# Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI)

## Introduction

This user manual provides the guidelines to build step-by-step a complete Artificial Intelligence (AI) IDE-based project for STM32 microcontrollers with automatic conversion of pre-trained Neural Networks (NN) and integration of the generated optimized library. It describes the X-CUBE-AI Expansion Package that is fully integrated with the STM32CubeMX tool. This user manual also describes optional add-on AI test applications or utilities for AI system performance and validation.

The main part of the document is a hands-on learning to generate quickly an STM32 AI-based project. A NUCLEO-F746ZG development kit and several models for Deep Learning (DL) from the public domain are used as practical examples. Any STM32 development kits or customer boards based on a microcontroller in the STM32F3, STM32F4, STM32G4, STM32L4, STM32L4+, STM32L5, STM32F7, STM32H7, STM32WB or STM32WL Series can also be used with minor adaptations.

The next part of the document details and describes the use of the X-CUBE-AI for AI performance and validation add-on applications. It covers also internal aspects such as the generated NN library. Additionally, more information (command-line support, supported toolboxes and layers, reported metrics) are available from the `Documentation` folder in the installed package.

# 1 General information

The X-CUBE-AI Expansion Package is dedicated to AI projects running on STM32 Arm® Cortex®-M-based MCUs.

The descriptions in the current revision of the user manual are based on:

- X-CUBE-AI 6.0.0
- Embedded inference client API 1.1.0
- Command-line interface 1.4.1

The pre-trained Keras DL model used for the example in this document is:

- https://github.com/Shahnawax/HAR-CNN-Keras: Human Activity Recognition using CNN in Keras

*Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

## 1.1 What is STM32Cube?

STM32Cube is an STMicroelectronics original initiative to significantly improve designer's productivity by reducing development effort, time, and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD) powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real-time
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeF7 for the STM32F7 Series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as RTOS, USB, FAT file system, graphics and TCP/IP
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

## 1.2 How does X-CUBE-AI complement STM32Cube?

X-CUBE-AI extends STM32CubeMX by providing an automatic NN library generator optimized in computation and memory (RAM and Flash) that converts pre-trained Neural Networks from most used DL frameworks (such as Keras, TensorFlow™ Lite and ONNX) into a library that is automatically integrated in the final user project. The project is automatically setup, ready for compilation and execution on the STM32 microcontroller.

X-CUBE-AI also extends STM32CubeMX by adding, for the project creation, specific MCU filtering to select the right devices that fit specific criteria requirements (such as RAM or Flash memory size) for a user's NN.

The X-CUBE-AI tool can generate three kinds of projects:

- System performance project running on the STM32 MCU allowing the accurate measurement of the NN inference CPU load and memory usage
- Validation project that validates incrementally the results returned by the NN, stimulated by either random or user test data, on both desktop PC and STM32 Arm® Cortex®-M-based MCU embedded environment
- Application template project allowing the building of AI-based application

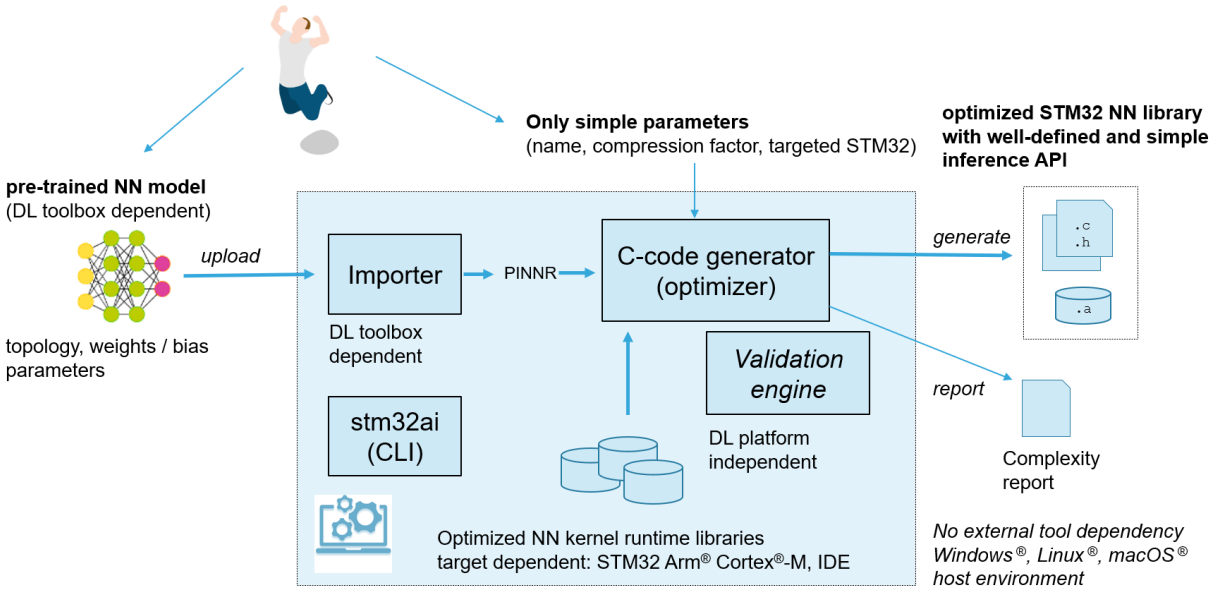
When using a TensorFlow™ Lite model, the tool can generate the code using the STM32Cube.AI library or using the TensorFlow™ Lite for Microcontrollers runtime provided in the TensorFlow™ source repository.

## 1.3 X-CUBE-AI core engine

The X-CUBE-AI core engine, presented in Figure 1 and Figure 2, is part of the X-CUBE-AI Expansion Package described later in Section 1.4 . It provides an automatic and advanced NN mapping tool to generate and deploy an optimized and robust C-model implementation of a pre-trained Neural Network (DL model) for the embedded systems with limited and constrained hardware resources. The generated STM32 NN library (both specialized and generic parts) can be directly integrated in an IDE project or makefile-based build system. A well-defined and specific inference client API (refer to Section 8 Embedded inference client API) is also exported to develop a client AI-based application. Various frameworks (DL toolbox) and layers for Deep Learning are supported (refer to Section 12 Supported toolboxes and layers for Deep Learning).

All X-CUBE-AI core features are available through a complete and unified *Command Line Interface* (console level) to perform the main steps to analyze, validate, and generate an optimized NN C-library for STM32 devices (refer to [6]). It provides also a post-training quantization support for the Keras model.

**Figure 1. X-CUBE-AI core engine**

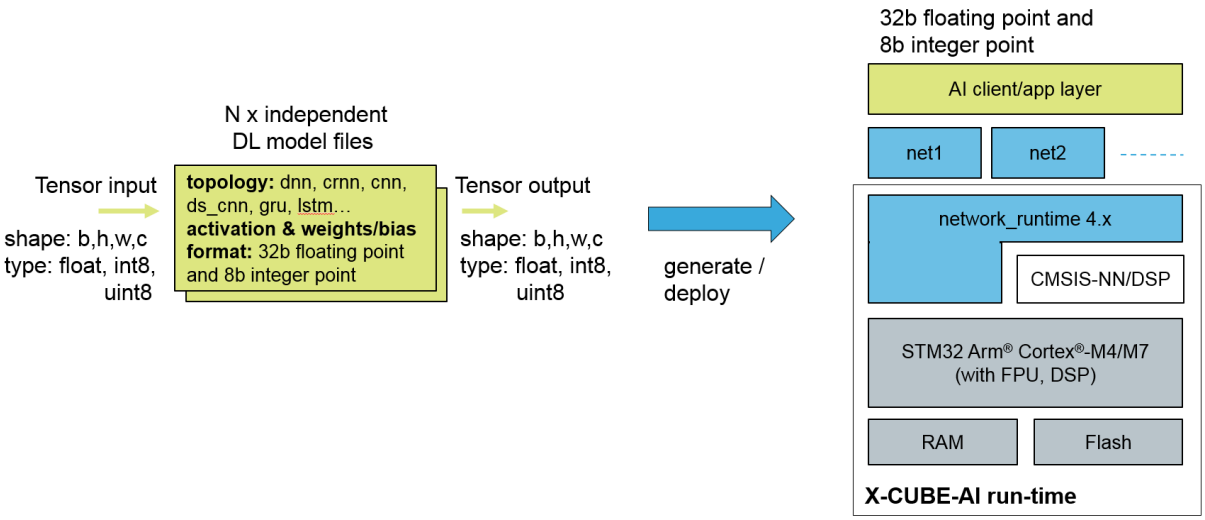



A simple configuration interface is exposed. With the pre-trained DL model file, only few parameters are requested:

- Name: indicates the name of the generated C model (the default value is *"network"*)
- Compression: indicates the compression factor to reduce the size of weight/bias parameters (refer to Section 6.1 Graph flow and memory layout optimizer)
- STM32 family: selects the optimized NN kernel run-time library

Figure 2 summarizes the main supported features of the uploaded DL model and targeted sub-system run-time.

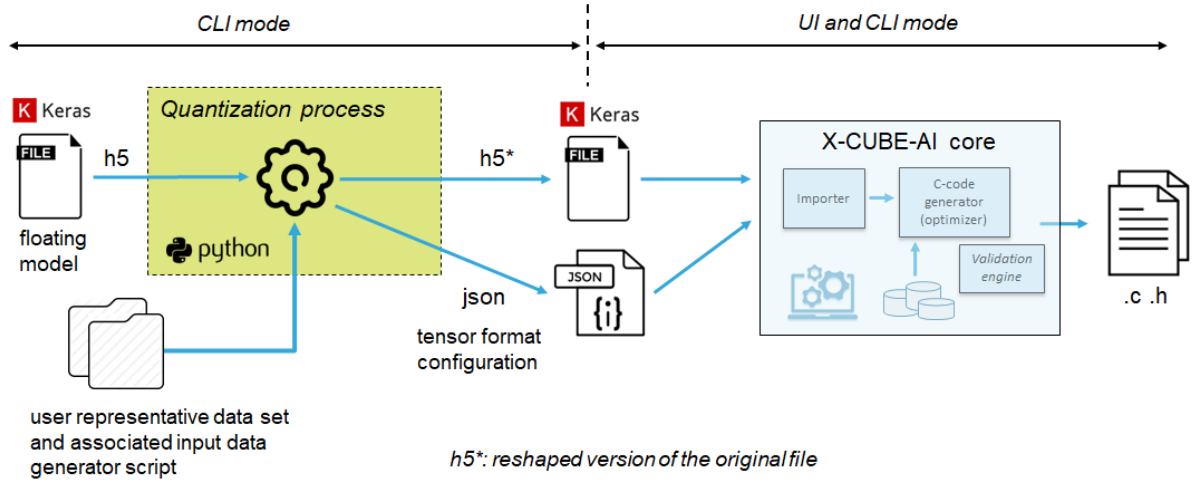
**Figure 2. X-CUBE-AI overview**




- Only simple tensor input and simple tensor output are supported
  - 4-dim shape: batch, height, width, channel (*"channel-last"* format, refer to [10])
  - Floating-point (32b) and fixed-point (8b) types
- Generated C models are fully optimized for STM32 Arm® Cortex®-M4/M7/M33 cores with FPU and DSP extensions

X-CUBE-AI code generator can be used to generate and deploy a pre-quantized 8-bit fixed-point/integer Keras model and the quantized TensorFlow™ Lite model. For the Keras model, a reshaped model file (`h5*`) and a proprietary tensor-format configuration file (`json`) are required.

**Figure 3. Quantization flow**



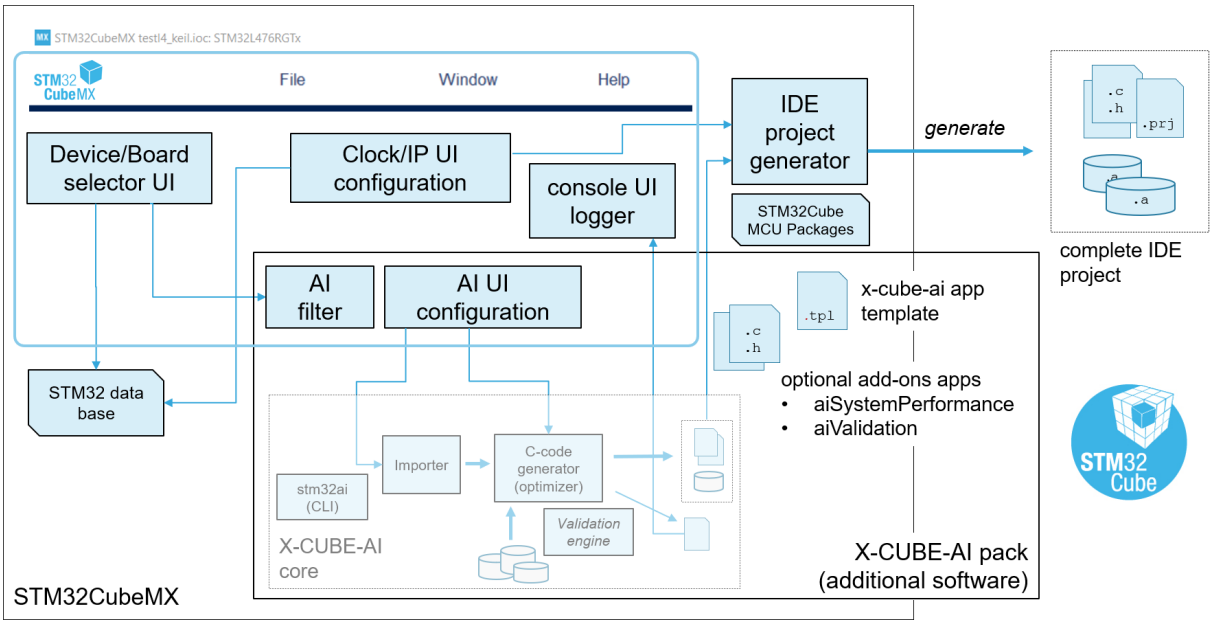

The code generator quantizes weights and bias, and associated activations from floating point to 8-bit precision. These are mapped on the optimized and specialized C implementation for the supported kernels (refer to [7]). Otherwise, the floating-point version of the operator is used and float-to-8-bit and 8-bit-to-float convert operators are automatically inserted. The objective of this technique is to reduce the model size while also improving the CPU and hardware accelerator latency (including power consumption aspects) with little degradation in model accuracy.

To generate the reshaped Keras model file and associated tensor-format configuration file from an already-trained floating-point Keras model, the `stm32ai` application (command-line interface) integrates a complete post-training quantization process (refer to [12]).

## 1.4 STM32CubeMX extension

STM32CubeMX is a software configuration tool for STM32 microcontrollers. In one click, it allows the creation of a complete IDE project for STM32 including the generation of the C initializing code for device and platform set up (pins, clock tree, peripherals, and middleware) using graphical wizards (such as the pinout-conflict solver, clock-tree setting helper, and others).

**Figure 4. X-CUBE-AI core in STM32CubeMX**




From the user point of view, the integration of the X-CUBE-AI Expansion Package can be considered as the addition of a peripheral or middleware SW component. On top of X-CUBE-AI core, the following main functionalities are provided:

- MCU filter selector is extended with an optional specific AI filter to remove the devices that do not have enough memory. If enabled, STM32 devices without Arm® Cortex®-M4, -M7 or -M33 core are directly filtered out.
- Provides a complete AI UI configuration wizard allowing the upload of multiple DL models. Includes a validation process of the generated C code on the desktop PC and on the target.
- Extends the IDE project generator to assist the generation of the optimized STM32 NN library and its integration for the selected STM32 Arm® Cortex®-M core and IDE.
- Optional add-on applications allow the generation of a complete and ready-to-use AI test application project including the generated NN libraries. The user must just have imported it inside his favorite IDE to generate the firmware image and program it. No additional code or modification is requested from the end user.
- One-click support to generate, program and run automatically an on-device AI validation firmware (including support for the external memory).
- Generation using STM32Cube.AI runtime or TensorFlow™ Lite for Microcontrollers runtime when the Neural Network file is a TensorFlow™ Lite file.

## 1.5 Acronyms, abbreviations and definitions

Table 1 details the specific acronyms and abbreviations used in this document.

**Table 1. Definition of terms used in this document**

| | |
|---|---|
| AI | Artificial Intelligence, sometimes called machine intelligence. Commonly, AI is the broad concept of machines being able to carry out tasks in a way that can be considered as “smart” from a human standpoint. It stands for the ability of a digital equipment to perform tasks associated with intelligent beings. |
| DL | Deep Learning (also known as deep structured learning or hierarchical learning). DL models are vaguely inspired by information processing and communication patterns in biological nervous systems. |
| ML | Machine Learning is an application of Artificial Intelligence (AI) that provides systems with the ability to automatically learn and improve from experience without being explicitly programmed. |
| MACC | Multiply-and-accumulate complexity is a unity that indicates the complexity of a DL model from a processing standpoint. |
| PINNR | Platform-independent Neural Network representation is a file generated by the front end (X-CUBE-AI core importer) to have a common and portable internal representation of the uploaded DL model for the next stages (optimizer and C-code generator). |

## 1.6 Prerequisites

The following packages must be installed (refer to Section 2 Installing X-CUBE-AI):

- STM32CubeMX version 5.4.0 or later
- Additional SW pack - STM32CubeMX AI (X-CUBE-AI) 6.0.0 pack
- STM32CubeProgrammer (STM32CubeProg) version 2.1.0 or later. Except when STM32CubeIDE is used, it is necessary to install STM32CubeProgrammer to be able to benefit from the automatic validation on the target.

One of the following toolchain or IDEs for STM32 must be installed:

- STMicroelectronics - STM32CubeIDE version 1.0.1 or later
- IAR Systems - IAR Embedded Workbench® IDE - ARM v8.x (www.iar.com/iar-embedded-workbench)
- Keil® - MDK-ARM Professional Version - µVision® V5.25.2.0 (www.keil.com)
- GNU Arm Embedded Toolchain (developer.arm.com/open-source/gnu-toolchain/gnu-rm)

X-CUBE-AI can be deployed on the following operating systems:

- Windows® 10
- Ubuntu® 18.4
- macOS® (x64) (tested on OS X® El Capitan and Sierra)

*Note:* *Ubuntu® is a registered trademark of Canonical Ltd.*

*macOS® and OS X® are trademarks of Apple Inc. registered in the U.S. and other countries.*

*All other trademarks are the property of their respective owners.*

## 1.7 License

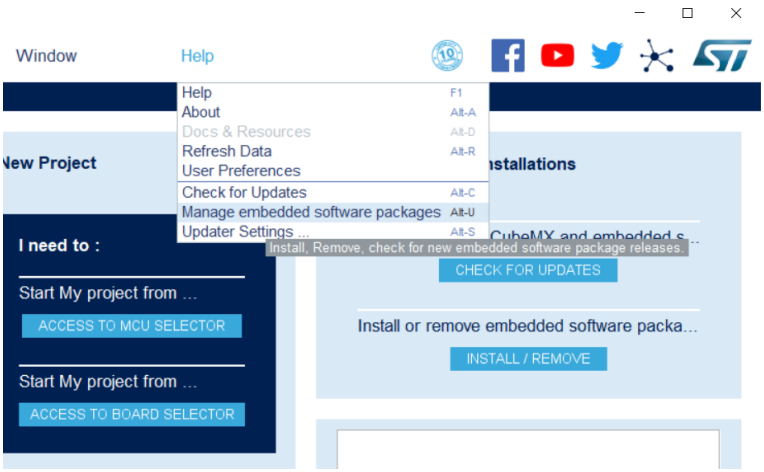
X-CUBE-AI is delivered under the *Mix Ultimate Liberty+OSS+3rd-party V1* software license agreement (SLA0048).

# 2 Installing X-CUBE-AI

After downloading, installing, and launching STM32CubeMX (version 5.4.0 or later), the X-CUBE-AI Expansion Package can be installed in a few steps.

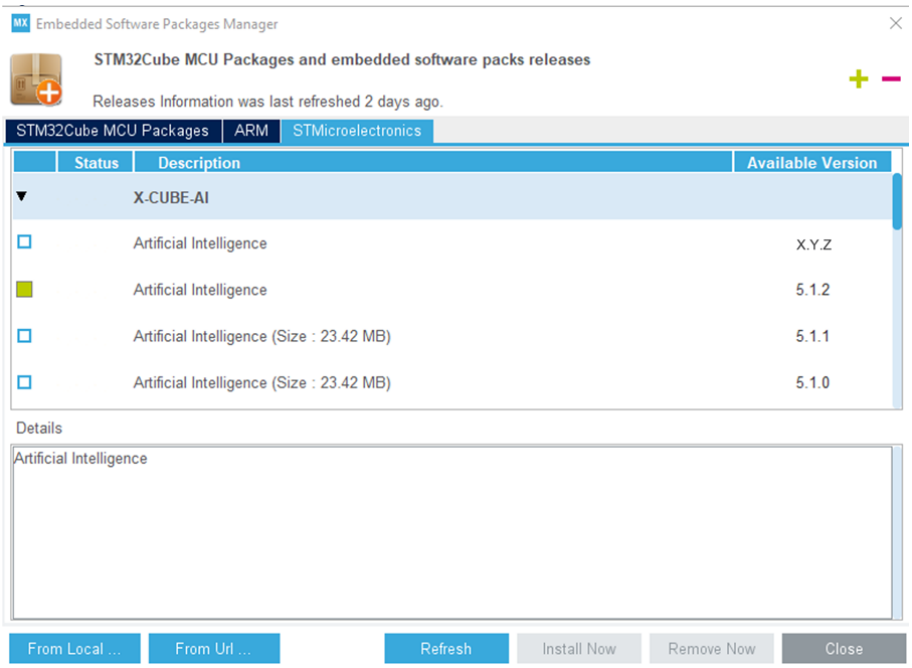
1. From the menu, select [**Help**]>[**Manage embedded software packages**] or directly click on the [**INSTALL/ REMOVE**] button.

**Figure 5. Managing embedded software packs in STM32CubeMX**




2. From the *Embedded Software Packages Manager* window, press the [**Refresh**] button to get an updated list of the add-on packs. Go to the *STMicroelectronics* tab to find X-CUBE-AI.

