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 6

Quantization and Pruning for Image Classification on STM32U5

ETH Center for Project-Based Learning

Wednesday 22nd October, 2025

1 Introduction

During the last exercise, you learned how to use the *TFLite* toolchain to execute TensorFlow Lite models on different Microcontroller (MCU) targets. We will come back to the same approach for the `B-U585I-IOT02A` board and perform real-time inference on the MCU. This is done using a simple *Python* script that establishes a Universal Asynchronous Receiver Transmitter (UART) connection to the MCU, sends the input data, and receives the classification result after inference. The data transfer will be done in a static fashion, meaning we will store the data locally and load it into the MCU from a *.npy*-array file. However, for your projects, you could also use the onboard sensors or other inputs such as your laptop camera, and transmit the data using a serial port to the MCU. To prepare you for your final assignment we will follow a bottom-up approach in creating a new project enabling real-time inference on your target architecture.

2 Notation

Student Task: 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 Preparation

We will use the STM32 CUBE IDE which you are already familiar with in order to port the *TFLite* model on your microcontroller.

All Python dependencies required for feature extraction and training (e.g., `numpy`, `scipy`, `tensorflow`, etc.) are provided via a **Docker setup**.

Please make sure you have installed the environment in the first exercise on your system before you start.

Now, please activate your Docker environment and make your workspace ready.

Starting a new project

In earlier exercises, you learned how to create a new project in the STM32 CUBE IDE . Revise the previous materials if you do not remember how it is done. Carefully follow these steps and read the task descriptions thoroughly to avoid extra work.

Student Task 1 (Configuring a new project with UART):

1. Create a new project with STM32 CUBE IDE . Make sure to give the project an appropriate name. Furthermore, you have to configure the project as a `C++` project as shown in Figure 1.
2. Configure the USART1 peripheral on pins `PA9` and `PA10` as done in the previous exercises. Make sure to clear the pinout assignments beforehand.
3. In order to generate a clean `main.c` file it is recommended to generate separate source and header files for the peripherals and main file. This can be done via the *Project Manager*, as shown in Figure 2.
4. Check the files generated in the `Core` folder of the project. Do you see the different files generated for the peripherals?

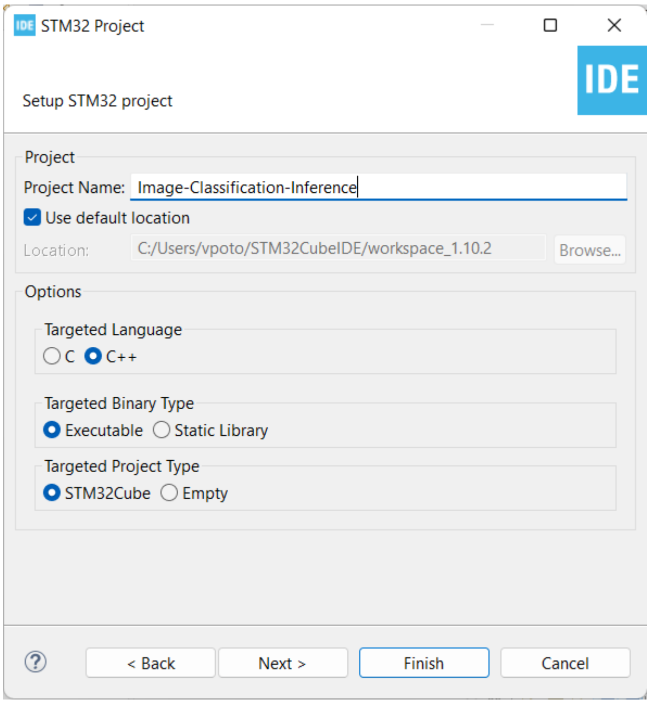

Figure 1: Creating a new `C++` STM32 CUBE IDE project from scratch for image classification on the MCU.

