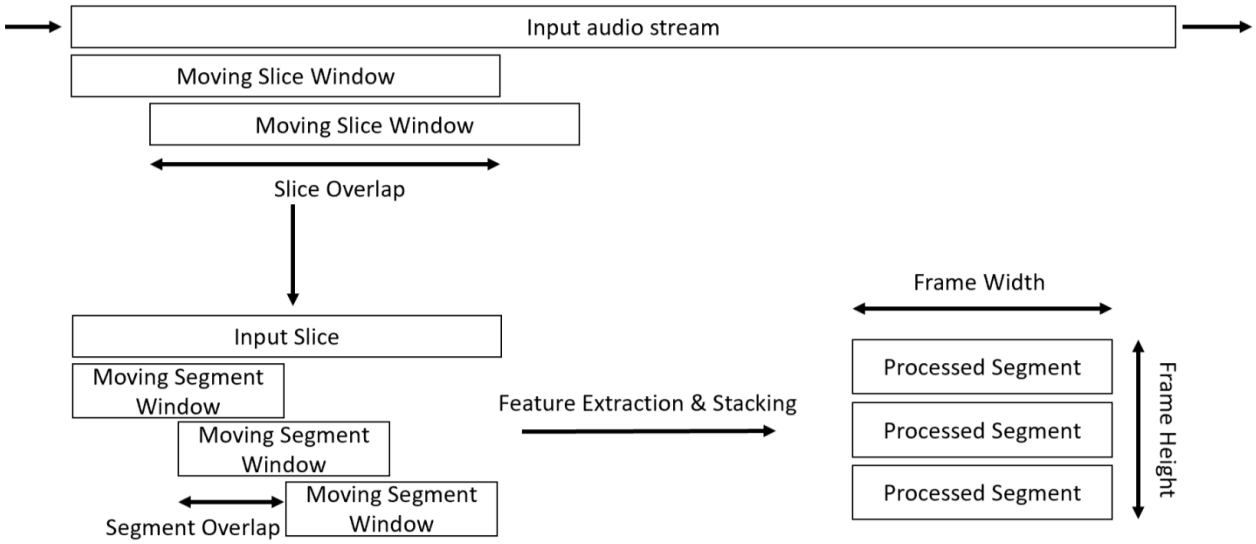

Figure 8: The workflow for audio recognition on an input stream

Many variations and combinations of both those approaches exist and there are many papers about the topic discussing advantages and disadvantages of these approaches.

The state-of-the art for the convolutional approach is structured around Mel Frequency Cepstral Coefficients (MFCC) features, which is a non-linear mapping of the frequency spectrum to considerably fewer features.

We will take a look at using CNNs to solve the audio recognition problem. We will also port the network to your microcontrollers using STM Cube AI. We will however not evaluate real world classification performance - but you can choose to work on that as a final project for this course.

Student Task 1 (Loading the databases):

- Download the "Hey Snips" dataset available here
- Copy the path of the hey_snips_kws_4.0. folder, as you will need it for loading the data in the Jupyter notebook.
- Open the notebook *./Jupyter/MFCCTraining.ipynb*.
- Run the first cell marked with Task 1 to load libraries and parse the file databases.

The first step is to zero-stuff the training and test samples so we have a constant slice length. We can then segment the slices. For simplifying this exercise, we will choose the segment overlap to be 0. We might also want our segments to have a certain length - Powers of 2 will enable us to do faster FFTs. For time reasons we will also not load all training and test samples, but a smaller subset of them.

Note 2: Zero-stuffing is the easiest way to arrive at a constant slice length. There are different more or less sophisticated approaches to achieving constant slice size.

Student Task 2 (Segmenting the input data):
Move on to the cells labeled Task 2.
Read the definition of the variable `sliceLength`. Why isn't it totalSliceLength $*$ fs?
To ensure that the slice length is a multiple of the total slice length. This makes it easier for further processing of the segments

How long (in seconds) is sliceLength now?
int(16000*10/1024)*1024 = 156*1024 = 159744 samples ..... t = 159744/16000 = 9.984 sec

Consider a frame without preprocessing. How much memory does it occupy?
1024 samples per frame and 16-bit = 2 bytes => 1024*2 Bytes = 2048 B = 2KB

Run the cells and ask an assistant if you are unsure about any of the code.