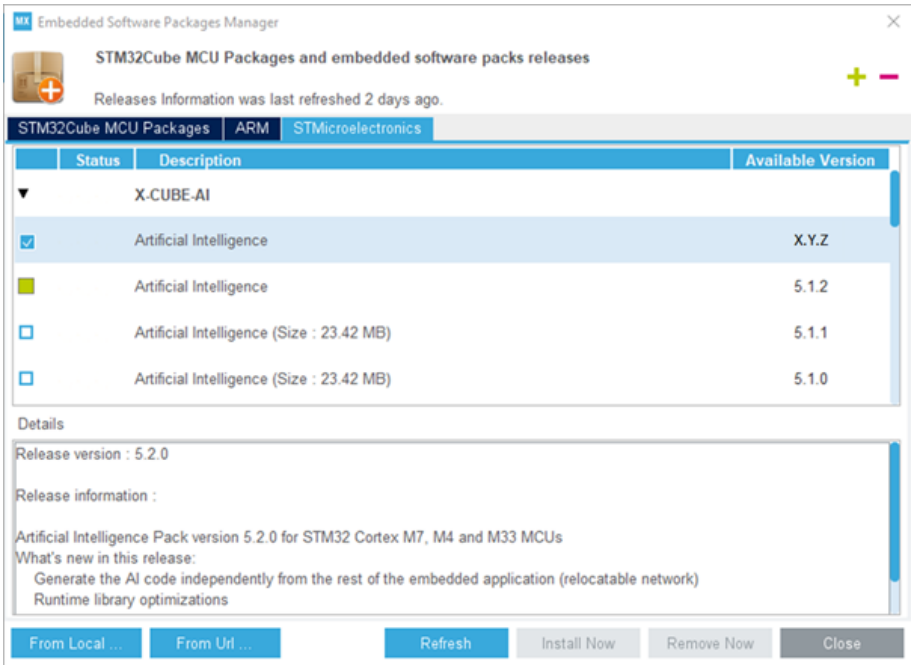
**Figure 6. Installing X-CUBE-AI in STM32CubeMX**




*Note: X.Y.Z stands for the current X-CUBE-AI release.*
*If X-CUBE-AI is already installed, preferably remove it before the new installation.*

3. Select it by checking the corresponding box and install it by pressing the [**Install Now**] button. Once the installation is completed, the corresponding box becomes green and the [**Close**] button can be pressed.

**Figure 7. X-CUBE-AI in STM32CubeMX**




The X-CUBE-AI Expansion Package contains an OS-specific part, which is downloaded at the first use of the pack.

To trigger the download, either:

- select the AI filter in the *MCU selector*
- add the X-CUBE-AI extension to an STM32CubeMX project

Alternatively, it is possible to download the OS-specific part directly from https://sw-center.st.com/packs/x-cube-ai/stm32ai-<os>-<version>.zip where:

- `os` is `windows`, `linux`, or `mac`
- `version` is the current 3-digit version (`X.Y.Z`)

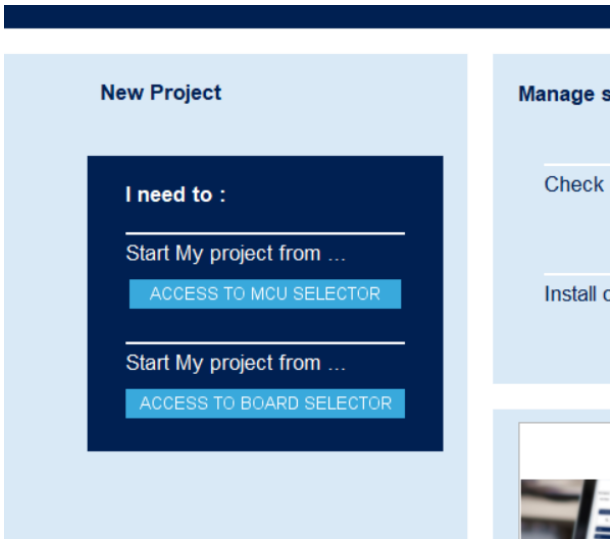
For example: https://sw-center.st.com/packs/x-cube-ai/stm32ai-windows-6.0.0.zip for the Windows®-specific part of X-CUBE-AI 6.0.0.

# 3 Starting a new STM32 AI project

## 3.1 MCU and board selector

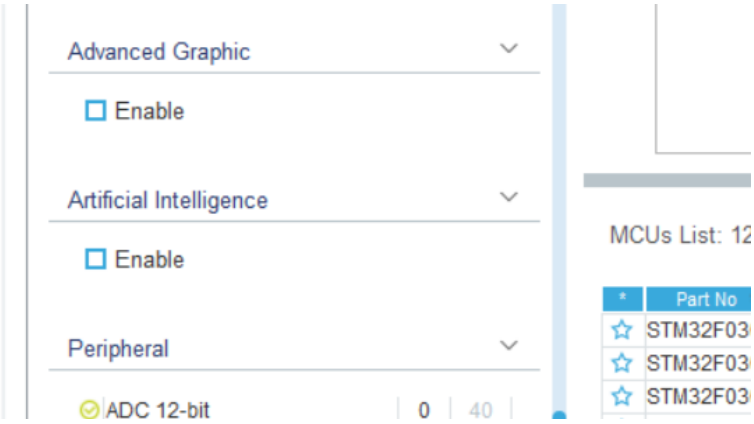
After launching the STM32CubeMX application, click on the [**ACCESS TO MCU SELECTOR**] or [**ACCESS TO BOARD SELECTOR**] button. Alternately, select [**File**]>[**New Project...**] or the `CTRL-N` shortcut.

**Figure 8. Creating a new project**




At this point, the typical STM32CubeMX flow can be used to select a specific MCU or board. An optional MCU filter entry allows the exclusion of the MCUs that do not have enough embedded memory (RAM, Flash, or both) to store the optimized STM32 NN library. This specific AI filter is shown in Figure 9.

**Figure 9. AI filter**




*Note: This feature is not available for the board selector and usable only for one NN model.*

Figure 10 illustrates the case where a DL model has been uploaded and analyzed with the default options. A pre-trained NN model (Keras type) from the public domain is used: Human Activity Recognition using CNN in Keras.

**Figure 10. AI filter with default option**

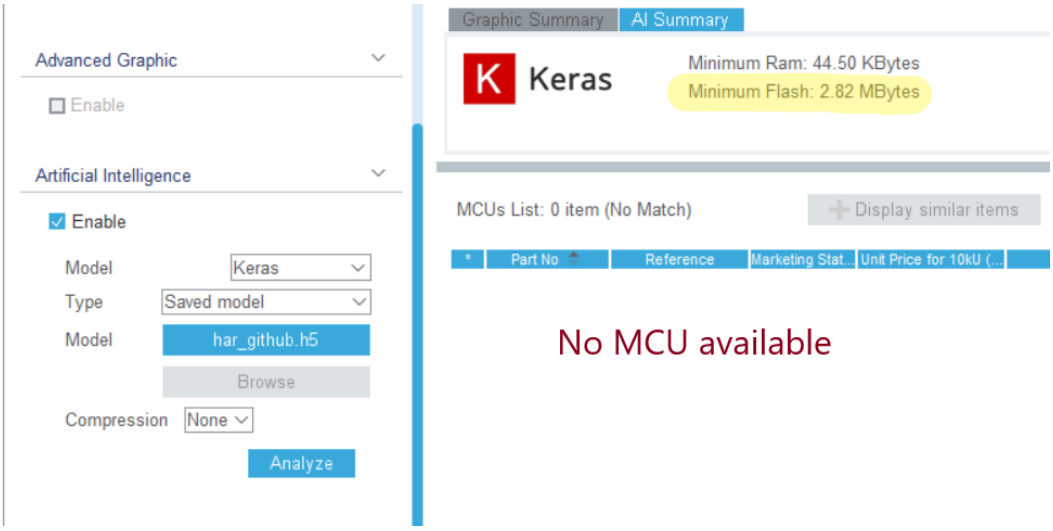
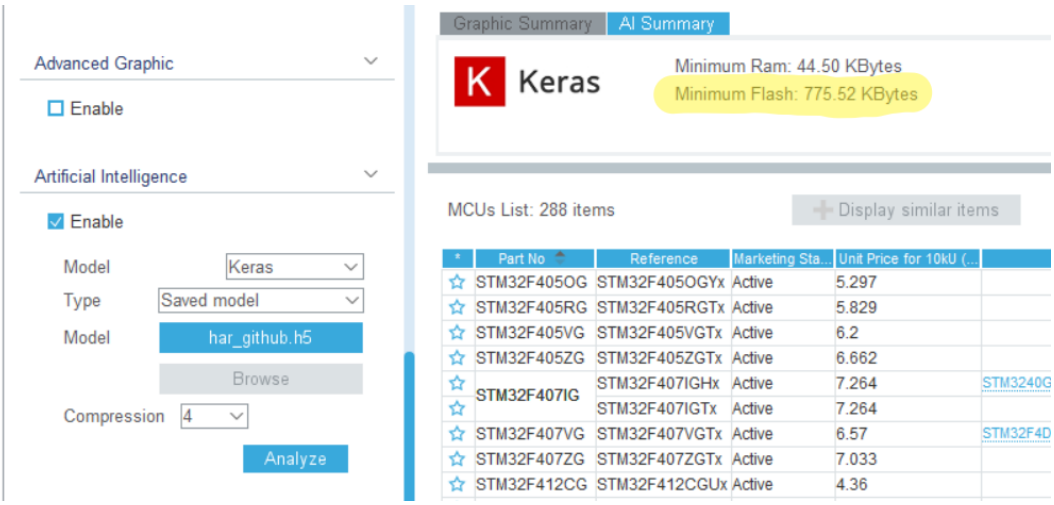



Figure 11 illustrates the case where a compression factor of 4 is applied.

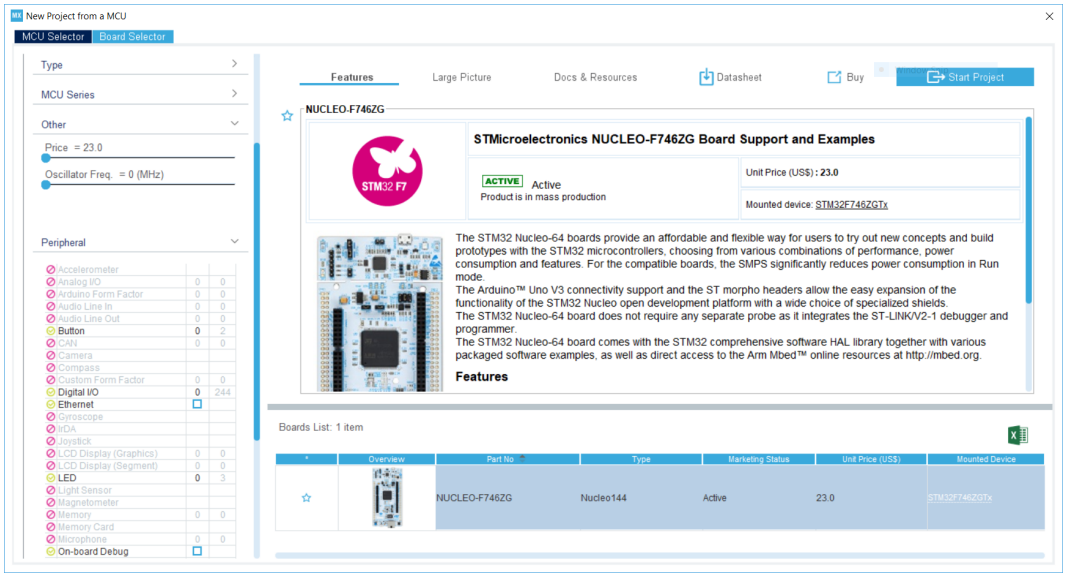
**Figure 11. AI filter with compression x4**




*Note: During the generation of the NN library, the size of the memory is also checked by the optimizer to notify the user if the minimal RAM and Flash size constraints are not respected according the selected MCU.*

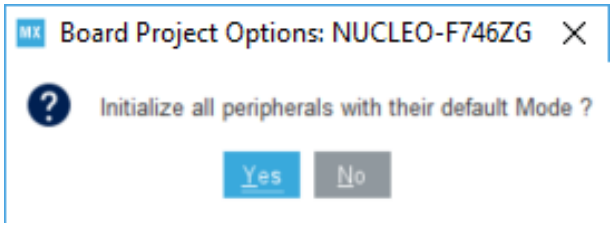
To continue, a NUCLEO-F746ZG development kit is selected as shown in Figure 12.

**Figure 12. NUCLEO-F746ZG board selection**




Click on the [**Start Project**] button to continue and confirm that all peripherals must be initialized with their default modes.

**Figure 13. Initialize all peripherals**

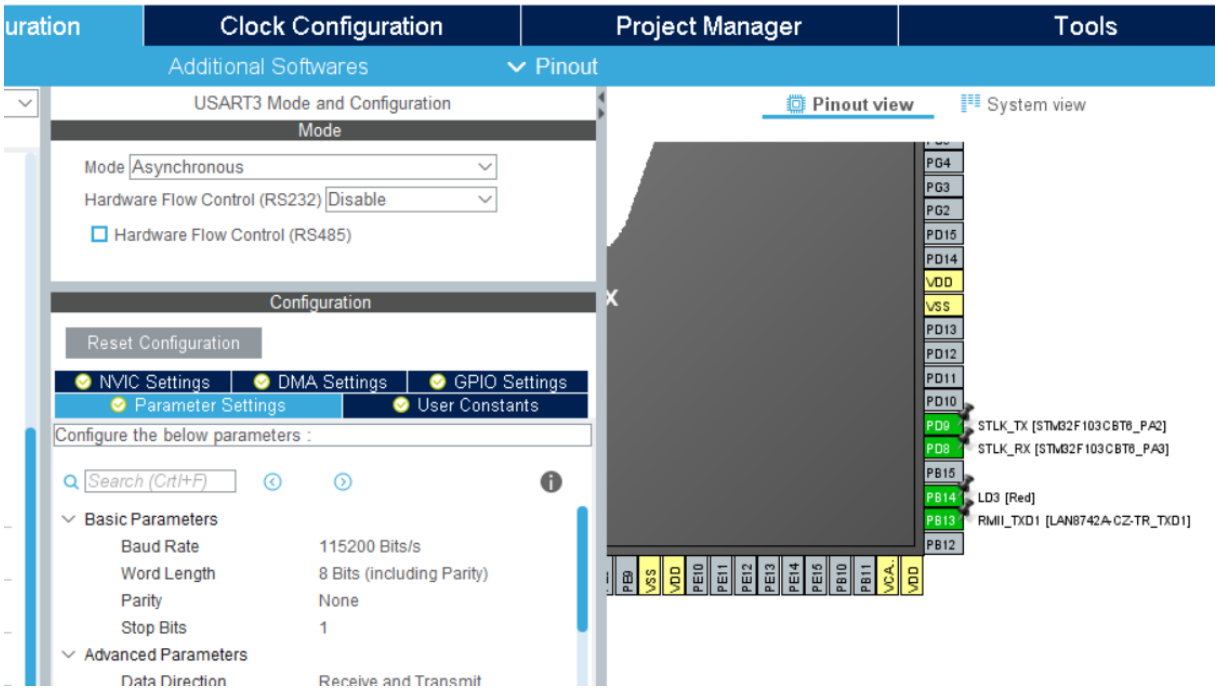



## 3.2 Hardware and software platform settings

Once an MCU or a board is selected, the related STM32 pinout is displayed. From this window, the user can set up the project by adding one or more additional software and peripherals, and configuring the clock.

If an add-on AI application (refer to Section 4.1 Adding the X-CUBE-AI component) is used, a USART-based link with the host development system is expected. For the STM32 Nucleo-144 development board, pins PD9 TX and PD8 RX are connected to the ST-LINK peripheral to support the Virtual COM port (refer to Figure 14).

**Figure 14. USART3 configuration**




For the NUCLEO-F746ZG, additional configurations of the clocks and memory sub-system are also expected to reach high-performance profile.

### 3.2.1 Increase or set the CPU and system clock frequency

1. Click the *Clock Configuration* tab.
   By default, in this lab, the system clock (SYSCLK, HCLK) is 72 MHz.
2. Type 216 in the HCLK (MHz) input blue box (refer to Figure 16) to call the clock wizard to configure automatically the PLL peripheral (and associated clock tree). If the the clock wizard pop-up appears as shown in Figure 15, click on the [**OK**] button to continue.

**Figure 15. Clock wizard pop-up**

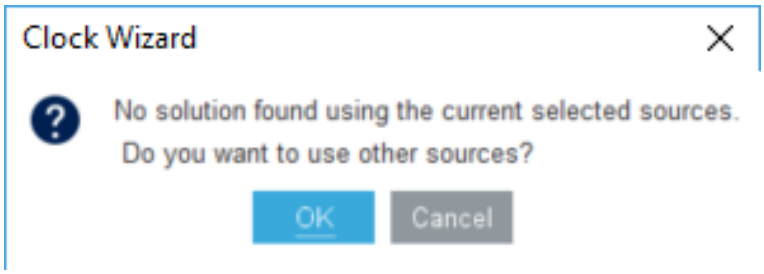
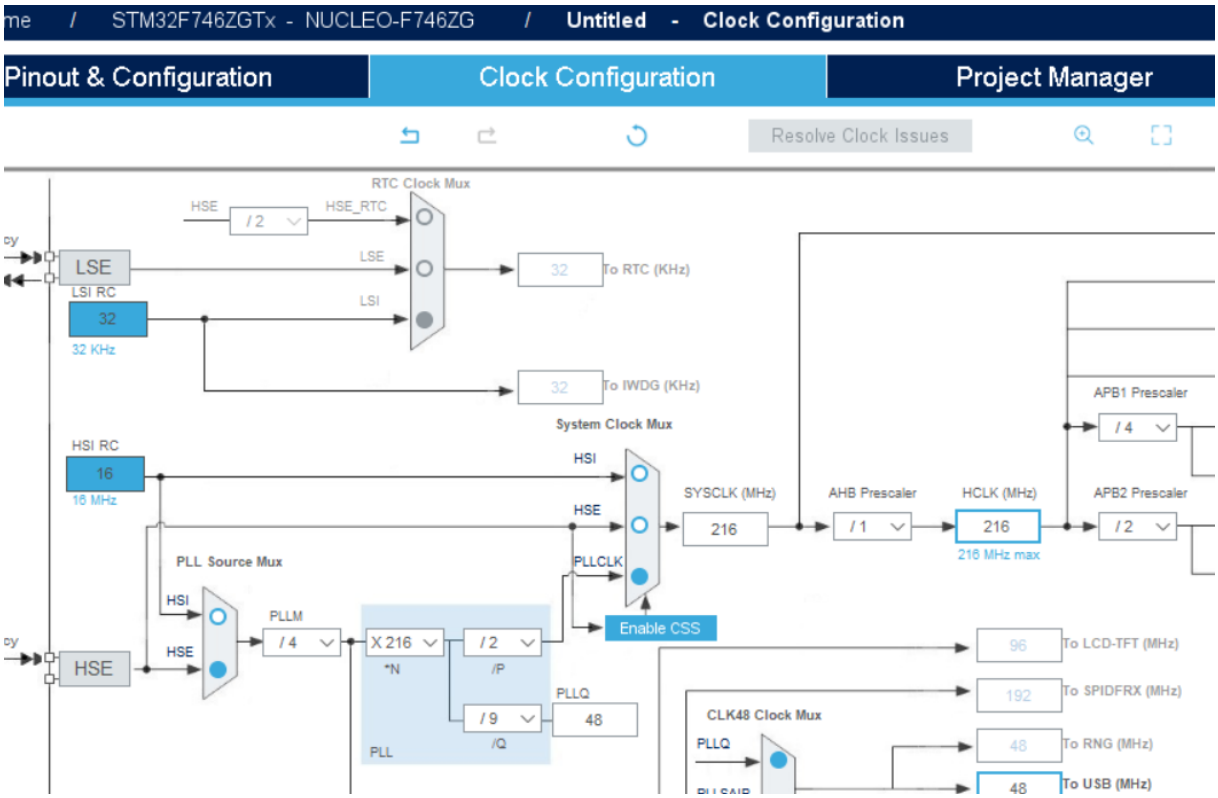


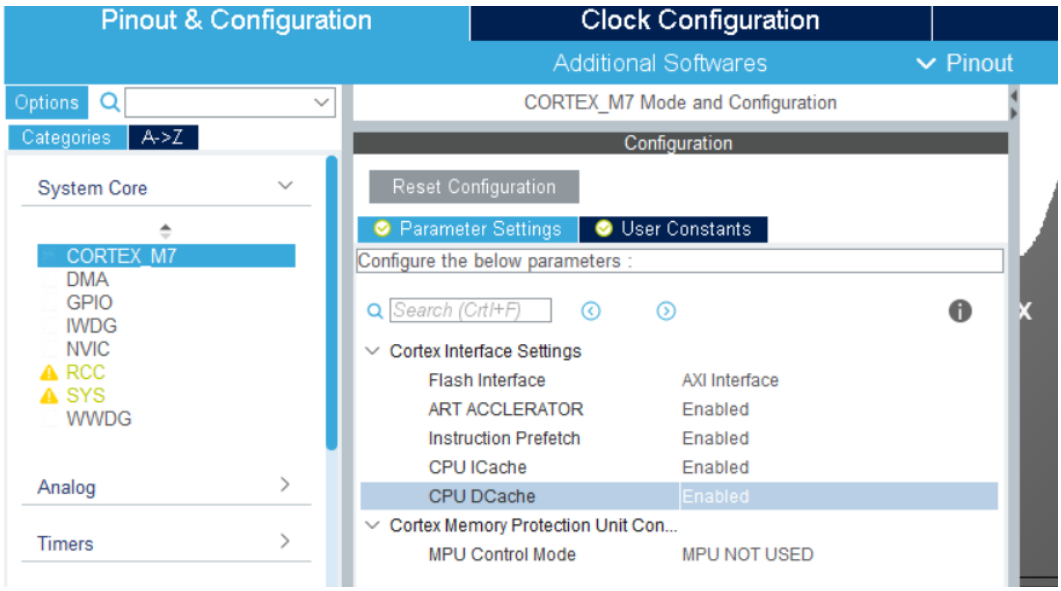
Figure 16. **System clock settings**




### 3.2.2 Set the MCU memory sub-system

- From the *Pinout & Configuration* tab (refer to Figure 17), click on the [**System Core**]>[**CORTEX_M7**] entry to open the Cortex®-M7 configuration wizard.
  The core instruction and data caches and ART accelerator sub-system must be enabled.