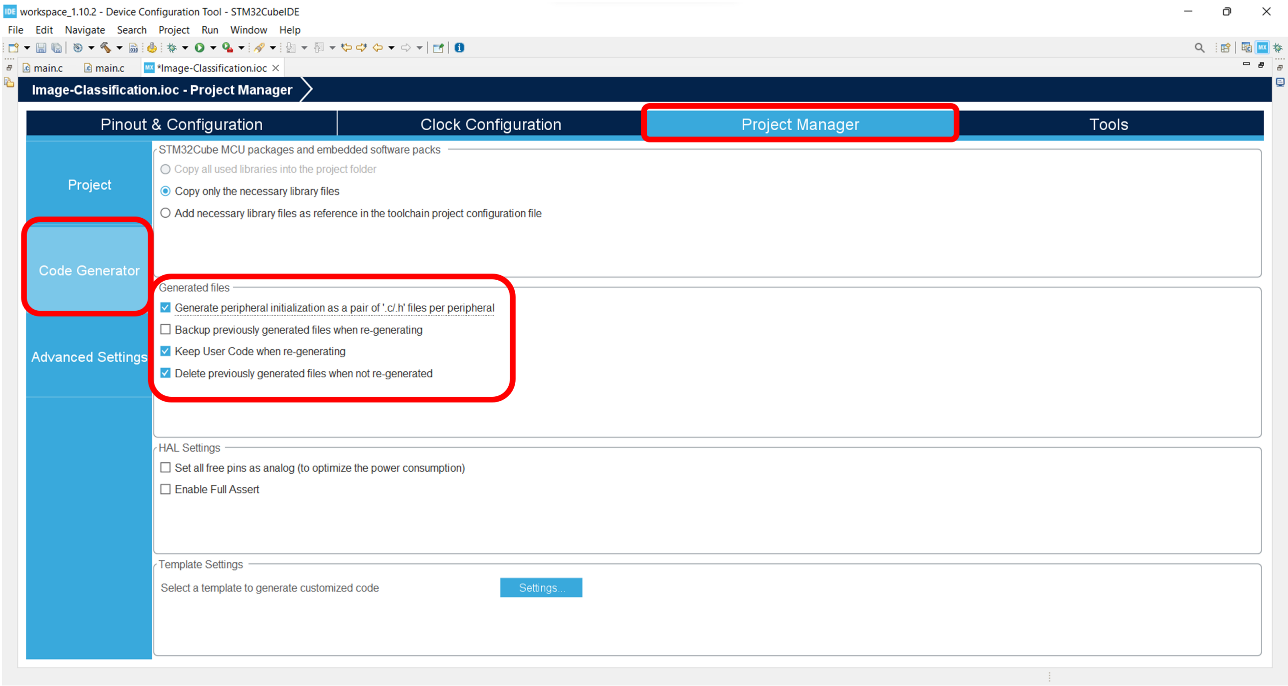

Figure 2: Project Manager configuration for separating the peripherals and main function.

Compiling and flashing a project

In the previous exercises, it was stated that the project can be compiled by *simply clicking the* BUILD *button* as shown in Figure 3.

However, this is not entirely true as you have to be very careful when starting your own exploration. We will give you a basic intuition of what can go wrong and how to fix it. In Figure 4, a general compilation flow is shown. We start with a simple `C` program. `C` is a *compiled language*, i.e. a compiler is needed to translate source code to machine-readable code. One of the most famous ones is the GNU Compiler Collection (GCC), which is also used in the STM32 CUBE IDE . Compilers translate source code in four steps: 1) Preprocessing, 2) Compiling, 3) Assembling, and 4) Linking. If you want to read up on the different steps you can refer to this blog post.

We will focus mainly on the **Linking** part, as this is usually where things go wrong. The **Assembler** goes through all the files in your `C/C++` project and generates so-called *object code (binary)*. This is essentially pure machine code, which runs on your target architecture. However, since we have multiple files in our project such as header files, or external libraries, we need the **Linker** to combine everything into a single *executable*. As the name suggests, this file contains the code which is in the end run on the MCU to perform the inference.