We now have a frame of raw audio data. The next step is to preprocess the data.

Student Task 3 (Extracting the features):
Move on to the cell labeled Task 3.
Complete the code to determine the compression ratio between the original train data and the MFCC-compressed data. By how much can you reduce the data size? How could you further reduce the dataset and what implication could it have on your network?
78 times. Decrease the number of mfccs, leading to decreased temporal resolution.
Lower bit representation of the same data 16 to 8.

We use the MFCC as our features. We have not yet discussed preprocessing.
What reasons can you think of to use preprocessing?

It will improve our features

Run the cell. Be patient, extracting the features can take some time.

Now we have prepared the data to be used with a neural network. The next step is to define a neural network to classify the data. Since we only have two classes (”Hey Snips” detected and ”Hey Snips” not detected), any good classifier should have very high accuracy.

Student Task 4:
Move on to the cell labeled Task 4. Define and train your network in the cell. We already set you up with a skeleton.
What accuracy did you get? 100% training, 94% test

Once you are done with your network, run the cells below to get a summary and inference accuracy on the test data set and save your network.

8 Using TF Lite Micro to port a neural network

Now that we have a TF Lite model from the previous task, start STM CUBE MX and we will set up a project using the to run inference and classify our audio data.

Student Task 5:

- Return to the Jupyter Notebook
- Run the rest of the cells to generate your network flatbuffer.
- In the STM32 CUBE IDE go to `File > Open Projects from File System` and select the `MFCC_TFLite` folder.
- After having generated the `.tflite` and `.h` files, copy the `.tflite` file to the `Core/Src` folder and the `.h` file to the `Core/Inc` folder inside the `MFCC_TFLite` folder.
- Build and flash the software project.
- Open a UART-terminal with baud rate 115200 and reset your microcontroller.

How many cycles does Tensorflow Lite Micro need for one inference?
For 1 forwards pass: 5 249 223 clock cycles

9 Using Cube AI to port a neural network

After deploying the model with Tensorflow lite Micro, let’s also take a stab at deploying it with Cube AI Micro.

Student Task 6: Use STM CUBE MX to benchmark your network.

- In the STM32 CUBE IDE create a new STM32 project as described in the previous exercises. Do not forget to clear the pinout.
- Setup USART1 in asynchronous mode, assign PB7 and PB6 respectively (as we are in

asynchronous we don't need CLK).

- Import the Cube AI package, choose the `Validation` template.
- Make sure USART1 is selected in the CubeAI platform settings as COMPort.
- Import your network, i.e. the `MFCCmodel.h5` file.
- Build the project for and make sure to set your stack and heap to 0x2000 and 0x4000, respectively.
- Save your .ioc file and generate code.

  Note 3: If you do not see a `Middlewares` folder after generating the code, go to the `Project Manager` in the `.ioc` file and select `Add necessary library files as referenced in the toolchain project configuration file` and uncheck `Delete previously generated files when not re-generated`.

- In STM32 CUBE IDE , use the VALIDATE ON TARGET button to automatically benchmark your network implementation.

How many cycles does Cube AI need for one inference?
6,556,060 CPU cycles for one inference

Compare your result to the Tensorflow lite implementation. Which is faster? By what factor?
Tensorflow lite implementation is 1.23 times faster

The number of cycles per inference is a good measure of performance because it directly relates to both execution speed and energy per inference.

You can also compress your network with Cube AI.

Student Task 7: Use compression to make your network smaller. How much memory can you save by using Low, Medium, and High compression?

Finally, you can generate an application template that can run your network.

Student Task 8:

- Choose the `Application Template` when importing the Cube AI package.
- Import your network and generate the code.
- Open the generated project.

Navigate to the main.c file to find the MX_X_CUBE_AI_Process function call. Right-click it and open the declaration to see an example of how you're supposed to use the network.

Once you've understood the general way the application template works you are ready to start your own project with STM Cube AI.

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

Appendix

Tensorflow Lite Micro Compilation

In the following, we also give a short explanation on how to reproduce and customize the flow to build TF Lite Micro.