Figure 17. **MCU memory sub-system (parameter settings)**



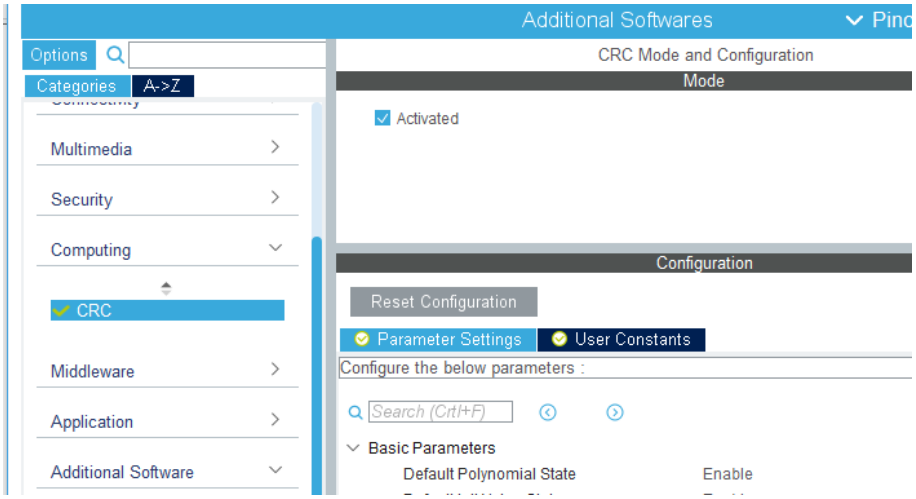
*Note: Setting the maximum value for the MCU clock is not mandatory. It must be aligned with the configuration that is used in the final design. The setting of the wait-state for the Flash is automatically adjusted by the STM32CubeMX platform code generator.*

### 3.2.3 CRC

The CRC peripheral is requested to support the NN library run-time protected mechanism. It must be enabled.

*Note: This is done automatically by the tool, so that it is not required to do it manually.*

**Figure 18. Enabling the CRC peripheral**

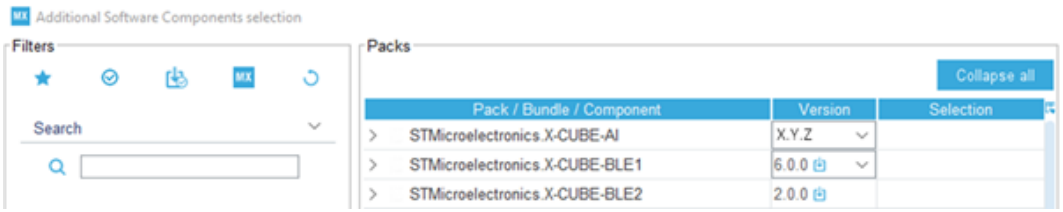


# 4 X-CUBE-AI configuration wizard

## 4.1 Adding the X-CUBE-AI component

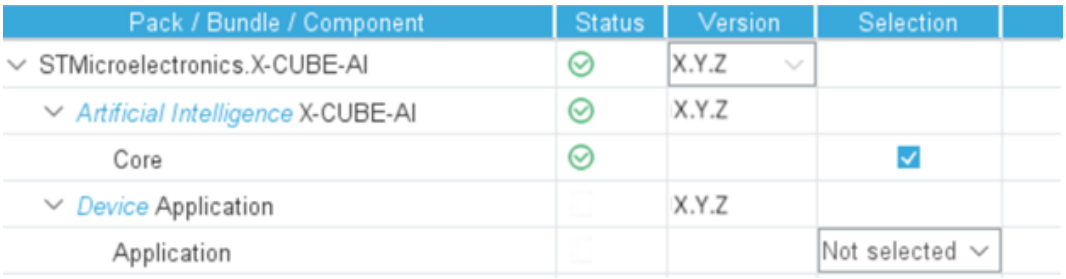
1. Click on the [**Additional Softwares**] button to add the X-CUBE-AI additional software to the project (refer to Figure 19).

**Figure 19. Additional software button**



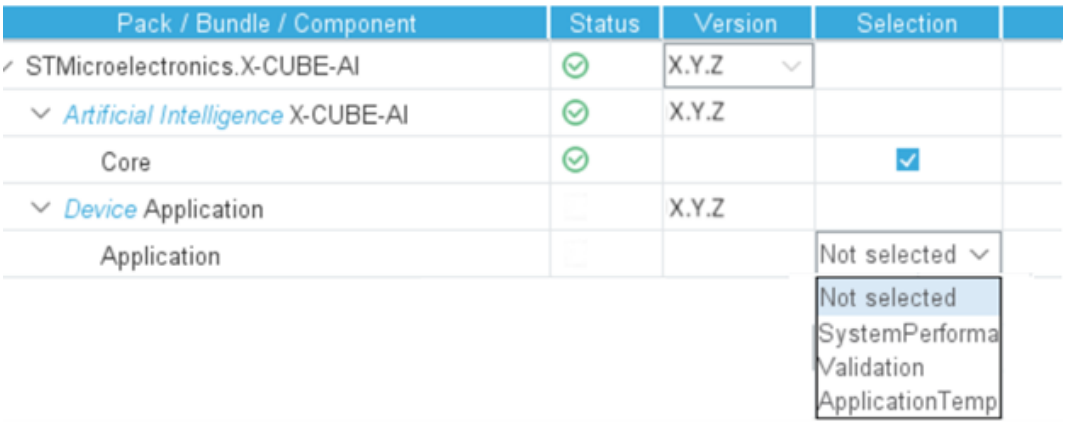

2. From the *Additional Software Component Selection* window, the X-CUBE-AI/core bundle (refer to Figure 20) must be checked to be able to upload the NN models and generate the associated STM32 NN library, In this case, as the library is fully integrated as a static library, the user only needs to implement his AI-based application/middleware on top of the generated well-defined NN API [10].

**Figure 20. Adding the X-CUBE-AI core component**



3. Optionally, one of the add-on X-CUBE-AI applications (refer to Figure 21) from the X-CUBE-AI/Application bundle can be selected.
   – System Performance: standalone AI test application for performance purpose
   – Validation: AI test application for validation purpose
   – Template application: basic application template for AI application

**Figure 21. Add-on X-CUBE-AI applications**




4. Click on [**OK**] to finalize the selection

## 4.2 Enabling the X-CUBE-AI component

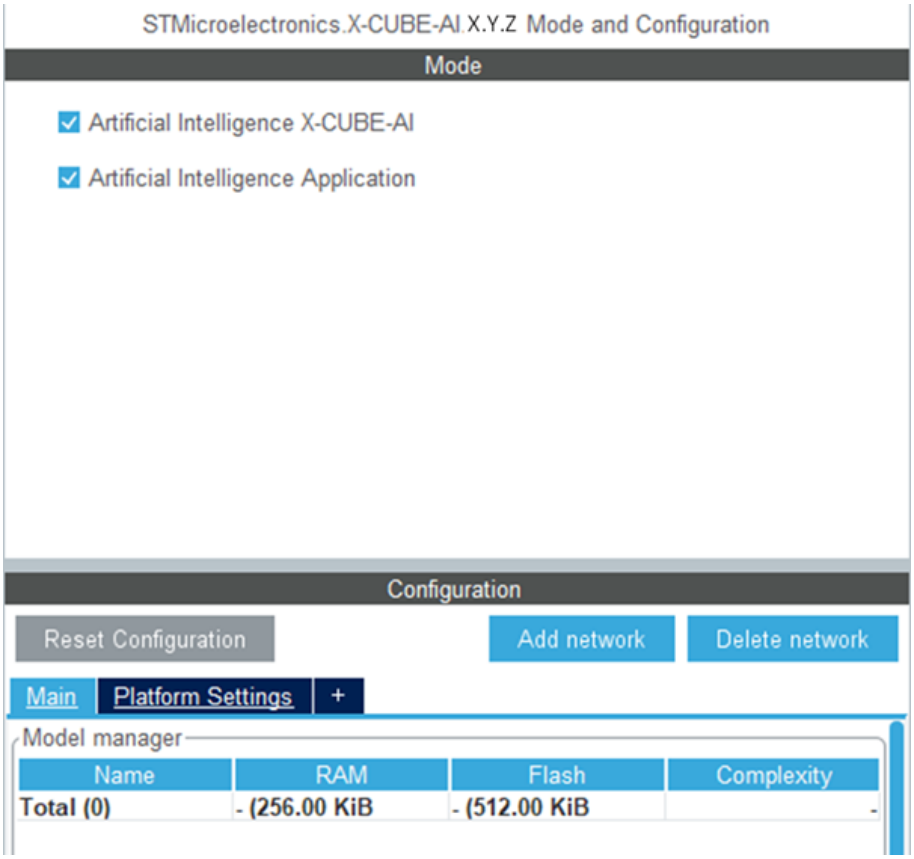
To enable and to configure the X-CUBE-AI component, the following additional steps are requested:

1. From the *Pinout & Configuration* tab, click on the [**Additional Software**] selector to discover the additional pieces of software. Click on [**STMicroelectronics X-CUBE-AI X.Y.Z**] to open the initial AI configuration window.

2. Check [**Artificial Intelligence Core**] to enable the X-CUBE-AI core component. [**Artificial Intelligence Application**] must be also checked to add the add-on AI application.

   The AI application that is selected here corresponds to the application enabled during the previous step (refer to Figure 21).

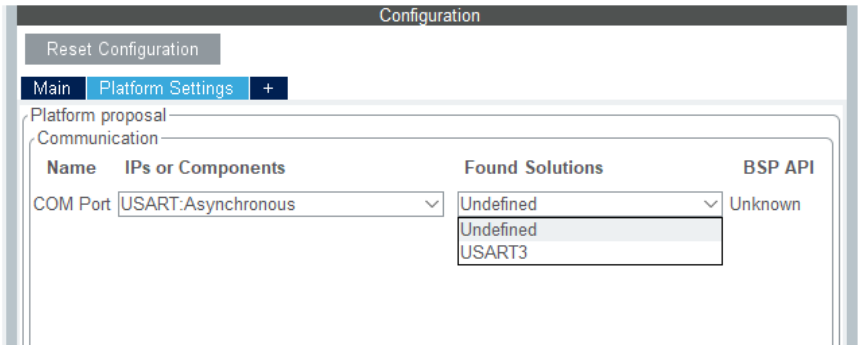
**Figure 22. Main X-CUBE-AI configuration panel**




– The *Main* tab provides an overview and the entry points to add or remove a network (respectively [**Add model**] and [**Delete model**] buttons). [**+**] can be also directly used to add a network.

– The *Platform Settings* tab indicates the handle of the USART peripheral used to report the information (AI System Performance application) or communicate with the host (AI validation application).

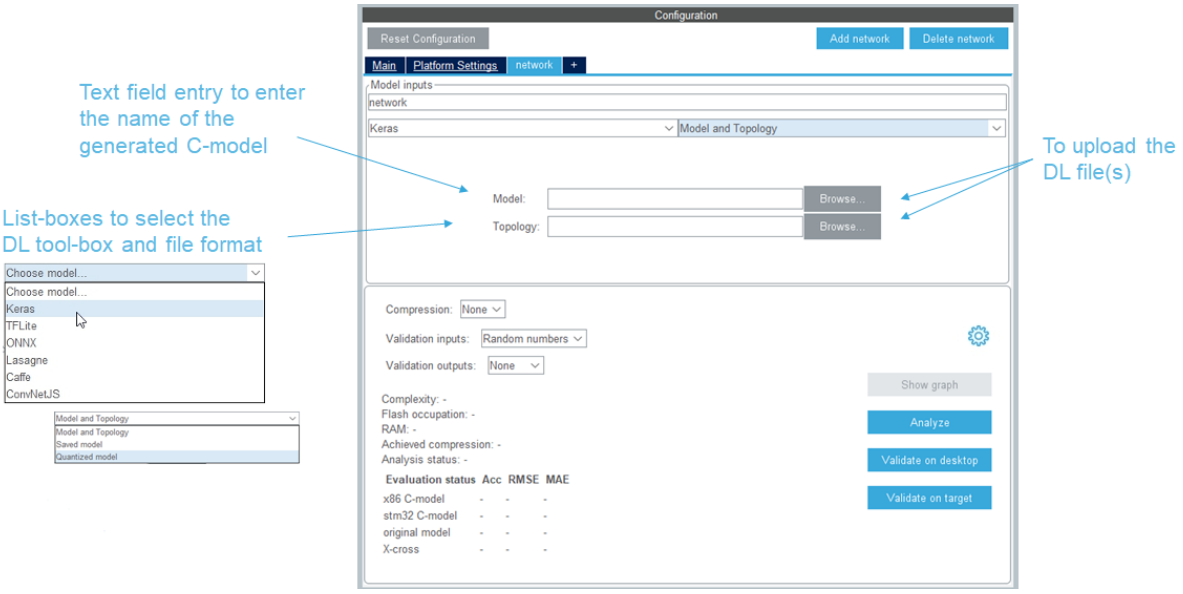
**Figure 23. X-CUBE-AI platform setting panel**



## 4.3 Uploading a pre-trained DL model file

From the *Main* tab, click on [**Add model**] or directly on [**+**] to open a new dedicated <*model_name*> configuration wizard. Alternatively, if the model was previously provided through the MCU filter, click directly on the *network* tab to open the NN *Configuration* pane.

**Figure 24. NN configuration wizard**




1. The text field entry is used to define the C name of the network (maximum 32 letters). This string is used directly to generate the name of the embedded client inference API (refer to [10]). If only one network is expected, the default network string name can be maintained.
2. The list box entries specify the DL toolbox used to export the DL model file and the associated file format(s) (refer to Section 12 Supported toolboxes and layers for Deep Learning for details).
   – Click on the [**Browse..**] button to upload the DL file(s) from the host file system. For this hands-on lab, a public Keras HAR model file is uploaded (saved model format).
3. Click on the [**Analyze..**] button to trigger a pre-analyze of the network reporting the dimensioning information (system integration point of view). Note that the compression factor was set before to 4, else a warning message pop-up is displayed as shown in Figure 25. If the `Invalid network` message box pops up, select [**Window**]>[**Outputs**] for more details in the log console (refer to Section 13 Error handling). Minimum RAM, Flash occupation and original DL model complexity are updated (refer to Section 4.5 ).

**Figure 25. Insufficient RAM/Flash message box**

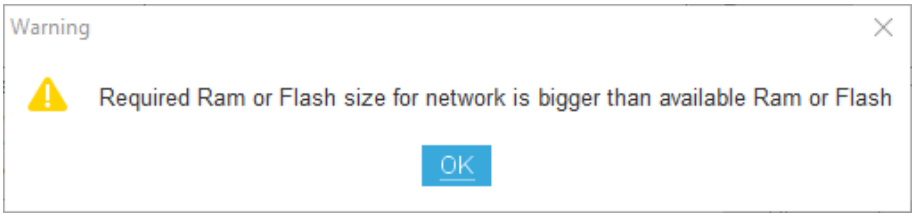
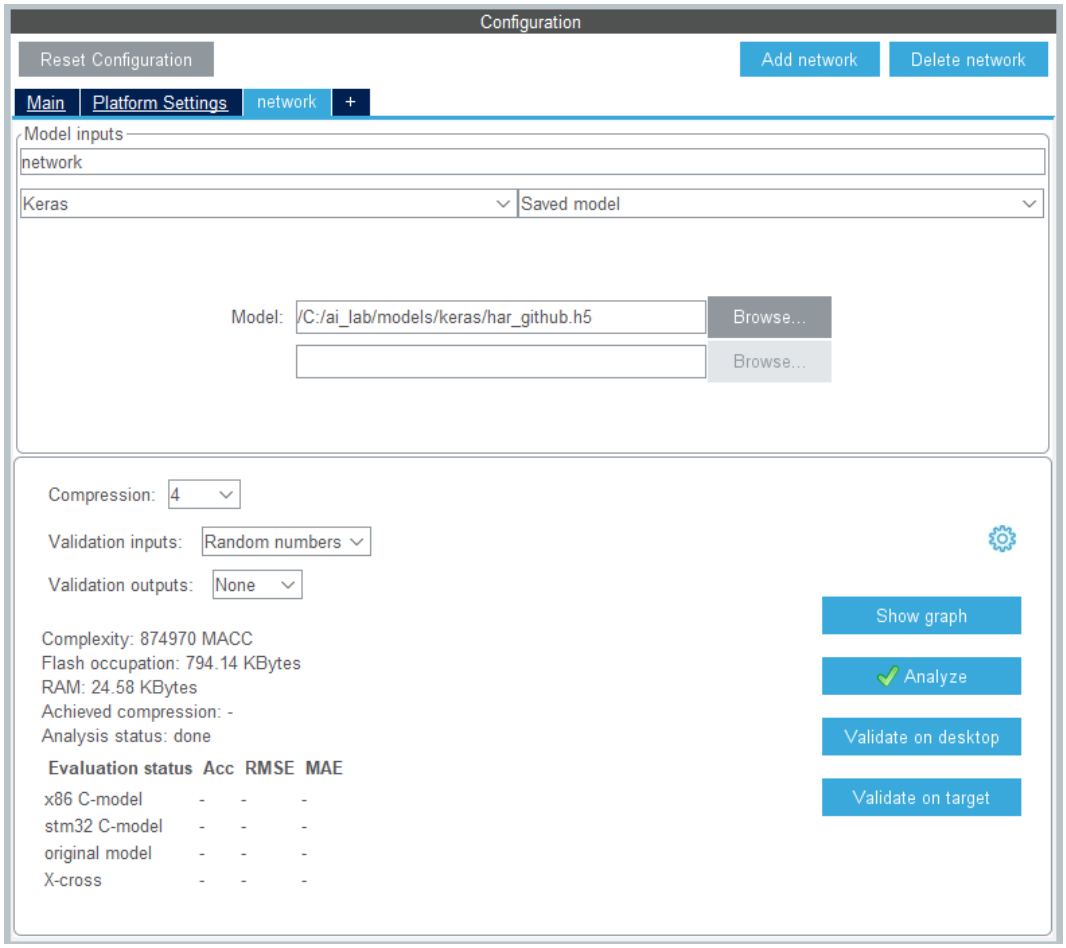


Figure 26. **Uploaded and analyzed DL model**




*Note: Additional debug/log information can be found in file* `C:\Users\<username>\.stm32cubemx\STM32CubeMX.log` *or* `$HOME/.stm32cubemx/STM32CubeMX.log`.

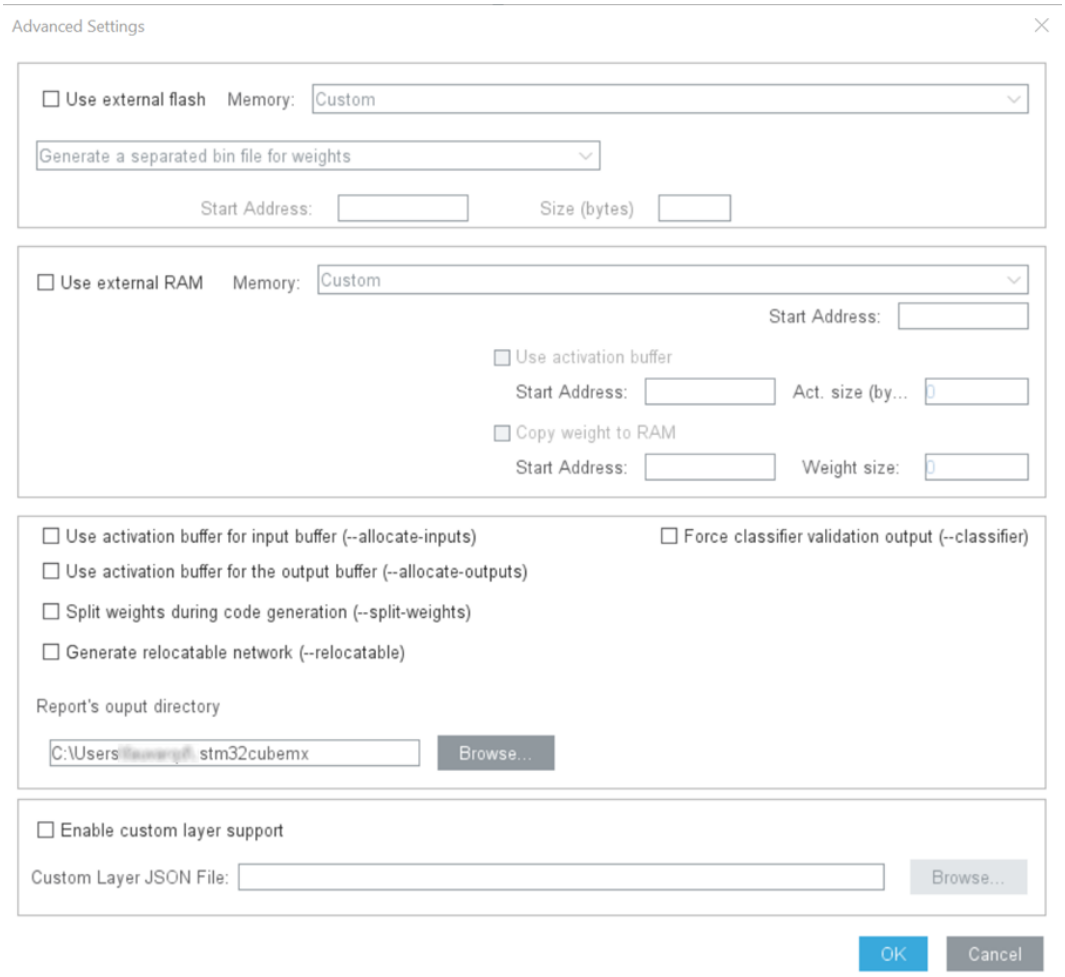
By clicking on the [**Advanced Settings**] button ( ), it is possible to configure the network to use external Flash memory for the weights or external RAM for weights or activations, to select extended options, or to configure custom layers. Refer to Advanced settings for details.