However, the **Linker** needs some help when using libraries and external files such as *CMSIS* or *TFLite*. Both tools contain a hierarchy of definitions and macros. Macros are essentially function definitions used across several source files. These definitions are usually all listed in so-called header files. You request the use of a header file in your program by *including* it, with the C preprocessing directive `#include`. Thus, you can access the macros within the *included* header file.

In Figure 5, an example of hierarchical header files is shown, as they occur e.g. in the *CMSIS* library. *header_three.h* has the lowest hierarchy and includes macros from the other two header files. In order for a project to successfully compile we have to tell the **Linker** the order of macro definitions. Otherwise, it will try to link functions that have not yet been defined. If you encounter error messages during building such `undefined reference to XYZ` it is probably due to misordering of the paths within your project. You can check the order of the search paths under `Properties→Paths and \ Symbols→Includes`.

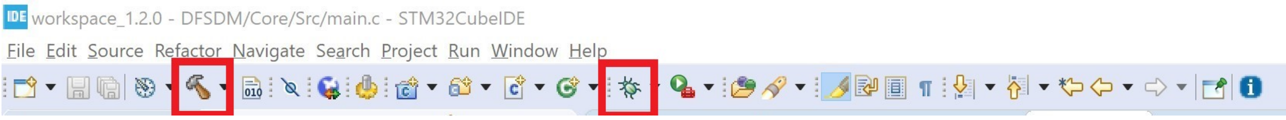

Figure 3: The Build and Debug icons in the Toolbar.

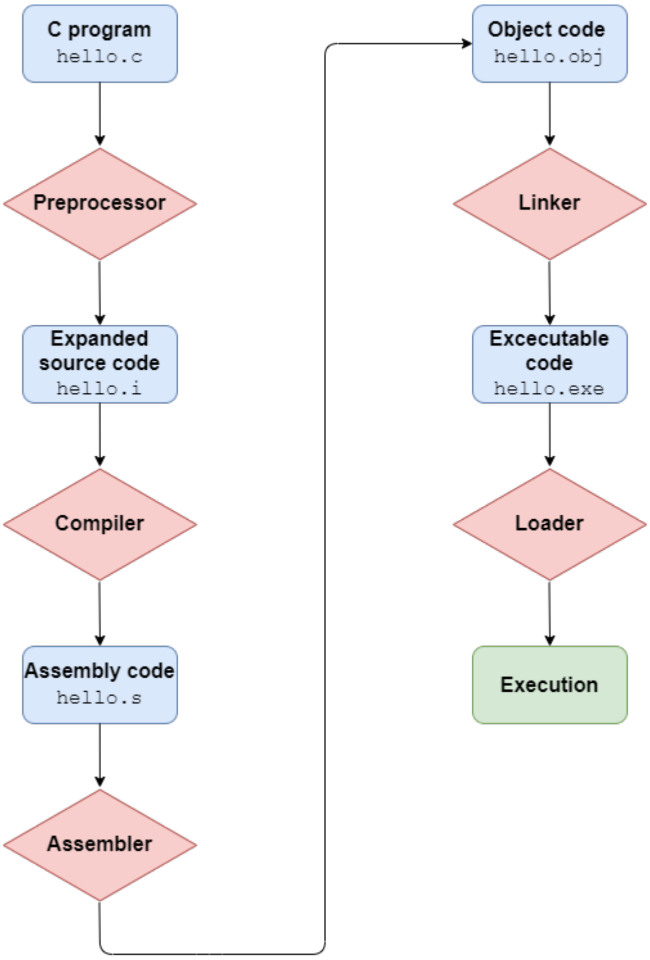

Figure 4: General compilation flow.