Start from an existing unix installation, such as Linux or Mac OS. If you are using Windows, we recommend the Ubuntu Subsystem for Windows 10.

Clone the Tensorflow repository.

```
git clone https://github.com/tensorflow/tensorflow.git tf
cd tf
```

Generate some example projects for generic Cortex-M4 devices, using CMSIS-NN kernels.

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET="cortex_m_generic" \
TARGET_ARCH="cortex-m4" \
TAGS="cmsis-nn" \
generate_projects
```

This will generate a lot of example projects in *tf/tensorflow/lite/micro/tools/make/gen/cortex_m_generic_cortex\ –m4/prj* and download dependencies such as CMSIS. We will just look at one project to show how to extract the required code.

```
cd tensorflow/lite/micro/tools/make/gen
cd cortex_m_generic_cortex-m4/prj/micro_speech/make
```

Copy both the *tensorflow* and *third_party* folders into your project. These are all the dependencies you need from the tensorflow repository.

```
mkdir ../../../../../../../../../../../TFLite
cp -R tensorflow ../../../../../../../../../../../TFLite/tensorflow
cp -R third_party ../../../../../../../../../../../TFLite/third_party
```

To compile the library correctly with a current version of the ARM GNU C/C++ cross-compiler, you will have to make sure that you include the following paths for all compilers, assuming you include the TFLite directory into your project:

```
TFLite
TFLite/third_party/flatbuffers/include
TFLite/third_party/gemmlowp
TFLite/third_party/ruy
TFLite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/DSP/Include
TFLite/tensorflow/lite/micro/tools/make/downloads/
TFLite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/NN/Include
```

Further, you should set the following pre-processor defines, either in the beginning of your main.cpp file, or globally:

```
#define ARM_MATH_CM4
#define ARM_MATH_DSP
#define CMSIS_NN
#define TF_LITE_USE_GLOBAL_CMATH_FUNCTIONS
#define TF_LITE_USE_GLOBAL_MAX
#define TF_LITE_USE_GLOBAL_MIN
```

Currently, there is a bug in CMSIS-NN, where the implementation of the `__patched_SXTB16_RORn` does not compile in the *arm_nn_mat_mult_ntv_t_s8.c* file. To fix this, we suggest you replace the function in *TFLite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/NN/Source/NNSupportFunctions\/arm_nn_mat_mult_nt_t_s8.c* by the following code:

```
__STATIC_FORCEINLINE uint32_t __patched_SXTB16_RORn(uint32_t op1,
                                                    uint32_t rotate) {
  uint32_t result;
  if (__builtin_constant_p(rotate) && ((rotate == 8U) ||
        (rotate == 16U) || (rotate == 24U))) {
        __ASM volatile ("sxtb16 \%0, \%1, ROR \%2" :
            "=r" (result) : "r" (op1), "i" (rotate) );
  } else {
        result = __SXTB16(__ROR(op1, rotate)) ;
  }
  return result;
}
```

The only thing missing now is to implement the logging function. This might differ depending on your platform. In this example we use the printf function. Replace the contents of *TFLite/tensorflow/lite/\micro/cortex_m_generic/debug_log.cc* with the following code:

```
#include <stdio.h>

extern "C" void DebugLog(const char* s) {
        printf("\%s", s);
}
```

If you extracted the library from an example project, make sure to delete all the project-specific example files from the tensorflow directory.

9.1 Pitfalls with STM32CubeIDE

This subsection goes over a couple of pitfalls that we experienced during creating this exercise.

- There's inconsistent behaviour between the ARM C++ Compiler and the arm-none-eabi-g++ cross-compiler. Define the TF_LITE_USE_GLOBAL flags as described above to make the behaviour consistent.
- The `-O0` option is turned on by default; When you create a new project, make sure to set this to `-O3` in `Project→Properties→C/C++ Build→MCU GCC/G++ Compiler→Optimization`.

- All changes you make in `Project→Properties→C/C++ General` apply only to one configuration by default. Make sure to add all your settings to all configurations to avoid mismatching behaviour between Building, Running and Debugging an application.
- By default, no directories that you include are compiled by STM32CubeIDE. Make sure you add all your source directories in `Project→Properties→C/C++ General→Paths and Symbols→\ Source Location`