If the project is started from an STM32 STMicroelectronics board with mounted external Flash memory or RAM, the configuration of the external Flash memory or RAM is automatic during code generation. It uses the board BSP provided in the STM32Cube MCU Package to initialize the Flash memory or the RAM correctly. The external Flash memory is used in the memory-mapped mode.

## 4.4 Advanced settings

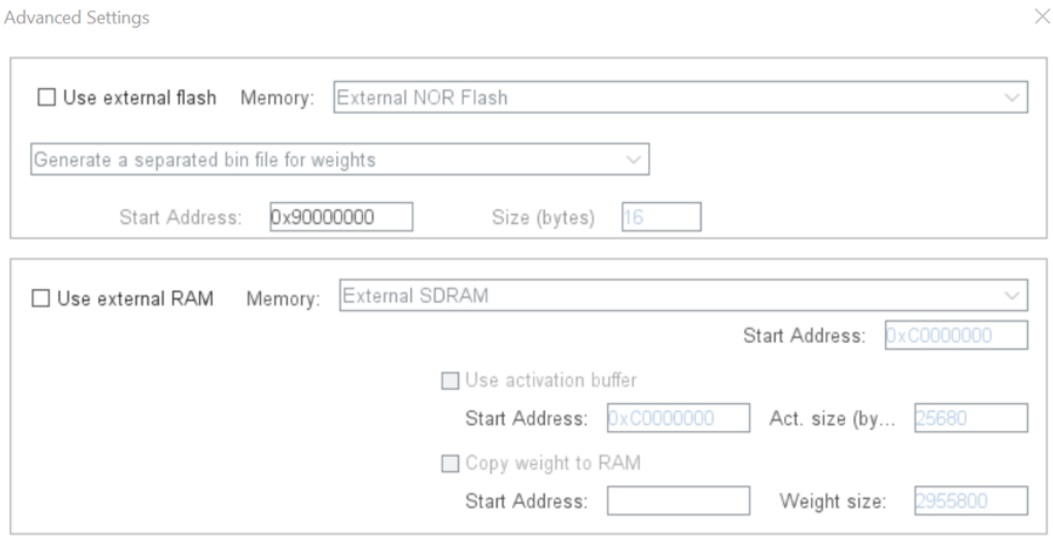
Figure 27 shows the content of the *Advanced Settings* window.

**Figure 27. Advanced settings**



### Use of external Flash memory or RAM

**Figure 28. Setting for external memories**



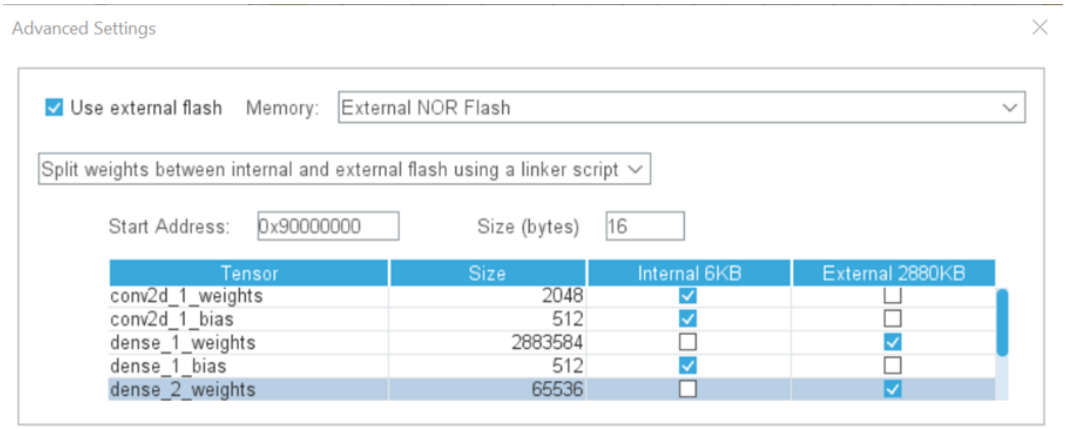

When [**Use external flash**] is selected, it is possible to:

- • Generate the weights in a separate `network_data.bin` file and the code is generated to point at the address of the beginning of the external Flash memory.

  The `network_data.bin` file must be programmed on the external Flash memory on the board manually using a tool such as STM32CubeProgrammer (STM32CubeProg).

  *Note: Using automatic validation on the target causes file `network_data.bin` to be automatically programmed on the external Flash memory on the board.*
- • Split the weights between internal and external Flash memory using a linker script.

**Figure 29. Weight memory split**

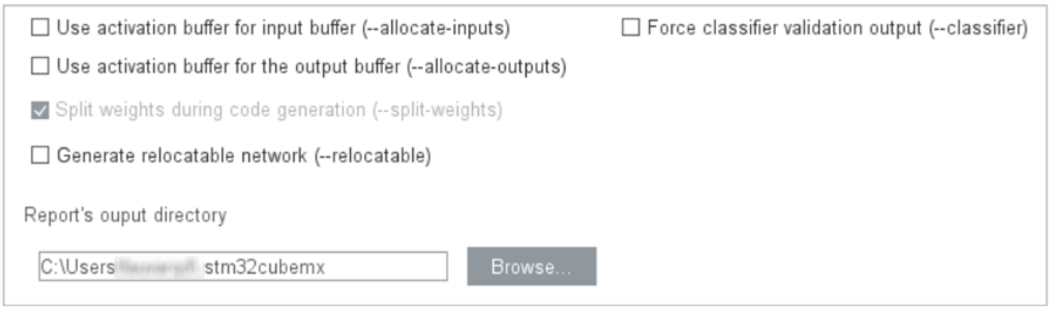



In the table of tensor, choose to place the tensor in internal or external memory.

When using an external Flash memory or RAM on a Cortex®-M7-based STM32 microcontroller, the ICache and DCache of the CORTEX_M7 CPU are enabled automatically, and the memory protection unit is configured to give access to the external RAM or Flash memory automatically.

### Extended options

**Figure 30. Extended options**




Selecting the [**Use activation buffer**] checkbox places the activation buffers in the external RAM at the address specified in the [**Start Address**] field

Optionally, it is possible to copy the weights to the external RAM at startup. In this case, the address where to copy the weights is requested.

When selecting the [**Use activation buffer for input buffer**] checkbox, the user does not need to allocate a specific input buffer and can put the input data in the pre-allocated space of the activation buffer. Depending on the size of the input data, the activation buffer may be larger, but overall less than the sum of the activation buffer plus the input buffer separately.

When selecting the [**Use activation buffer for the output buffer**] checkbox, the user does not need to allocate a specific output buffer. It is allocated automatically in the activation buffer, saving the overall used memory.

Selecting the [**Generate relocatable network**] checkbox activates the generation of a separated binary for the network and the embedded Neural Network library. This binary can be placed anywhere on the target, and the address is passed to the initialization function. This allows the update of the full network, weights, and topology, without having to reprogram the entire application. As the embedded Neural Network library is linked with the network, only the used kernels are in the final binary. This option can only be used in the case of a single network.

Selecting [**Split weights during code generation**] creates one array of weights per layer in the generated `<network name>_data.c` file. Each array can then be placed in different memory sections if needed using a dedicated linker script.

### Custom layers

**Figure 31. Custom layers**


Custom layers are a way to extend the current X-CUBE-AI capabilities. Specify the json file for configuring the custom layer in that panel. Refer to the internal documentation on custom layer for implementation details

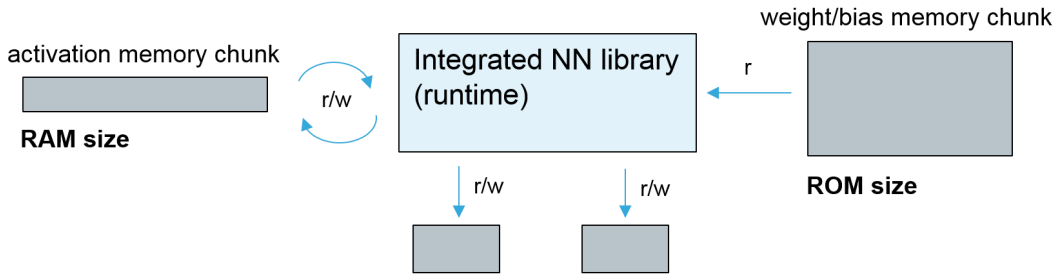
## 4.5 Dimensioning information report

When a DL model is processed, the dimensioning system informations presented in Table 2 and Figure 32 are reported.

**Table 2. System informations reporting**

| Reported information | Description |
|---|---|
| RAM | Indicates the size (in bytes) of the expected RW memory chunk used to store the intermediate inference computing values (.data or .bss section). |
| ROM/Flash | Indicates the size (in bytes) of generated RO memory chunk to store the weight/bias parameters after compression if requested (.rodata section). |
| Complexity | Indicates the functional complexity of the imported DL model in Multiply And Accumulate operations (MACC). It includes also an approximation of the activation functions (expressed with the same unity). |

**Figure 32. Integrated C-model (runtime-view)**




*Note: The minimum RAM and Flash size requirements listed in the AI summary do not take into consideration the memory constraints of the user application (including the RAM to store the input and output tensors). Only the DL model weights/bias and activation memory requirements are considered here. NN kernel functions and specialized model code, including the minimum stack/heap size, are not considered also.*

### 4.5.1 CPU cycles/MACC?

No theoretical relation is defined between the reported complexity and the real performance of the generated NN C library (CPU cycles / MACC). Due to the variability of the targeted environments (including Arm® tool-chain, MCU and underlying sub-system memory setting, NN topology and layers, and optimizations applied), it is difficult to estimate off-line an accurate CPU cycles/MACC vs. STM32 system settings. However, out-of-the-box, the following rough estimations can be used (for a 32-bit floating-point C model):

- STM32 Arm® Cortex®-M4/M33: ~9 cycles/MACC
- STM32 Arm® Cortex®-M7:- ~6 cycles/MACC

The add-on *"AI System Performance"* test application has been specifically designed to report the factual on-device performance (refer to Section 9 AI system performance application for details).

### 4.5.2 Generated C-model graph representation

Click on the [**Show graph**] button to show the main structural information of the uploaded DL model that are considered by the C-code generator. Three graphs are available:

1. The internal representation of the imported DL model before applying the optimizations as shown in Figure 33
2. The representation of the generated C code after all optimizations as shown in Figure 34
3. Click on a layer to get additional information on the layer as shown in Figure 35

**Figure 33. Network before optimizations**

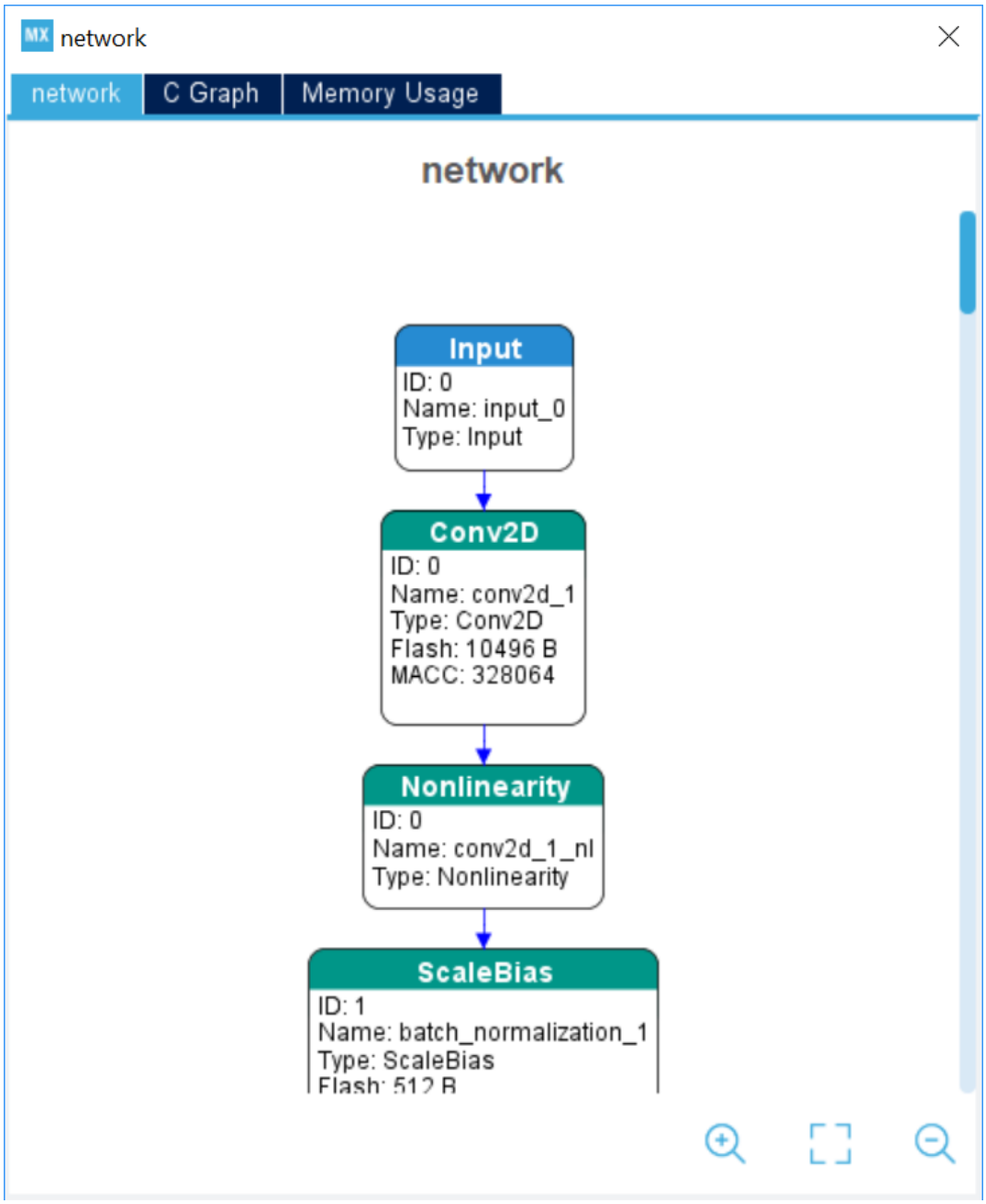
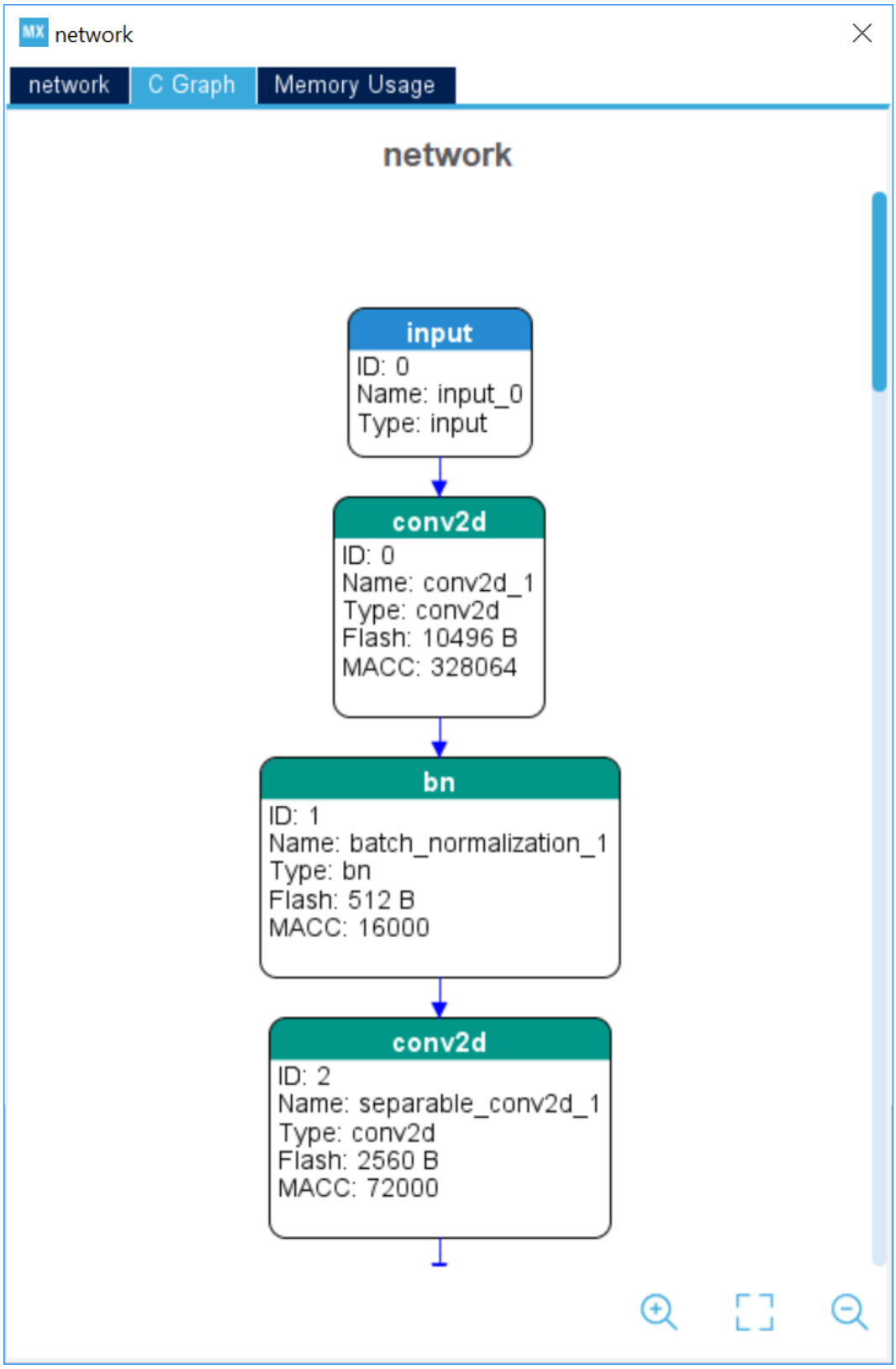
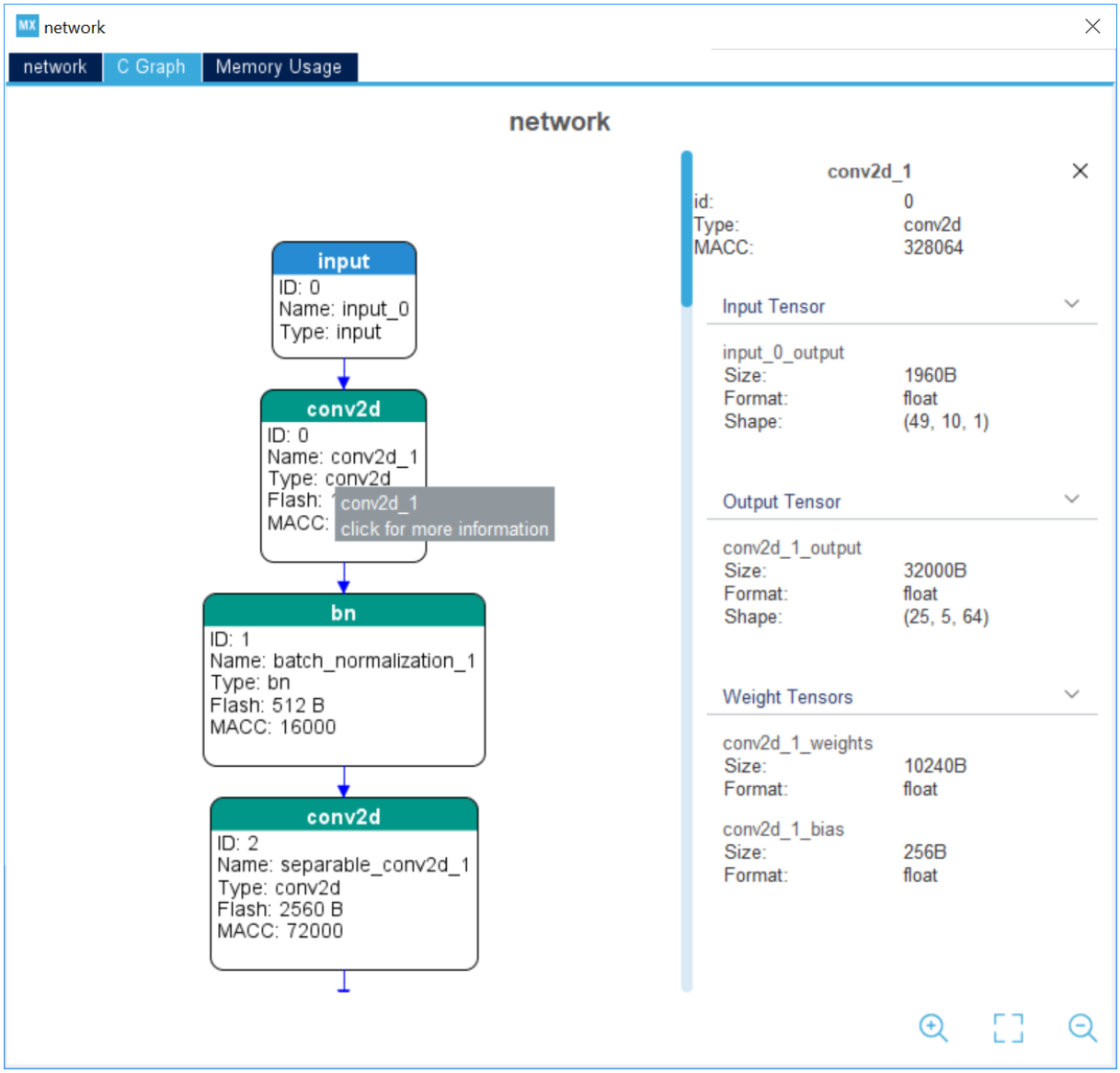


Figure 34. **C graph of the generated code**



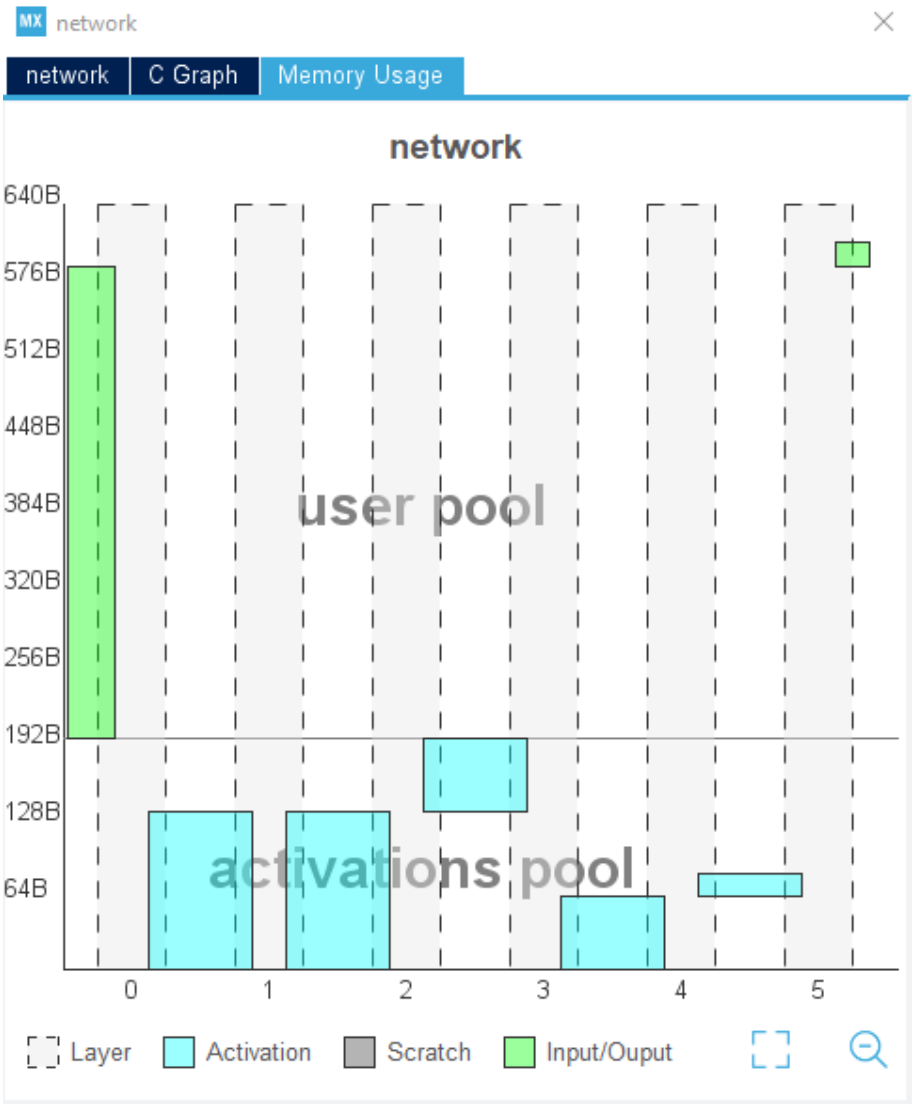
Clicking on a layer gives more information on the layer.