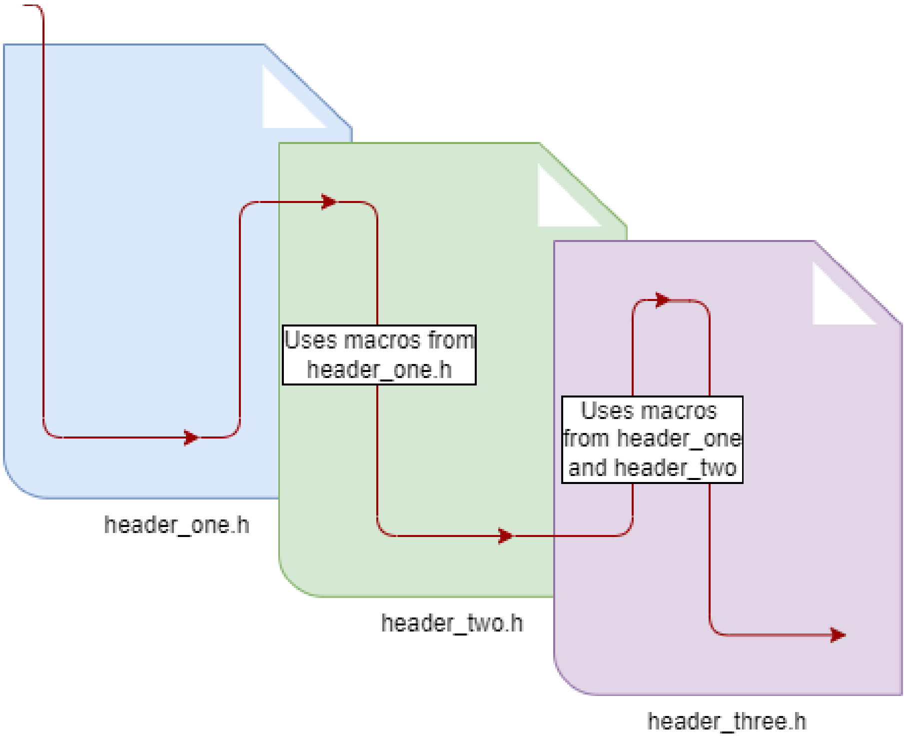

Figure 5: Header file structure example.

Student Task 2 (Project preparation):

1. Download the exercise materials from the course page.
2. Copy the `tensorflow_lite` folder to the projects root.
3. Copy `CMSIS/DSP`, `CMSIS/NN` and `CMSIS/Core` to the existing `Drivers/CMSIS` folder.
4. Make sure that your search paths for compilation and linking are set up correctly.
5. Under `Core`→`Inc` add a new folder called `models`. This is where we will store the TFLite model header files for the inference step on the MCU.

Note: When adding the includes, do this through the IDE's file browser. Also check that you have a Symbol `CMSIS_NN` and make sure to add `tensorflow_lite` as a source location in the `Paths and Symbols` section of the Project properties.

Note: In this exercise, we have provided the *CMSIS* and *TFLite* toolchains for you. If you want to update the toolchains for your own projects you can find both CMSIS and TFLite on GitHub. It is usually good to check these repositories from time to time, as more and more kernels are added to these toolchains, which could improve your performance and accuracy significantly.

3.1 Printing output with UART

Since we want to be able to communicate via UART we will add a function to write characters to the serial communication stream. Thus, we can communicate out of the microcontroller via UART. This can be useful for debugging purposes. **Do not** flash the MCU yet.

Student Task 3:

1. Revise the previous exercise material on how to include the `stdio.h` header.
2. Add a function definition for `_write` to write to the output stream in the appropriate code section of the main file.
3. Add a `printf` statement to the main function body to announce the beginning of the inference; this will furthermore test the serial communication.

4 Preparing the Inference step on the MCU

In the exercise materials we provide you with a `Jupyter notebook` to convert a trained *Keras* model via *TFlite* to a *C header file*. The header file contains all the network parameter information used for *inference*. We will start with a very simple dataset, i.e. the Modified National Institute of Standards and Technology (MNIST) dataset and a straightforward network implementation. Afterwards, you will implement your own network, targeting classification on the CIFAR-10 dataset.