**Figure 35. Layer information**



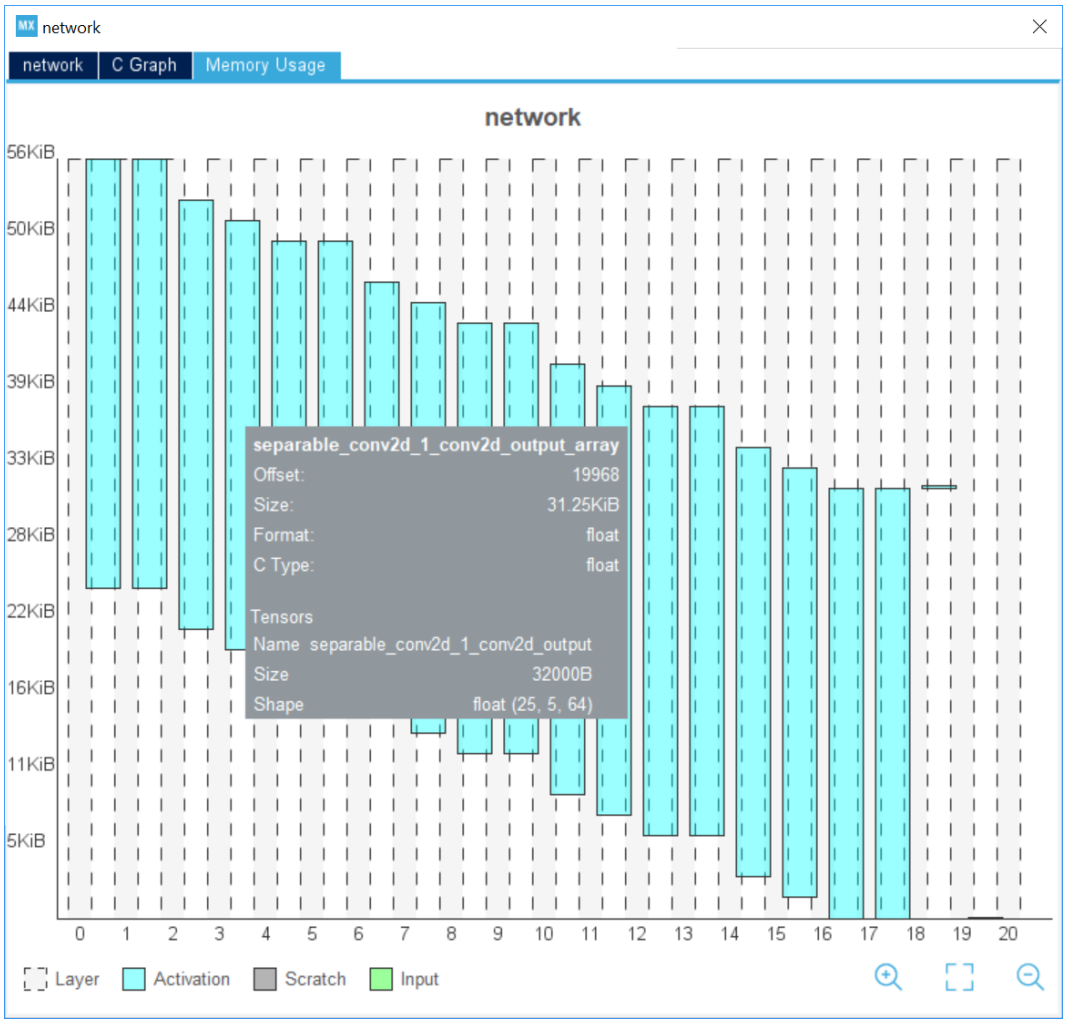
The memory usage for activation and internal buffers is shown in Figure 36. This view shows the total amount of memory needed to run the network. When the input or output buffers are put in the activation buffer, they do not appear anymore in the user pool.

**Figure 36. Memory usage**



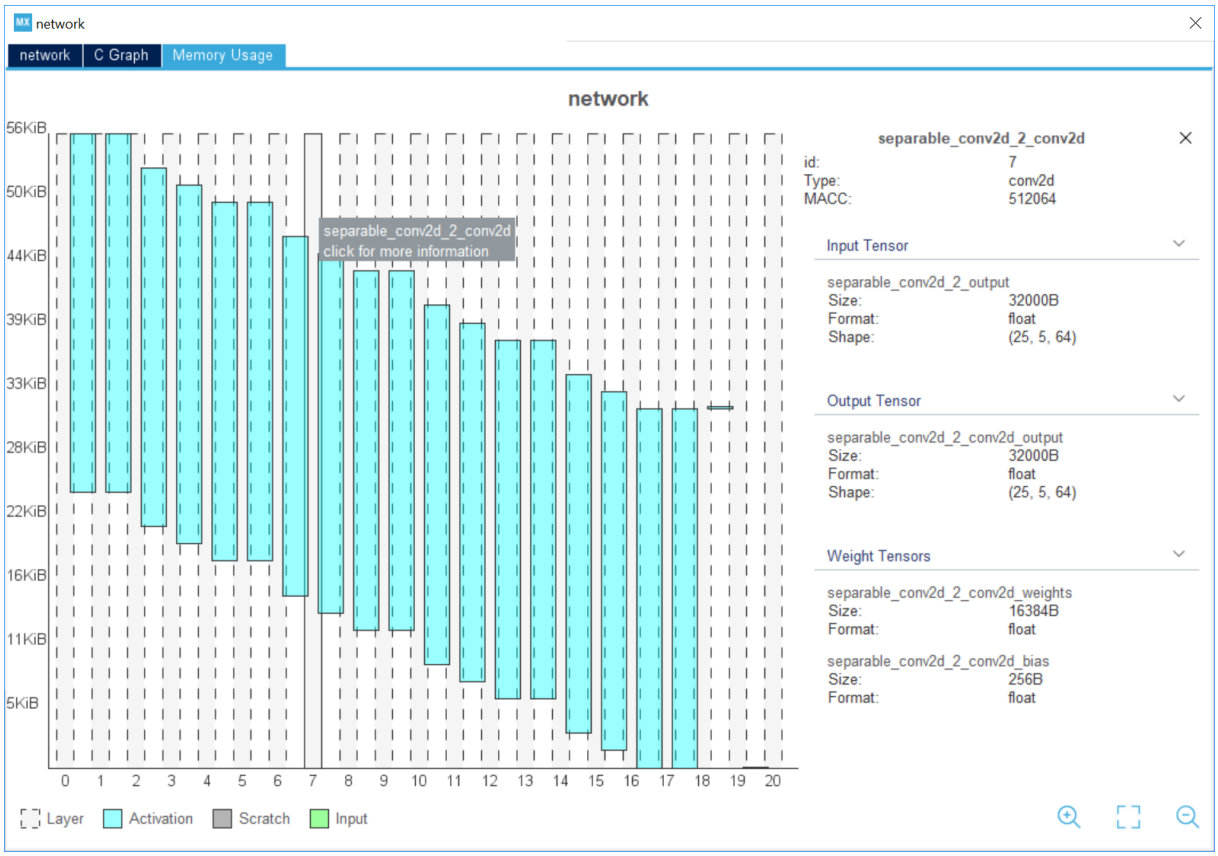
In the memory graph, a tooltip on a buffer shows the buffer details.

**Figure 37. Buffer details as tooltip**



In the same view, click on a layer to display its detailed information.

**Figure 38. Layer detailed information**



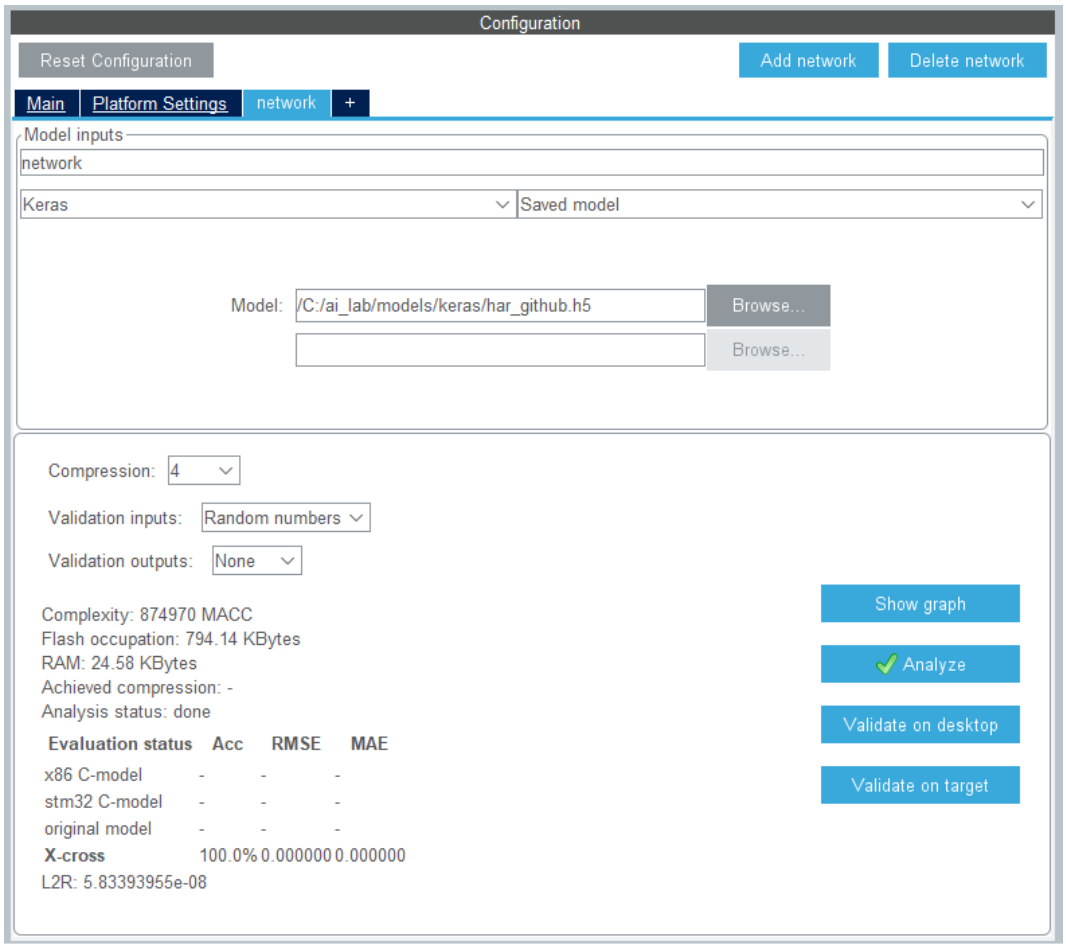

## 4.6 Validating the generated C model

Click on the [**Validate on desktop**] button to launch a validation process of the generated C model. According to the availability of the custom data, different metrics are computed (refer to Section 6.2 Validation engine). Note that this step is optional but preferable, in particular when a compression factor for instance is applied (refer to Section 6.1 Graph flow and memory layout optimizer). When the reference or ground-truth output values are provided with the associated input samples, the predicted values are used to calculate the metrics listed in Table 3 (refer to [11] for more details).

**Table 3. Metrics**

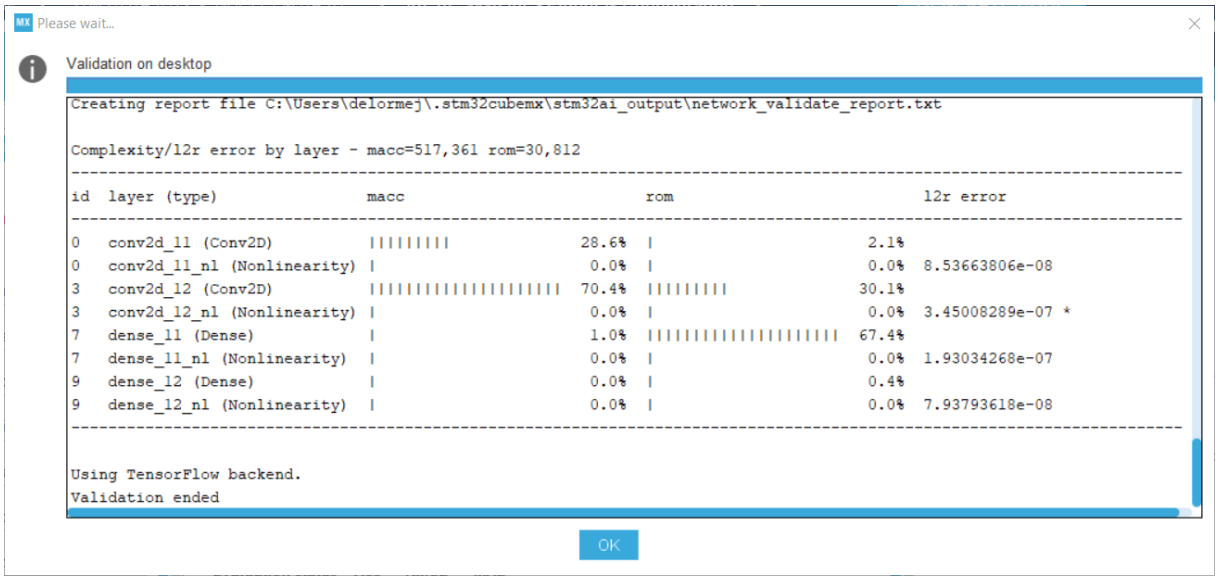
| Metric | Description |
|---|---|
| ACC | Classification accuracy |
| RMSE | Root mean square error (classification accuracy) |
| MAE | Mean absolute error |
| L2r | L2 relative error |

Figure 39. **Validation status field**




More detailed information is reported in the UI log console as shown in Figure 40. In particular the L2r error is also reported for each generated C layer matching with an original layer.

Figure 40. **Validate on desktop - log report**

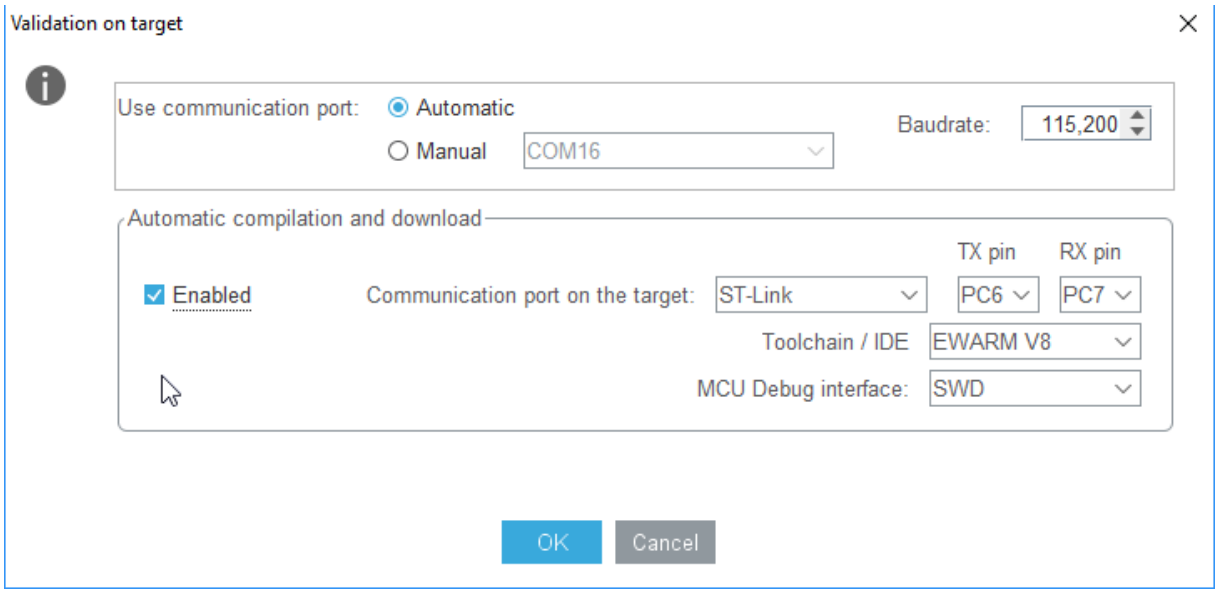


*Note: At this step, the uploaded DL model is ready to be integrated in the generated IDE project.*

The [**Validate on target**] option must be used only later when the targeted device is programmed with the special test application *"AI Validation"*. It must be selected during the previous step (step 3 of Section 4.1 Adding the X-CUBE-AI component). Reported information and usage are fully described in Section 10 AI validation application.

The [**Validate on target**] button allows the user to run the validation on the target and optionally automatically generate, compile, program and run a temporary project corresponding to the current network.

**Figure 41. Validation on target**




For the automatic compilation, programming and run to work, verify that the proposed communication port on the target corresponds to the USART (UART, LPUART) connected to the ST-LINK for the Virtual COM port. Optionally, it is possible to force the peripheral instance to use and the pins used for the transmit and receive signals.

By default, the proposed toolchain is selected for the project. It can be updated if needed.

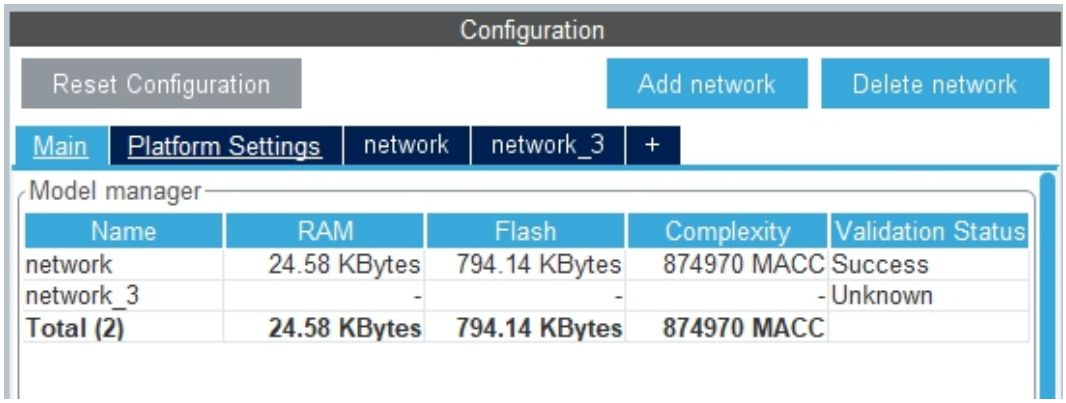
If a JTAG interface is used to program the MCU, the default debug interface must be changed.

When pressing [**OK**], a temporary project is generated, compiled, programmed and started on the target. Then the regular validation of the network takes place.

## 4.7 Adding a new DL model

Multiple DL models can be imported. The total number is not limited by the wizard, however the initial limitation is mostly related to the sizes of available RAM and Flash memory in the selected STM32 MCU device. Click on the [**+**] button to import a new DL model and apply the same previous steps. The *Main* view summarizes the total RAM and Flash memory occupations.

**Figure 42. Main view with multiple networks**



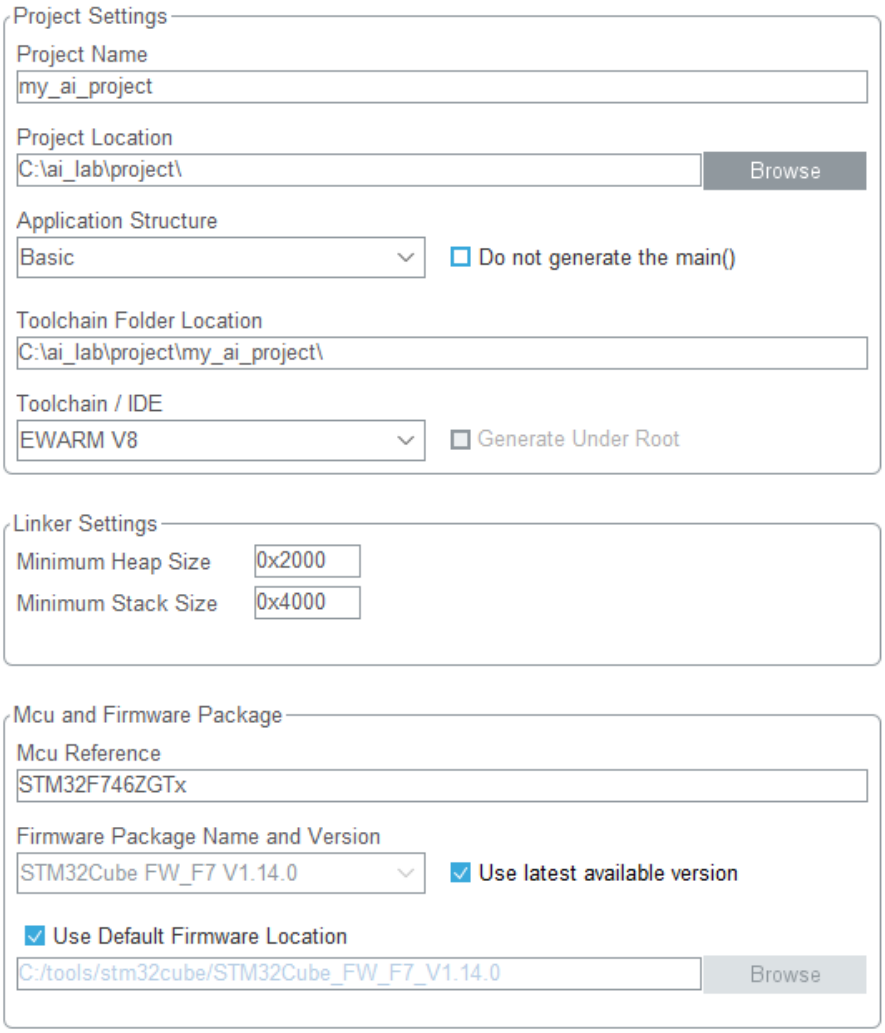
# 5 Generating, building and programming

## 5.1 Generating the IDE project

The following steps show in sequence the classical STM32CubeMX process to generate the IDE project without addition of any specific AI extension:

1. Click on the *Project Manager* view.
2. Set the project location and name.
3. Select one Toolchain and IDE (such as EWARM for IAR Systems IAR Embedded Workbench®, Keil® MDK-ARM, or STM32CubeIDE).
4. Verify that the heap/stack size is properly set to minimize in a first time the possible overflow (2 Kbytes minimum of heap is expected for the *"AI Validation"* test application).

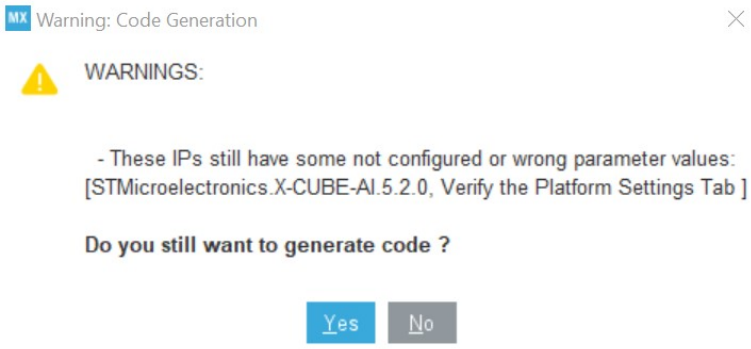
**Figure 43. Project settings view for IDE Code generator**



5. Click on the [**GENERATE CODE**] button to generate the code corresponding to the current project configuration (including the IDE project files).

   During the generation of the IDE project, the message box shown in Figure 44 can pop up if the user has selected and enabled an add-on AI application and forgotten to set the expected platform dependency (such as USART handle). Refer to Section 4.1 Adding the X-CUBE-AI component for details.

**Figure 44. AI peripheral not fully configured**




At this stage, the STM32CubeMX UI application can be closed. It is possible to re-open it later with the `<project_name>.ioc` file to enable and set a new peripheral, a middleware component, or both, or perform the *Validation on target* process.

## 5.2 Building and programming

When the IDE project is successfully generated by the STM32CubeMX tool, the standard build process is used to build and flash the STM32 board development kit or customer board:

1. Launch the IDE application and open the generated project file
2. Build and flash the firmware image. If the AI test application has been selected, no code modification or update is expected. Otherwise, user AI-based application code must be added to use the generated inference C API.

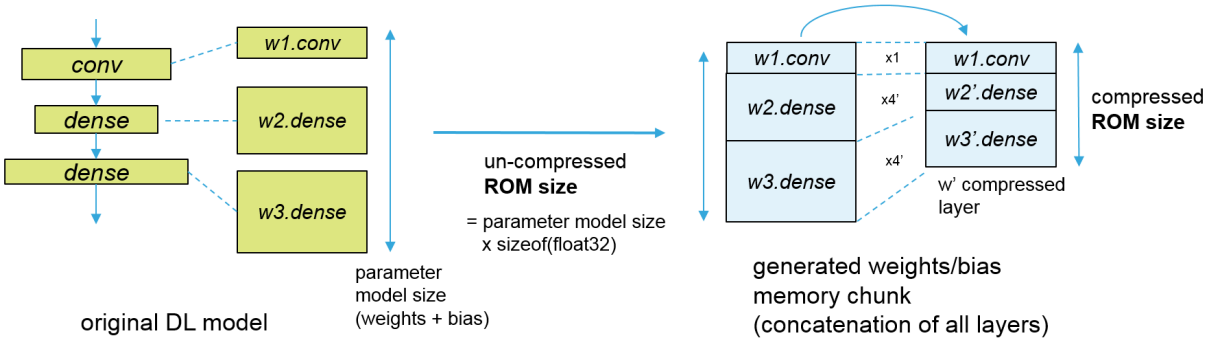
# 6 X-CUBE-AI internals

## 6.1 Graph flow and memory layout optimizer

The C-code generator optimizer engine seeks to optimize memory usage (RAM & ROM) against inference computing time (power consumption is also considered). It is based on a dataset-less approach, which means that no trained valid or test dataset is requested to apply the compression and optimization algorithms (no re-trained/refined weights/bias stage is expected to preserve the accuracy of the initial model).

- **Weight/bias compression** (targeted factor: none, x4, x8)
  - Only applicable for dense (or fully-connected) layer type
  - Weight-sharing-based algorithm is applied (K-means clustering)
  - If *"none"*, the initial DL model accuracy is guaranteed. The residual error (~$10^{-08}$) is related to the native-model floating-point 64-bit size against the 32-bit C-floating-point size used. For large networks however, $10^{-06}$ is more common.

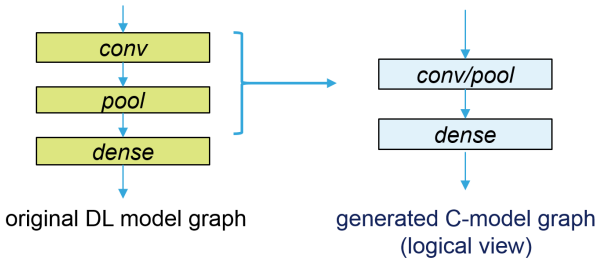
**Figure 45. Weight/bias compression**




The advantage of this approach is to have a quick compression process, but the final result is not lossless and the global accuracy can be impacted. A *"Validation"* process of the generated C model is provided as a mitigation to evaluate the generated error (refer to Section 6.2 ).

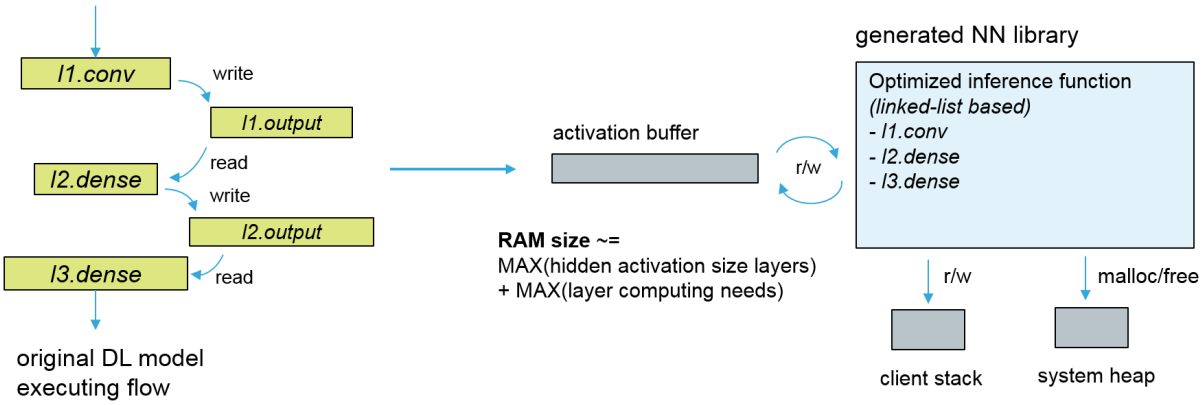
- **Operation fusing**
  - Merge two layers to optimize data placement and associated computing kernel. Some layers (like *"Dropout"*,*" Reshape"*) are removed during the conversion or optimization, and others (like nonlinearities and pooling after a convolutional layer) are fused in the previous layer. The effect is that the converted network has often a lower number of layers compared with the original network.

**Figure 46. Operation fusing**



- **Optimal activation/working memory**: A R/W chunk is defined to store temporary hidden layer values (outputs of the activation operators). It can be considered as a scratch buffer used by the inference function. The activation memory is reused across different layers. As a result, the activation buffer size is defined by the maximum memory requirements of two consecutive layers.

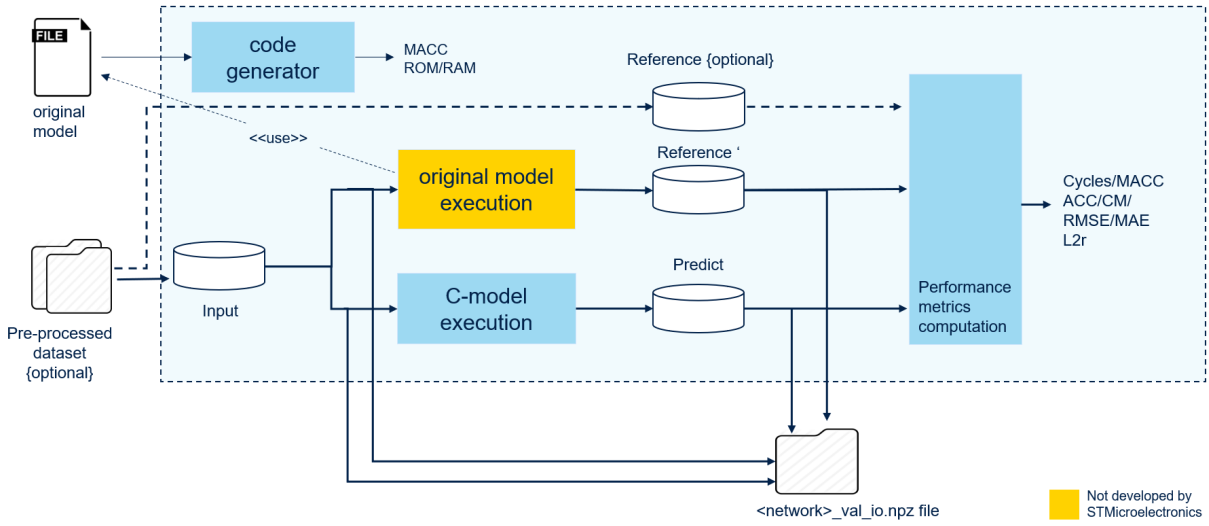
**Figure 47. Optimal activation/working buffer**




## 6.2 Validation engine

A simple and quick validation mechanism is provided to compare the accuracy of a generated model and uploaded DL model from a numerical standpoint. Both models are fed with the same input tensors (fixed random inputs or custom dataset; refer to [11]). To be more accurate, as detailed in [11], additional metrics are reported to evaluate the generated C model. The X-CUBE-AI Expansion Package provides an inference DL executing engine for all supported DL frameworks. Note that it is still possible to consider the generated optimized C model even if the tool reports a failed validation. The performance may not be aligned with the original Python™ model but the C-model can still be used. Further inspection is required, using, for example, a custom dataset and NN output tracing.

**Figure 48. Validation flow overview**



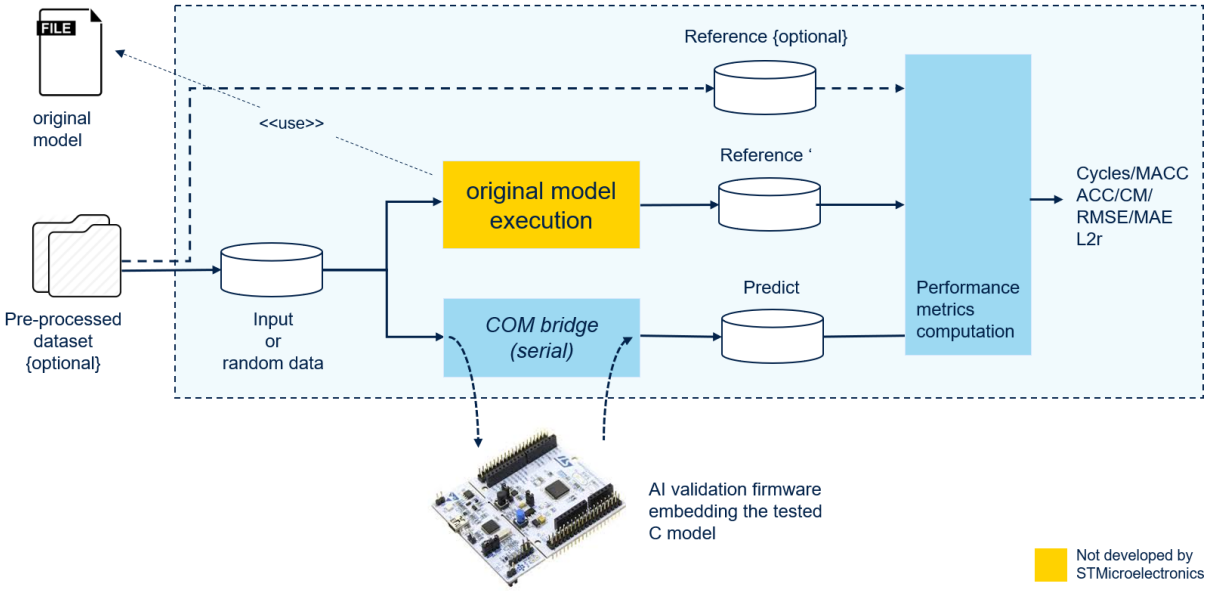
Two executing modes are provided:

- *Validation on desktop*: this mode allows the comparison of the DL model with its generated X86 C model. It runs on the host. The related output is illustrated in Section 4.6 Validating the generated C model.
- *Validation on target*: this mode compares the DL model with the C model that runs on the targeted device. It requires a special AI test application that embeds the generated NN libraries and the COM agent to communicate with the host system. Output and usage are illustrated in Section 10 AI validation application.

*Validation on target* features:

- Automatic detection of the connected STM32 boards
- The signature of the embedded generated C model is checked with the validated DL model
- The L2 error is only calculated on the last output layer (because of the COM speed to upload the data)
- Additional information is reported such as inference executing time by layer or others (refer to Section 10 AI validation application)

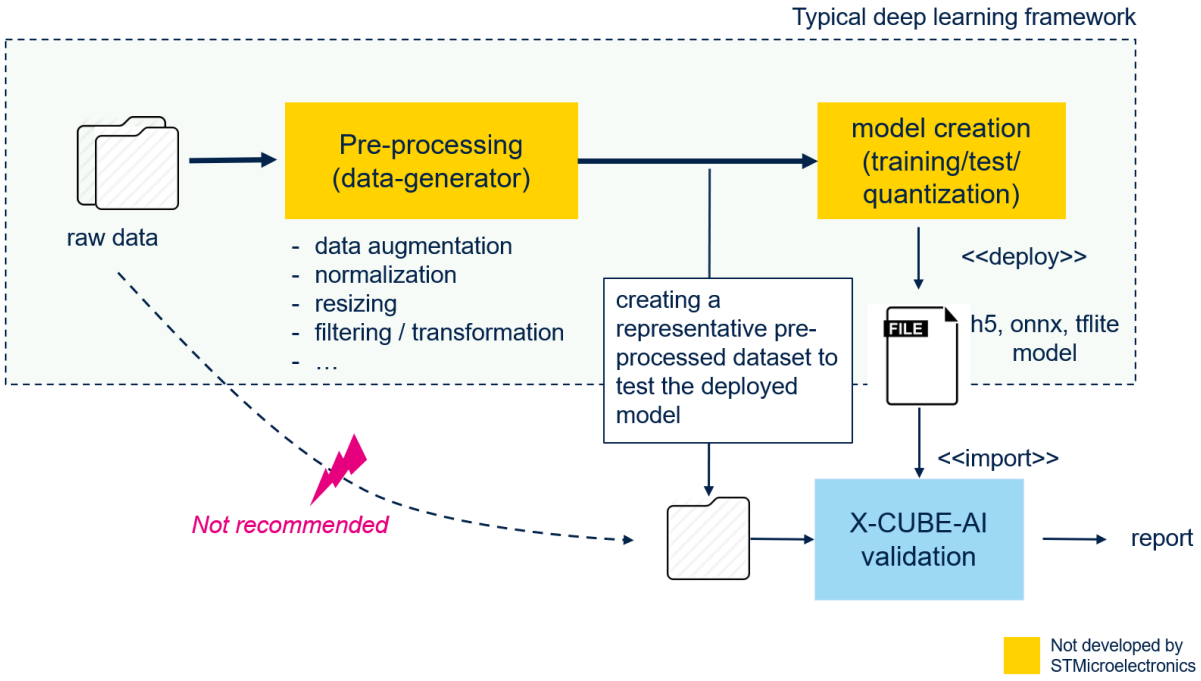
**Figure 49. Validation on target**



### Specific attention to custom data

To have an accurate validation process or more significant metrics, it is important to feed the original model and the generated C model with the closest possible data to the validating dataset used to to test the original model. If the raw dataset has been pre-processed, the user must build a representative dataset with a sub-set of these pre-processed data.

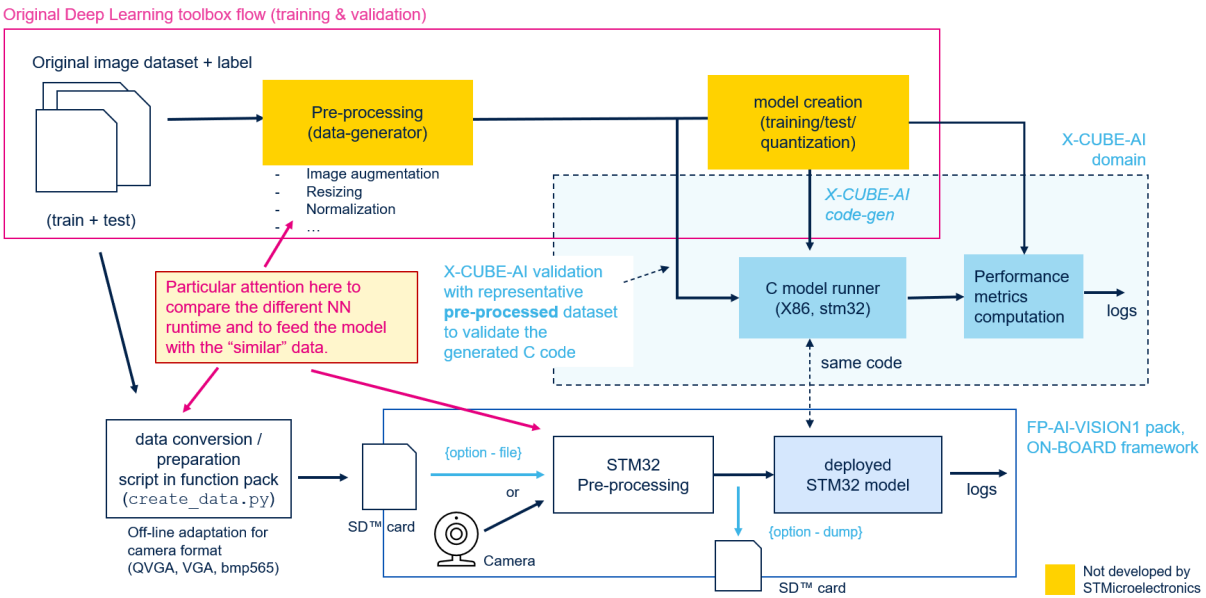
**Figure 50. Representative dataset**




This recommendation also applies to the output data. The computation of the metrics are based on the element-wise operations between the provided references and the predicted values. For a classifier (one or multiple classes) for example, one-hot encoding data can be provided. Integer Encoding is not supported.

Figure 51 illustrates a typical example, where a particular attention is requested. The STM32Cube function pack for computer vision (FP-AI-VISION1) provides an advanced debug mode, which allows the injection or dump of images. If the user wants to validate its pre-trained model with the X-CUBE-AI validation engine and compare with the deployed model, he must ensure that the format and the applied pre-processing are similar to have the more accurate validation metrics. In this use case, to take the pre-processing done by the pack into account, it is recommended to create a set of representative pre-processed images to test different models off-line, or to refine an existing pre-trained model.