Student Task 4:

1. Open the `Jupyter notebook` file `lab7.ipynb` inside your Docker, using JupyterLab. We will start with **Task 1**.
2. What is the validation loss and accuracy that you can achieve on the test set?
   - Validation loss:

     ____

   - Validation accuracy:

     ____

3. Briefly explain the difference between the test and validation set.
   - Validation set:

     ____

   - Test set:

     ____

4. What is the size of the `.h5` model and the `.tflite` model? By how much can we reduce the model size?

- Size of the `.h5` file:

- Size of the `.tflite` file:

- Compression factor:

Now you have successfully converted your *Keras* model to a *TFLite* model for the MCU.

5 Deploying the network to the MCU for inference

Deploying the network effectively means storing the network's parameters (e.g., weights and biases), usually in the MCU's read-only memory, as well as generating the C code implementing the network's computational graph, managing intermediate buffer memory and calling (optimized) kernel implementations of individual layers. Therefore, the first prerequisite for a successful deployment is ensuring that the network's size, given the number of parameters and their precision, is smaller than the available storage space.

Student Task 5:

1. Move on to **Task 2** in the `Jupyter Notebook`.
2. Count the model parameters, considering the weights and biases within each layer. Verify your results by comparing them with the output of the `summary()` [a] method.

---

3. By how much can you reduce the model size by performing full 8-bit quantization?

---

4. What accuracy can we achieve with the fully quantized model? Why do you think the quantized model might achieve higher accuracy than the full precision model?

[a] Note that similar functionality is covered in PyTorch by the `torchsummary` library.

Another hardware-associated constraint that must be addressed when deploying a model on an MCU is represented by the memory limitations. These limitations refer to the available read-write memory used for the intermediate buffers; in the absence of tiling and multi-buffered memory accesses, the memory requirements of a network can be defined as:

$$M = \max_{l \in L} l_{in} + l_{out} + l_p \tag{1}$$

, where l_{in} represent the input activations of a layer l of a network comprised of L layers, l_{out} represent the output activations, whilst l_p are the layer's parameters. Similarly to the storage requirements, the memory limitations also depend on the precision used to represent the data.

Student Task 6:

1. What are the memory requirements of your network? Do they fit the constraints of your target platform?

In the next step, we have to include the trained network in our project and deploy it on the MCU.

Student Task 7:

1. In the exercise material we provide you the application source code and header file to run the inference on the MCU. Add the `.h` and `.cpp` file at the right locations to your project.
2. Add an `include` directive in the main function body for the application and invoke the `application` macro at the right location of the code.
3. Add the model `.h` file at the correct location to your project that you generated from the `Jupyter notebook`. Make sure you include the model in your application.
4. Now you can compile and flash your project. Once you verify that the build is successful and the application is started, the `printf` statement from the main function body, announcing the inference step, can be commented out.

Note: To measure the inference duration of your model use the *CycleCounter* as before. To use it in `app.cpp` you have to include it using `extern "C" {#include "CycleCounter.h"}`. This will ensure that the functions declared in `CycleCounter.h` are treated correctly by both C and C++ compilers, resolving the conflicting declaration error.

Note: Our real-time inference script in the next tasks reads the bytes directly from the serial stream. Make sure that you do not have any `printf` statements such as e.g. from the *Cycle-Counter* in your code that contaminate the stream.

6 Real-time inference on the MCU

Finally, we can perform the inference on the MCU. For this purpose, we provide you with a small Graphical User Interface (GUI), which is programmed in the `test.py` script provided in the exercise materials. In Figure 6, the general working principle of the test script is shown. We send the test image data together with its label to the MCU. After one inference step we read out the UART port to retrieve the predicted label.

In order to achieve real-time operation for our system, our model's latency (i.e., the interval between the model receiving the input and said model producing the prediction) has to be smaller than the data acquisition time, the latter being emulated here using the `test.py` script. The inference latency can thus be considered a third hardware-associated constraint, further determining the energy consumption (i.e., $E = P \cdot t$) of our system. Although the number of Floating Point Operations (FLOPs) might represent a sufficiently good proxy for the latency when comparing different networks with a similar architecture, optimizations such as the usage of Single Instruction Multiple Data (SIMD)-based kernels, tiling, or double buffering could make the FLOPs-based comparison obsolete. It is thus recommended, when optimizing a neural network considering an accuracy-latency trade-off, to perform hardware-in-the-loop optimizations by measuring the network's latency on the target platform.