**Figure 51. Example with the function pack for the computer vision**



# 7 Generated STM32 NN library

Only the specialized (DL model dependent) C files are generated for each imported DL model. The name of these files are prefixed with the network name provided by the user (refer to Section 4.3 Uploading a pre-trained DL model file). They are based on an internal and private API implemented by the *network_runtime.a* library:

- *<name>.c* and *<name>.h* files for the topology
- *<name>_data.c* and *<name>_data.h* files for the weights/bias

*Note: Generated specialized data and network files are common to all toolchains and STM32 MCU series.*

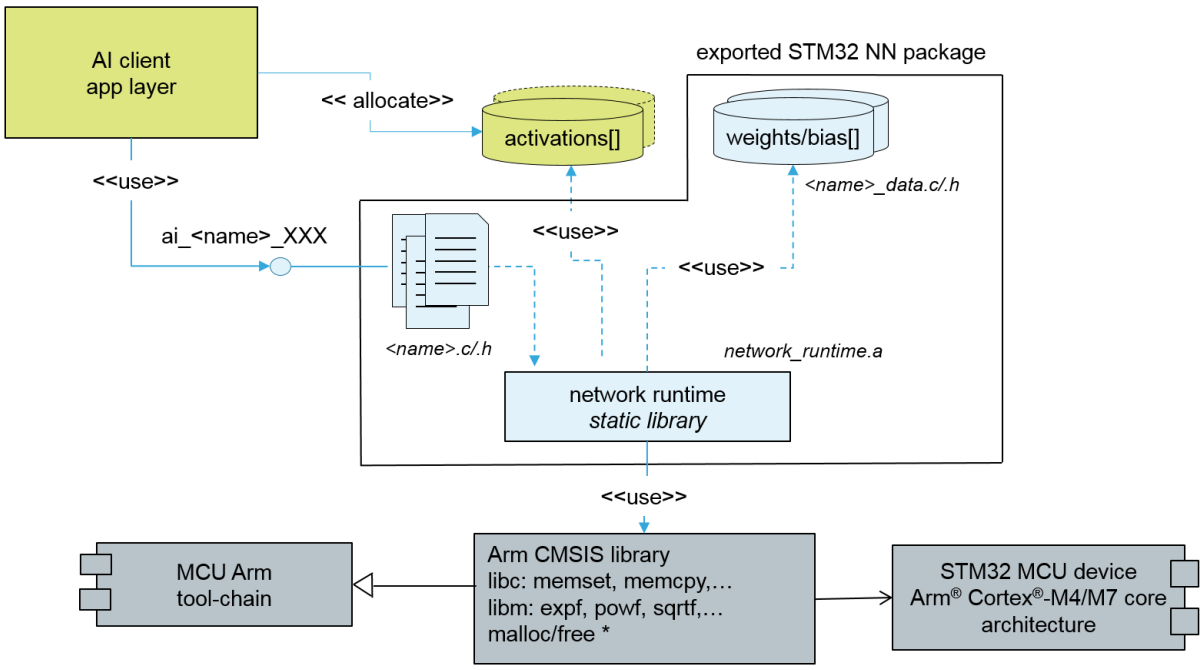
The *network_runtime.a* library or NN computing kernel library is provided as a static library:

- All unused symbols and methods are removed at link stage
- Not based on a network-graph interpreter approach (like ARM-NN, lite deployment environment) since it is not optimal for devices with limited memory resources.

## 7.1 Firmware integration

Figure 52 illustrates the MCU integration model and view (including run-time dependencies) of the generated STM32 NN package.

**Figure 52. MCU integration model and view**




For the application layer, the exported NN library is considered as a “black box” or self-content object. Only the specialized files, network topology (*<name>.c* and *<name>.h* files), and weights/bias parameters (*<name>_data.c* and *<name>_data.h* files) are provided as source files. They are based on a common network run-time library (*network_runtime.a*). The dependencies with the system run time are minimal:

- standard *libc* memory manipulation functions (`memcpy`, `memset`). They are generally provided by the MCU toolchain.
- CMSIS library to support the Cortex®-M optimized operations (FPU and DSP instructions). It is part of the STM32 HAL package.
- `malloc`/`free` is currently expected to support the *“Recurrent-type”* layer (*“GRU”* and *“LSTM”* layers). In future releases, the use of a static buffer allocation approach is planned instead. Performance impact is mitigated by the number of recurrent cell units and associated processing time.

- A mathematical library (with DSP/FPU support) is also requested to support the `expf`, `powf` , `tanhf`, and `sqrtf` functions.
- A minimal stack is requested (real value can be measured with the *"AI system performance"* application; refer to Section 9 AI system performance application).

*Note: All external dependencies must be solved during the end-user link stage (firmware image generation).*

*Activation memory buffers can be allocated dynamically in the heap or as a global array (.bss and .data sections). Refer to the `ai_<name>_init()` function to show how to pass the weights/bias buffer and activation memory buffers to the NN core library.*

## 7.2 Library source tree view

When the IDE project is created, the network runtime library is exported into sub-folder `<project_name>/Middlewares/ST/AI/`. Specialized or dedicated NN files are stored in standard STM32CubeMX `Inc` and `Src` sub-folders. Requested specific files from the CMSIS-DSP libraries are also added.

```
<project_name>
    |- Inc
    |      |- app_x-cube-ai.h /* entry points - MX_X_CUBE_AI_xx() fcts */
    |      |- bsp_ai.h        /* BSP AI adapt. for AI validation/systemperf application */
    |      |- constants_ai.h  /* BSP constant AI definition */
    |      |- <name_1>.h      /* specialized NN files */
    |      |- <name_1>_data.h
    |      |- <name_2>.h
    |      \- ...
    |- Src
    |      |- app_x-cube-ai.c
    |      |- <name_1>.c      /* specialized NN files */
    |      |- <name_1>_data.c
    |      \- ...
    | ...
    \--Middlewares
        \- ST/AI
             |-- include
             |    \- *.h               /* Internal/private AI headers */
             |-- lib
             |    \- network_runtime.a /* generic run-time library */
             \-- Application
                  \- SystemPerformance       /* generic sample application */
                      |- Inc
                      |   \- aiSystemPerformance.h
                      \- Src
                          \- aiSystemPerformance.c
```

The file name `network_runtime.a` depends on the X-CUBE-AI version and IDE used, such as `NetworkRuntime410_CM4_GCC.a` or `NetworkRuntime410_CM4_IAR.a`.

## 7.3 Multi-network inference API

The *app_x-cube-ai.c* and *app_x-cube-ai.h* files provide also a generic multi-network inference API, which can be used by the AI client application. It is very close to the native embedded inference client API (refer to [10]); only the `create()` function is different. The C-name string of the network must be passed to create the instance of the underlying network. This interface is mainly used by the add-on AI test applications to have a generic way to address the different embedded networks.

```
/* @file - app_x-cube-ai.h/.c - GENERATED CODE by STM32Cube MX */
 ...
const char* ai_mnetwork_find(const char *name, ai_int idx);

ai_error ai_mnetwork_create(const char *name, ai_handle* network, const ai_buffer*
network_config);

ai_bool ai_mnetwork_get_info(ai_handle network, ai_network_report* report);
ai_error ai_mnetwork_get_error(ai_handle network);
ai_handle ai_mnetwork_destroy(ai_handle network);
ai_bool ai_mnetwork_init(ai_handle network, const ai_network_params* params);
ai_i32 ai_mnetwork_run(ai_handle network, const ai_buffer* input, ai_buffer* output);
```

## 7.4 Re-entrance and thread safety considerations

No internal synchronization mechanism is implemented to protect the entry points against concurrent accesses. If the API is used in a multi-threaded context, the protection of the instantiated NN(s) must be guaranteed by the application layer itself.

To minimize the usage of the RAM, a same activation memory chunk (`SizeSHARED`) can be used to support multiple network. In this case, the user must guarantee that an on-going inference execution cannot be preempted by the execution of another network.

```
SizeSHARED = MAX(AI_<name>_DATA_ACTIVATIONS_SIZE) for name = "net1" … "net2"
```

*Note: If the preemption is expected for real-time constraint or latency reasons, each network instance must have its own and private activation buffer.*

## 7.5 Code and data placement considerations

For the current STM32 memory architecture (STM32L4/STM32F4/STM32F3-based and STM32F7/STM32H7-based), no specific data or code placement is expected for performance reason. The Flash ART peripheral and the Arm® core sub-system cache (Cortex®- M7-based architecture) efficiently limit memory latency side effects. NN code (*.text* section) and RO data (*.rodata* section) can be placed in the internal Flash area. RW data (*.data* and *.bss* sections) must be placed in the embedded SRAM. The client stack is used; it must be placed in a zero-wait-state memory.

*Note: There is no memory retention requirement on the activation buffer. It can be really considered as a scratch or working buffer. Between two inferences, buffer can be reused for pre-processing purpose for example, or the associated memory device can be switched off when the system goes into Deep Sleep.*

## 7.6 Debug considerations

The library must be considered as an optimized black box in binary format (sources files are not deliveries). There is no support for run-time internal data or state introspection. Mapping and port of the NN is guaranteed by the X-CUBE-AI generator. Some integration issues can be highlighted by the `ai_<name>_get_error()` function.

# 8 Embedded inference client API

To use the generated NN code, a simple embedded inference client API is generated (see the `ai_<name>_XX()` functions in Figure 52). It is part of the `<project_name>/Src/<name>.h` file. All functions and macros are generated according to the C-network name provided. For usage and detailed description, refer to [10].

# 9 AI system performance application

The AI system performance application is a self and bare-metal on-device application, which allows the out-of-the-box measurement of the critical system integration aspects of the generated NN. The accuracy performance aspect is not and cannot be considered here. The reported measurements are:

- CPU cycles by inference (duration in ms, CPU cycles, CPU workload)
- Used stack and used heap (in bytes)

Execute the following series of steps in sequence to run the application:

1. Open and configure a host serial terminal console connected via a COM port (usually supported by a Virtual COM port over a USB connection, such as an ST-LINK/V2 feature).
2. Set the COM setting. It must be aligned with the setting of the STM32 USART (refer to Section 3.2 Hardware and software platform settings):
   - 115200 bauds
   - 8 bits
   - 1 stop bit
   - No parity
3. Reset the board to launch the application

When the application is running, typing `p` or `P` in the console suspends the main loop. The application embeds a minimal interactive console, which supports the following commands:

```
Possible key for the interactive console:
    [q,Q]      quit the application
    [r,R]      re-start (NN de-init and re-init)
    [p,P]      pause
    [h,H,?]    this information
     xx        continue immediately
```

## 9.1 System run-time information

Figure 53 and Figure 54 show the first part of the log, which indicates the useful information of the STM32 run-time or executing environment for the Keil® and Atollic IDEs respectively: device ID, system clock value, used toolchain, and others.

**Figure 53. System run-time information - Keil® IDE**

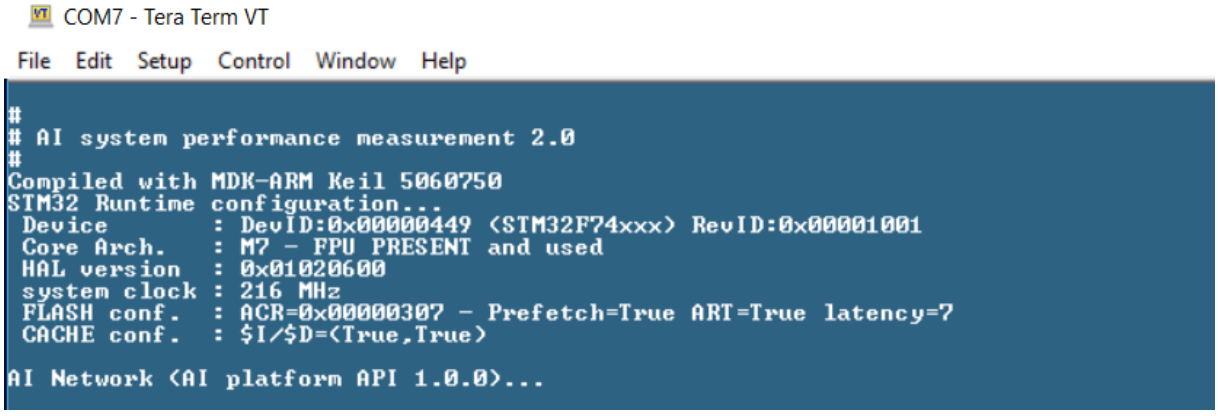
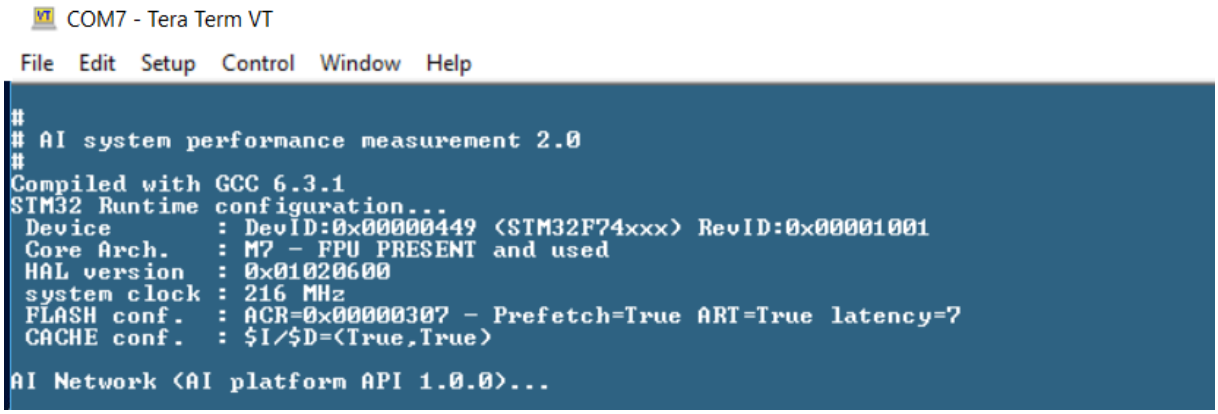


Figure 54. **System run-time information - Atollic IDE**




*Note: To retrieve these informations in the log, type `r` or `R` in the console during the execution of the main loop.*

## 9.2 Embedded C-model network information

This second part shown in Figure 55 indicates the main static characteristics of the generated NN(s). In particular, it provides the RAM/Flash size (in bytes, respectively activation/weights fields) and the logical complexity (MACC, complexity field). Shape definitions of the input and output tensors are also reported. These informations are available also by the client application code through the `ai_<name>_get_info()` client API function.

Figure 55. **C-model network information**

```
Found network "net1"
Creating the network "net1"..
Network configuration...
 Model name          : net1
 Model signature     : dc582ba86ee8a11fd06b698b258a4310
 Model datetime      : Sun Dec  9 08:56:25 2018
 Compile datetime    : Dec  9 2018 09:01:15
 Runtime revision    :  (3.3.0)
 Tool revision       : (rev-) (3.3.0)
Network info...
  signature          : 0x0
  nodes              : 7
  complexity         : 874970 MACC
  activation         : 45572 bytes
  weights            : 794136 bytes
  inputs/outputs     : 1/1
  IN tensor format   : HWC layout:90,3,1 (s:270 f:AI_BUFFER_FORMAT_FLOAT)
  OUT tensor format  : HWC layout:1,1,6 (s:6 f:AI_BUFFER_FORMAT_FLOAT)
Initializing the network

Found network "net2"
Creating the network "net2"..
Network configuration...
 Model name          : net2
 Model signature     : b773f449281f9d970d5b982fb57db61f
 Model datetime      : Sun Dec  9 08:57:12 2018
 Compile datetime    : Dec  9 2018 09:01:15
 Runtime revision    :  (3.3.0)
 Tool revision       : (rev-) (3.3.0)
Network info...
  signature          : 0x0
  nodes              : 21
  complexity         : 4550024 MACC
  activation         : 64004 bytes
  weights            : 159536 bytes
  inputs/outputs     : 1/1
  IN tensor format   : HWC layout:49,10,1 (s:490 f:AI_BUFFER_FORMAT_FLOAT)
  OUT tensor format  : HWC layout:1,1,12 (s:12 f:AI_BUFFER_FORMAT_FLOAT)
Initializing the network
```

*Note: To retrieve these informations in the log, type `r` or `R` in the console during the execution of the main loop.*

## 9.3 Embedded C-model run-time performance

As illustrated in Figure 56 and Figure 57, the last part of the log (main loop) reports the measured out-of-the-box system performance. Random inputs are injected in the network to measure the number of CPU cycles by inference (`CPU cycles`). The `CPU workload` and `cycles/MACC` are deduced from this value. During the measurement, the IRQs are masked.

- `duration` indicates the duration in ms for one inference.
- `CPU cycles` indicates the number of CPU cycles for one inference.
- `CPU workload` indicator corresponding to the associated CPU workload during 1 s.
- `cycles/MACC` is the number of CPU cycles by MACC operation.

**Figure 56. C-model run-time performance**

```
Running PerfTest on "net1" with random inputs (16 iterations)...
---------------

Results for "net1", 16 inferences @216MHz/216MHz (complexity: 874970 MACC)
 duration      : 28.047 ms (average)
 CPU cycles    : 6058169 -198595/+29532 (average,-/+)
 CPU Workload  : 2%
 cycles/MACC   : 6 (average for all layers)
 used stack    : 352 bytes
 used heap     : 0:0 0:0 (req:allocated,req:released) cfg=0

Running PerfTest on "net2" with random inputs (16 iterations)...
---------------

Results for "net2", 16 inferences @216MHz/216MHz (complexity: 4550024 MACC)
 duration      : 150.323 ms (average)
 CPU cycles    : 32469972 -42110/+14510 (average,-/+)
 CPU Workload  : 15%
 cycles/MACC   : 7 (average for all layers)
 used stack    : 352 bytes
 used heap     : 0:0 0:0 (req:allocated,req:released) cfg=0
```

**Figure 57. C-model run-time performance with heap and stack checking**

```
Running PerfTest on "network" with random inputs (16 iterations)...
---------------

Results for "network", 16 inferences @80MHz/80MHz (complexity: 3729752 MACC)
 duration      : 569.124 ms (average)
 CPU cycles    : 45529988 -134210/+211019 (average,-/+)
 CPU Workload  : 56%
 cycles/MACC   : 12 (average for all layers)
 used stack    : 284 bytes
 used heap     : 16:29568 16:29568 (req:allocated,req:released) cfg=3
```

*Note: "used heap" indicates the number of `malloc()` and cumulated allocated size (respectively `free()`) requested during the execution of all inferences. The counter is not reset between two inferences or test iterations to detect hypothetic memory leak. In the present case, the minimum heap size is `29568 / #iter = ~2Kbytes`.*

**Caution:** Today, the heap monitor is only supported for a GCC-based environment.

# 10 AI validation application

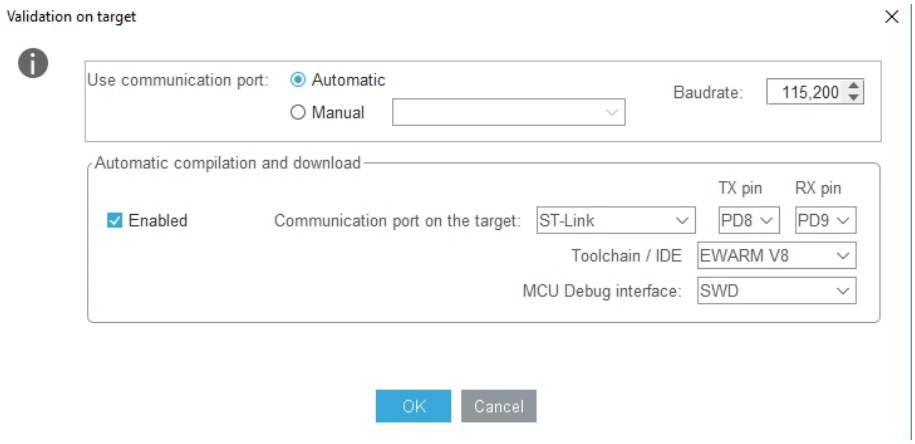
The *AI validation* application is a self and bare-metal on-device application, which supports the *Validation on device* as presented in Section 6.2 Validation engine. It provides a USART-based interface with the host to export the inference API.

The whole AI validation application is either generated and programmed as a normal IDE project, or compiled automatically and downloaded from STM32CubeMX itself (refer to Figure 58).

If this application is not generated automatically, the following steps are requested:

1. When the board is programmed, reset or restart it.
2. Reopen (if closed) the `ioc` file with which firmware was generated.
3. Go to the AI configuration panel and open the *<network_name>* tab, which must validated.
4. Click on the [**Validation on target**] button to start the validation process. Before clicking on the [**OK**] button, the user has the possibility to indicate the host COM port that is used as shown in Figure 58. Otherwise, all available COM ports are discovered to detect a valid connected STM32 board (the first board found is used).

**Figure 58. Host COM port selector for validation on device**

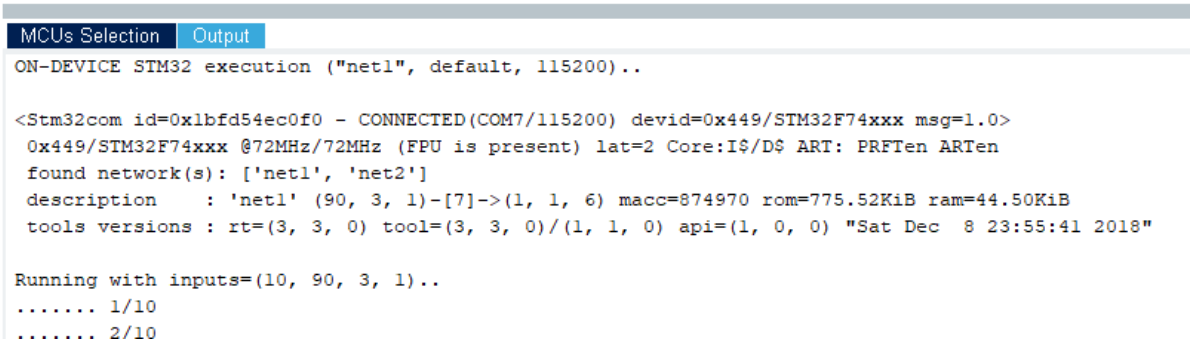



As for *Validation on desktop*, the final result is reported in the [**Validation status**] field. More detailed information is reported in the UI log console.