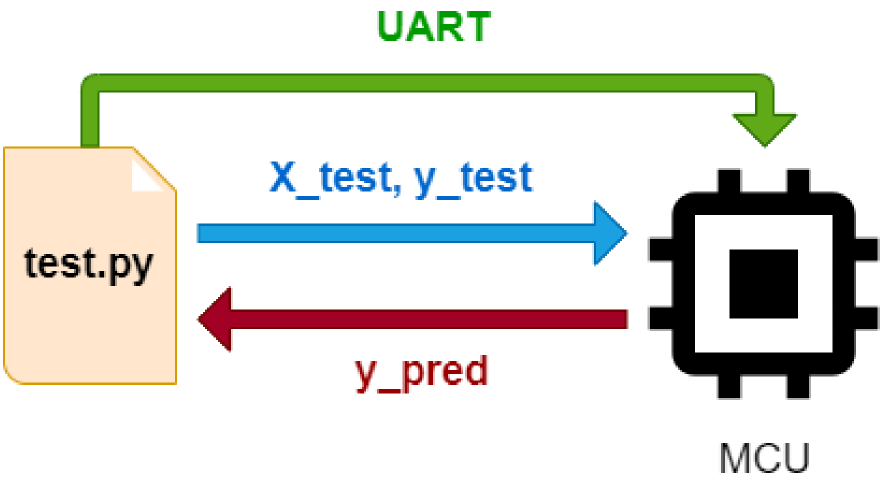

Figure 6: Real-time Inference step on the MCU.

Student Task 8:

1. Navigate to the location of your `inference.py` script.
2. Check out the source code of the `inference.py` script and try to understand how it works.
3. To perform the inference run the following command:
   `python inference.py <dataset_name>`. *Attention:* You might have to modify the script slightly, for your communication interface.
4. What is the latency and memory usage of your network?

____

Congratulations, you managed to run real-time inference on the microcontroller! As you have probably seen, the `inference.py` script supports also other datasets, such as *Fashion MNIST* and *CIFAR-10*.

7 Advanced Quantization Techniques

So far, we have seen *Post-training quantization* (PTQ). However, there exist more elaborate quantization techniques, such as *Quantization-aware training* (QAT). For our simple model we do not observe any loss in accuracy, but in more complex models, the accuracy loss can be significant. To accomodate for this, we can use QAT instead. In QAT, we train the model with quantization already in mind. The drawback of this approach is that we need more time to train the model. Depending on your application, different techniques can be more suitable. In Figure 7 we provide you with a decision tree to help you choose the right quantization technique.

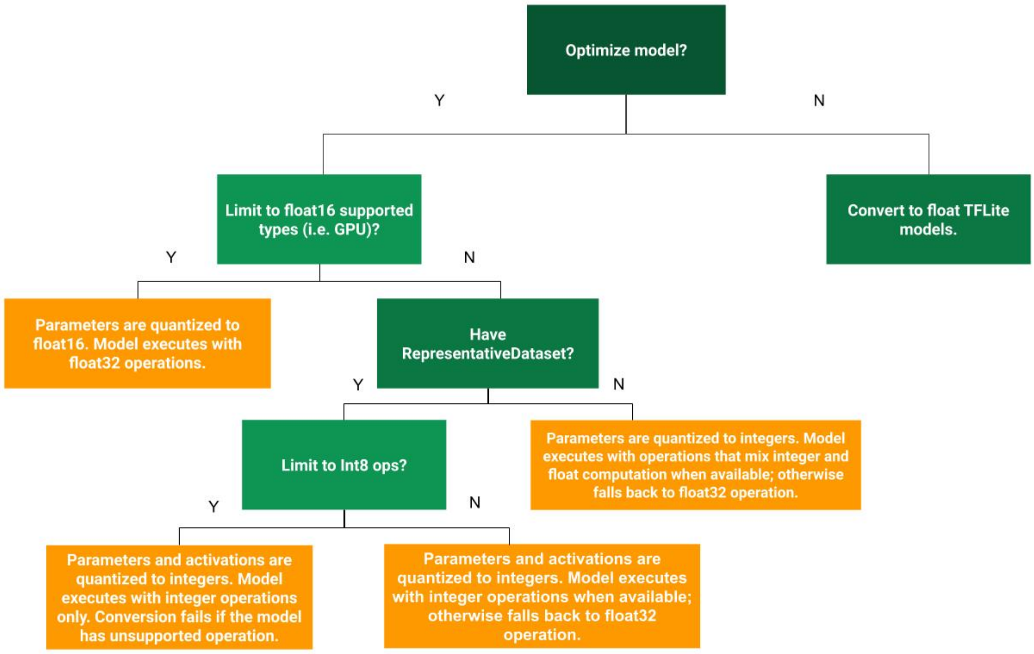

Figure 7: Decision Tree for choosing the right quantization technique. Source: TensorFlow

Student Task 9:

1. Move on to **Task 3** in the `Jupyter notebook`.
2. What are the accuracy and loss we can achieve on the test set with QAT?
   - Accuracy:

   - Loss:

3. Do we sacrifice memory for the accuracy gain? Do you think it is worth it?

8 Pruning

In the previous tasks, we have seen how to quantize a model. However, we can also reduce the number of parameters in a model. This is referred to as *pruning*. Pruning in machine learning refers to the technique of reducing the size and complexity of a trained model by removing unnecessary or redundant parameters or connections. By redundant, we try to remove parameters that do not contribute to the model's accuracy (i.e. non-critical weights). The goal of pruning is to make the model more efficient in terms of memory usage, computation speed, and energy consumption, without sacrificing its performance on the task it was trained for.

Student Task 10:

1. Move on to **Task 4** in the `Jupyter notebook`.
2. What are the accuracy and loss we can achieve on the test set with the structured pruned model using constant sparsity?
   - Accuracy:
   - Loss:
3. What is the compression factor we can achieve with structured pruning using constant sparsity?
4. Write down the results for unstructured pruning with constant sparsity.
   - Accuracy:
   - Loss:

- Compression factor:

5. Is the accuracy for the unstructured or structured pruning better? How can you explain your observations?

6. Write down the results for unstructured pruning with dynamic sparsity.
   - Accuracy:
   - Loss:
   - Compression factor:

7. Why do you think our compression factor is now lower compared to constant sparsity pruning?

9 Pruning-preserving Quantization-aware Training

We saw that we can reduce the model size by means of pruning, however, we sacrifice accuracy. In order to mitigate the accuracy loss, we can combine pruning with quantization-aware training. Thus, we can find a good trade-off between accuracy and model size.

Student Task 11:

1. Move on to **Task 5** in the `Jupyter notebook`.
2. What are the accuracy and loss we can achieve on the test set with pruning-preserving QAT?
   - Accuracy:

- Loss:

- Compression factor:

3. Try to interpret the table and plot in the *Result Summary* section of the `Jupyter notebook`.

Now you have generated all the files neccessary to run the inference of each model on the microcontroller.

Student Task 12:

1. Repeat the steps for running the inference as described in Student Task 8.
2. What is the latency of each model?
   - *Post-training quantization*:
   - *Quantization-aware training*:
   - *Constant Structured Pruning*:
   - *Constant Unstructured Pruning*:
   - *Dynamic Unstructured Pruning*:
   - *Pruning-preserving quantization-aware training*:

3. What is the memory usage of each model?
    - *Post-training quantization*:
    - *Quantization-aware training*:
    - *Constant Structured Pruning*:
    - *Constant Unstructured Pruning*:
    - *Dynamic Unstructured Pruning*:
    - *Pruning-preserving quantization-aware training*:

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