## 10.1 System run-time information

The first part of the reported log shown in Figure 59 indicates the main system information: device ID, clock frequency, memory sub-system configuration, list of the embedded networks. For the validated network, shape-in and shape-out tensor description is provided as well as used AI tools versions.

**Figure 59. System run-time information**



## 10.2 Embedded C-model run-time performance

The second part of the log shown in Figure 60 reports the out-of-the-box system performance measurements (`duration` average executing time by inference). `cycles/MACC` is deducted from the `duration` value. During the measurement, the IRQs are masked.

- `duration` indicates the duration in ms for one inference.
- `CPU cycles` indicates the number of CPU cycles for one inference.
- `cycles/MACC` is the number of CPU cycles by MACC operation.

**Figure 60. C-model run-time performance**

```
....... 9/10
....... 10/10
 RUN Stats    : batches=10 dur=24.266s tfx=21.957s 0.491KiB/s (wb=10.547KiB,rb=240B)

Results for 10 inference(s) @72/72MHz (macc:874970)
 duration    : 78.846 ms (average)
 CPU cycles  : 5676924 (average)
 cycles/MACC : 6.49 (average for all layers)
```

## 10.3 Layer-by-layer run-time performance

The next part of the log shown in Figure 61 provides the additional information about the generated C model: name and type of the implemented C layer (`Clayer` / `id` / `desc`), output shape (`oshape`), and average executing time by inference (`ms`).

**Figure 61. Layer-by-layer results - Validation on target**

```
Inspector report (layer by layer)
 signature      : EF7C5473
 n_nodes        : 7
 num_inferences : 10

Clayer  id  desc                          oshape            ms
--------------------------------------------------------------------------------
0       0   10011/(Merged Conv2d / Pool)  (10, 44, 1, 128)  24.277
1       4   10005/(Dense)                 (10, 1, 1, 128)   53.165
2       4   10009/(Nonlinearity)          (10, 1, 1, 128)   0.010
3       5   10005/(Dense)                 (10, 1, 1, 128)   1.303
4       5   10009/(Nonlinearity)          (10, 1, 1, 128)   0.010
5       6   10005/(Dense)                 (10, 1, 1, 6)     0.066
6       6   10014/(Softmax)               (10, 1, 1, 6)     0.015
                                                            78.846 (total)
```

## 10.4 Final result for validation on target

The last part of the log shown in Figure 62 provides the final result of the validation process. It is similar to the result of the validation on desktop but only the L2 error on the last layer is reported (refer to Section 6.2 Validation engine).

**Figure 62. Final report for validation on target**

```
  MACC / frame: 874970
  ROM size:     775.52 KBytes
  RAM size:     44.50 KBytes (Minimum: 44.50 KBytes)


Matching criteria: L2 error < 0.01 on the output tensor

  Ref layer 6 matched with C layer 6, error: 0.0010825497

Validation: OK
```

## 10.5 Returned error during the connection

The following USART setting is used by default:

- 8 bits
- 1 stop bit
- No parity
- 115200 bauds

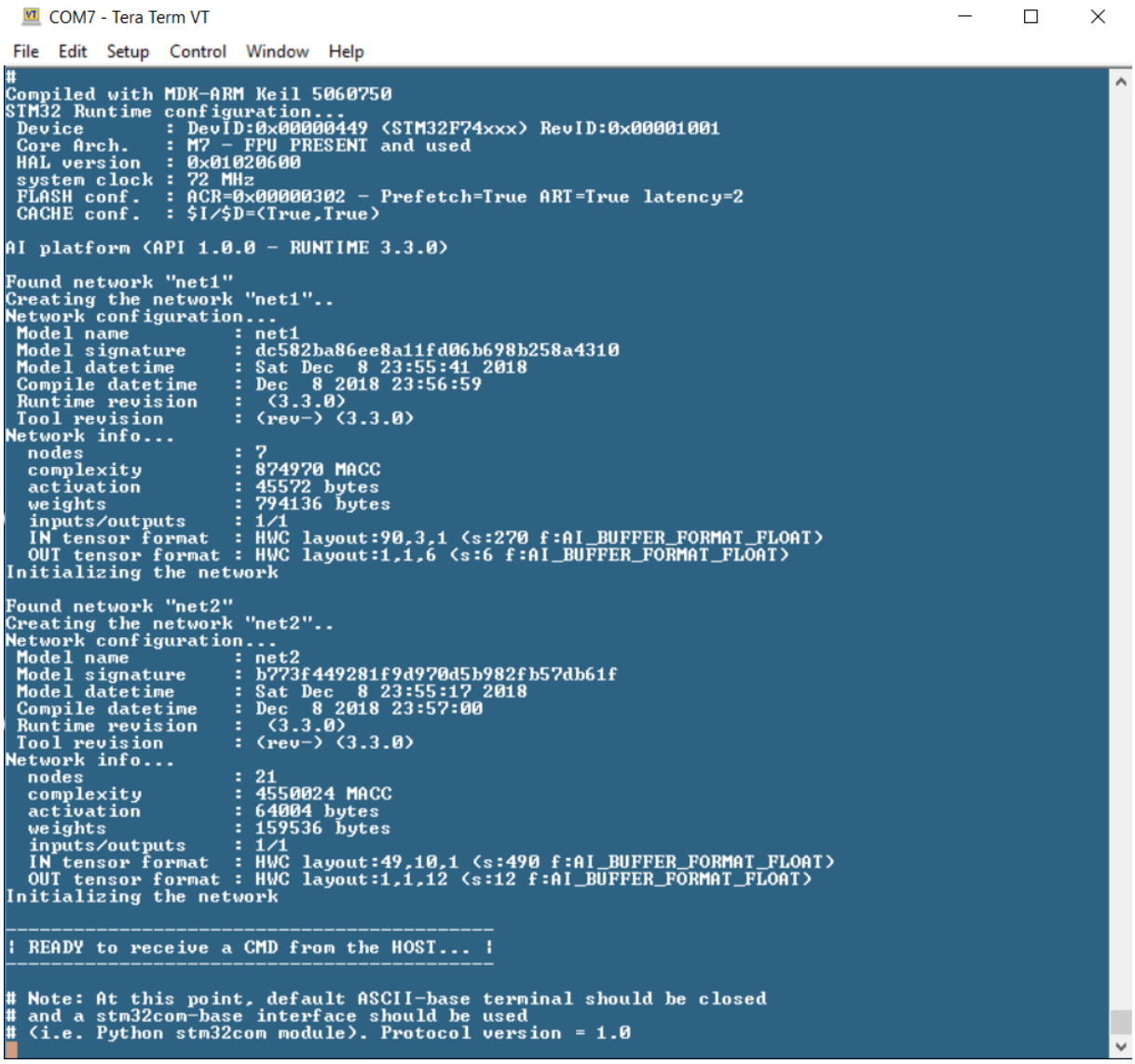
If redefined by the user, the baud rate must be kept aligned across all settings (refer to Section 3.2 Hardware and software platform settings and Figure 58. Host COM port selector for validation on device).

### 10.5.1 Error: no connected board, invalid firmware, or board restart needed

Indicates that no board is connected or can be found, or that firmware is not the expected *"AI Validation"* firmware. This error can also indicate an incoherent firmware state, in which case the board must be restarted.

To check that the firmware is correctly programmed, open a host serial terminal console at boot time, which generates an ASCII-based log. Do not forget to close the connection before to launching the *Validation on target* process again.

Figure 63. **AI valid - Initial log**




### 10.5.2 Error: *"network_name"* is not a valid network

Indicates that the expected C model identified by its name is not available in the connected board. See the UI log console (*Outputs* window) for more details.

### 10.5.3 Error: the embedded STM32 model does not match the C model

Indicates that the signature of the generated C model is not coherent with the expected model. The parameters used to check the signature are:

- RAM/ROM size
- MACC
- Number of nodes
- Tool versions

See the UI log console (*Outputs* window) for more details.

# 11 AI template application

When selected, the generated IDE project is not really a complete AI template application with an example of a basic AI application to use the generated C models. *aiSystemPerformance.h* and *aiSystemPerformance.c* files represent a good example for this purpose. Only the specific generated files are generated. This can be used as started point to develop an initial bare-metal application with two simple entry points (`init` and `process` functions). Refer to [10] for details.

# 12 Supported toolboxes and layers for Deep Learning

The X-CUBE-AI core currently supports the following DL toolboxes:

- Keras: //keras.io/
- TensorFlow™ Lite: www.tensorflow.org/lite/
- ONNX: //onnx.ai/

*Note: TensorFlow is a trademark of Google Inc.*

For each toolbox, only a subset of all possible layers and layer parameters are supported, depending on the expressive power of the network C API, and on the parser for the specific toolbox. Supported configurations are detailed in [7], [8] and [9].

# 13 Error handling

The X-CUBE-AI core handles a range of different errors and reports them to the user as detailed in [6].

# 14 FAQs

## 14.1 Log files for debug purpose?

When a *"validation on target"* process is performed, all messages exchanged with the target (including the data) are stored in a dedicated log file:

`C:\Users\<username>\.stm32cubemx\ai_stm32_msg.log`

If a validation or generation process fails, additional debug/log info is available in file:

`C:\Users\<username>\.stm32cubemx\STM32CubeMX.log`

## 14.2 Unable to compile file *"arm_dot_prod_f32.c"*

The compilation of `arm_dot_prod_f32.c` may fail when the IDE project files are regenerated:

```
compiling arm_dot_prod_f32.c...
../Drivers/CMSIS/Include/arm_math.h(314): error: #35:
#error directive: "Define according the used Cortex core ARM_MATH_CM7, ARM_MATH_CM4,
ARM_MATH_CM3, ARM_MATH_CM0PLUS or ARM_MATH_CM0"
#error "Define according the used Cortex core ARM_MATH_CM7, ARM_MATH_CM4, ARM_MATH_CM3,
ARM_MATH_CM0PLUS or ARM_MATH_CM0"
../Drivers/CMSIS/DSP_Lib/Source/BasicMathFunctions/arm_dot_prod_f32.c: 0 warnings, 1 error
```

According to the targeted STM32 device, the following C defines must be redefined in the project setting:

```
ARM_MATH_CM7,__FPU_PRESENT=1
```

## 14.3 Used heap or stack: disabled or not yet supported

This log indicates that the stack (respectively heap) monitor is explicitly disabled or not yet implemented for the toolchain used.

**Table 4. Heap and stack monitoring support**

| Toolchain | Stack monitor | Heap monitor | Note |
|---|---|---|---|
| GCC | Supported | Supported | STM32CubeIDE |
| IAR Systems EWARM 8.x | Supported | Not implemented | - |
| Keil® MDK-ARM | Not implemented | Not implemented | - |

Example of heap and stack monitoring activation:

```
/* @file: aiSystemPerformance.c */
...
#if defined(__GNUC__)
#define _APP_STACK_MONITOR_ 1
#define _APP_HEAP_MONITOR_ 1
#elif defined (__ICCARM__)
#define _APP_STACK_MONITOR_ 1
#define _APP_HEAP_MONITOR_ 0
#else
#define _APP_STACK_MONITOR_ 0
#define _APP_HEAP_MONITOR_ 0
#endif
...
```

## 14.4 Why is *“used heap”* always zero?

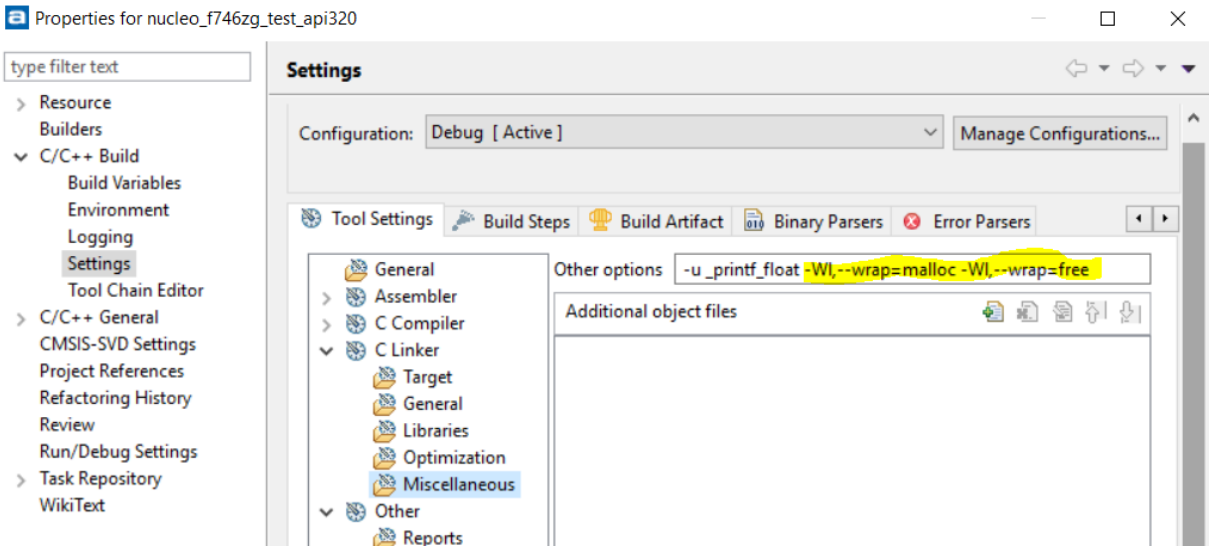
For GCC-based projects only:

```
used heap : 0:0 0:0 (req:allocated,req:released) cfg=0
```

Such a result is not necessarily a problem. Most of the *network_runtime.a* library is based on a preallocated R/W buffer scheme (activation buffers). For some specific layers (recurrent layer type), the current implementation requests to allocate dynamically a part of these work buffers through the `malloc()` function.

The heap monitor is based on a toolchain specific mechanism, which allows the wrapping of the system `malloc()` and `free()` functions. To enable this wrapping, the `-Wl,--wrap=malloc -Wl,--wrap=free`miscellaneous linker options must be set in the build system as shown in Figure 64.

**Figure 64. Linker options to enable the heap monitor**




## 14.5 Formatted floating-point numbers are empty for a GCC-based project

The following link option must be added to output a formatted floating-point number:

```
-u _printf_float
```

## 14.6 CPU cycles/MACC?

Refer to Section 4.5.1 CPU cycles/MACC? and Section 9 AI system performance application.

## 14.7 Is it necessary to enable or configure a TIMER peripheral?

This is not necessary. The mechanism to measure the number of CPU cycles by inference uses a dedicated Arm® Cortex®-M debug unit (DWT: Data Watch-point and Trace unit), which is available on all supported STM32 devices. It uses a free-running counter that is clocked by the CPU clock (HCLK system clock).

## 14.8 How to update only the exported NN library in my generated project?

This is straightforward if the STM32CubeMX design guide lines are applied (`/* USER CODE BEGIN*/`, `/*... END*/` tags) for the changes to the exported and generated files. The `<project_name>.ioc` file can be directly re-opened to upload a new NN model and to update the IDE project.

## 14.9 Is it possible to export an NN library for a non-STM32CubeMX-based project?

Since the exported NN library is located in a well-defined and self-content sub-folder (refer to Section 7 Generated STM32 NN library), this AI sub-folder can be fully copied in the source tree of the destination project:

1. Create a new dummy STM32CubeMX project for the user's STM32 MCU device.
2. Generate the IDE project for the user's toolchain/IDE. This step is requested to include the correct `network_runtime.a` library, which is toolchain- and Arm®-Cortex®-M-architecture dependent.
3. Copy the generated AI sub-folder in the source tree of the new project.
4. Add the `network.c` and `network_data.c` files, and the `network_runtime.a` library to the system build and update the C/C++ compiler and linker options as described in Section 7 Generated STM32 NN library.
5. Return to step 1 to update and evaluate a modified NN model.

## 14.10 Command-line interface?

A complete command-line interface is provided in the X-CUBE-AI Expansion Package (refer to [6]).

# 15 References and documentation

## 15.1 References used in this user manual

**Table 5. References**

| ID | Description | Link |
|---|---|---|
| [1] | NUCLEO-F746ZG development kit | www.st.com/en/product/nucleo-f746zg |
| [2] | STM32CubeIDE | www.st.com/stm32cubeide |
| [3] | IAR Embedded Workbench® IDE - ARM v8.x or v7.x | www.iar.com/iar-embedded-workbench |
| [4] | µVision® V5.25.2.0 - Keil® MDK-ARM Professional Version | www.keil.com |
| [5] | STM32CubeMX - initialization code generator | www.st.com/en/product/stm32cubemx |
| [6] | X-CUBE-AI stm32ai command-line interface | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/command_line_interface.html`(1) |
| [7] | X-CUBE-AI Keras toolbox support | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/supported_ops_keras.html`(1) |
| [8] | X-CUBE-AI ONNX toolbox support | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/supported_ops_onnx.html`(1) |
| [9] | X-CUBE-AI TensorFlow™ Lite toolbox support | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/supported_ops_tflite.html`(1) |
| [10] | X-CUBE-AI embedded inference client API | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/embedded_client_api.html`(1) |
| [11] | X-CUBE-AI evaluation report and metrics | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/evaluation_metrics.html`(1) |
| [12] | X-CUBE-AI quantized model and quantize command | Documentation embedded in X-CUBE-AI Expansion Package in `Documentation/quantization.html`(1) |

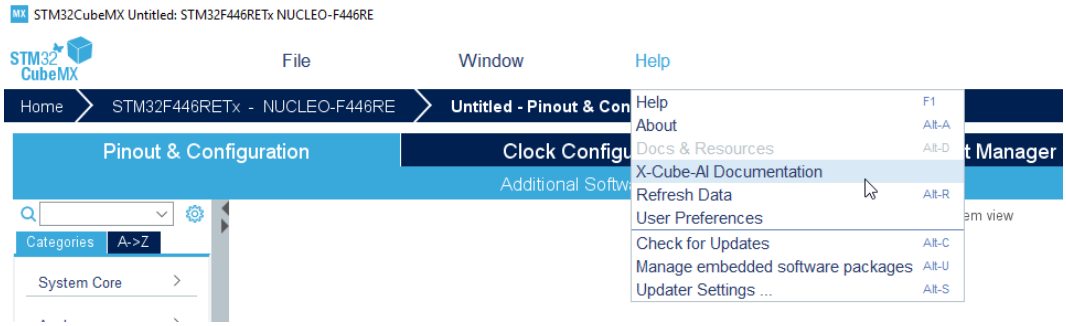
1. *Refer to Section 15.2 Access to X-CUBE-AI in-package documentation.*

## 15.2 Access to X-CUBE-AI in-package documentation

Follow one of the two solutions proposed below for access to the documentation available in the X-CUBE-AI Expansion Package:

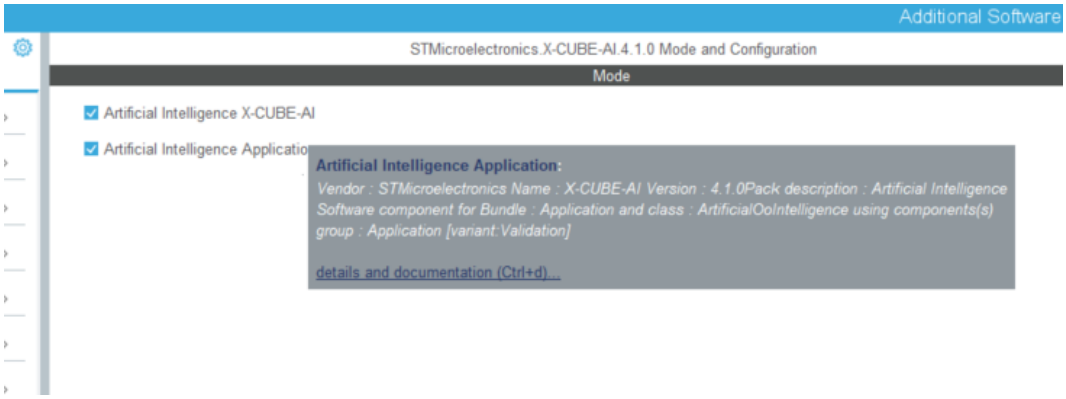
1. Direct access through the menu:
   a. Click [**Help**]>[**X-CUBE-AI Documentation**] as shown in Figure 65.

**Figure 65. Direct menu access to in-package documentation**



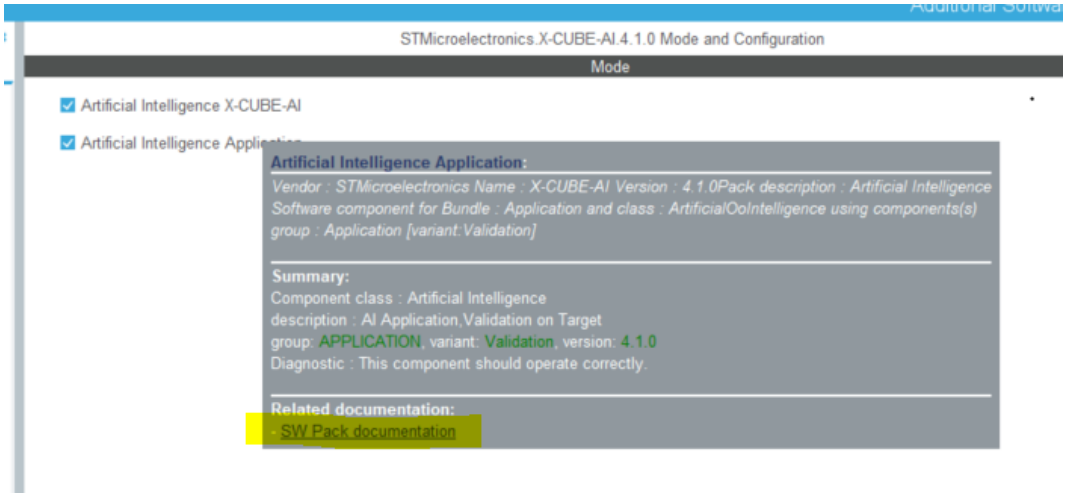
2. Access through X-CUBE-AI modes:
   a. Hover the cursor over one of the X-CUBE-AI modes and click on [**details and documentation**] as shown in Figure 66.

**Figure 66. In-package documentation access through X-CUBE-AI modes (1 of 2)**



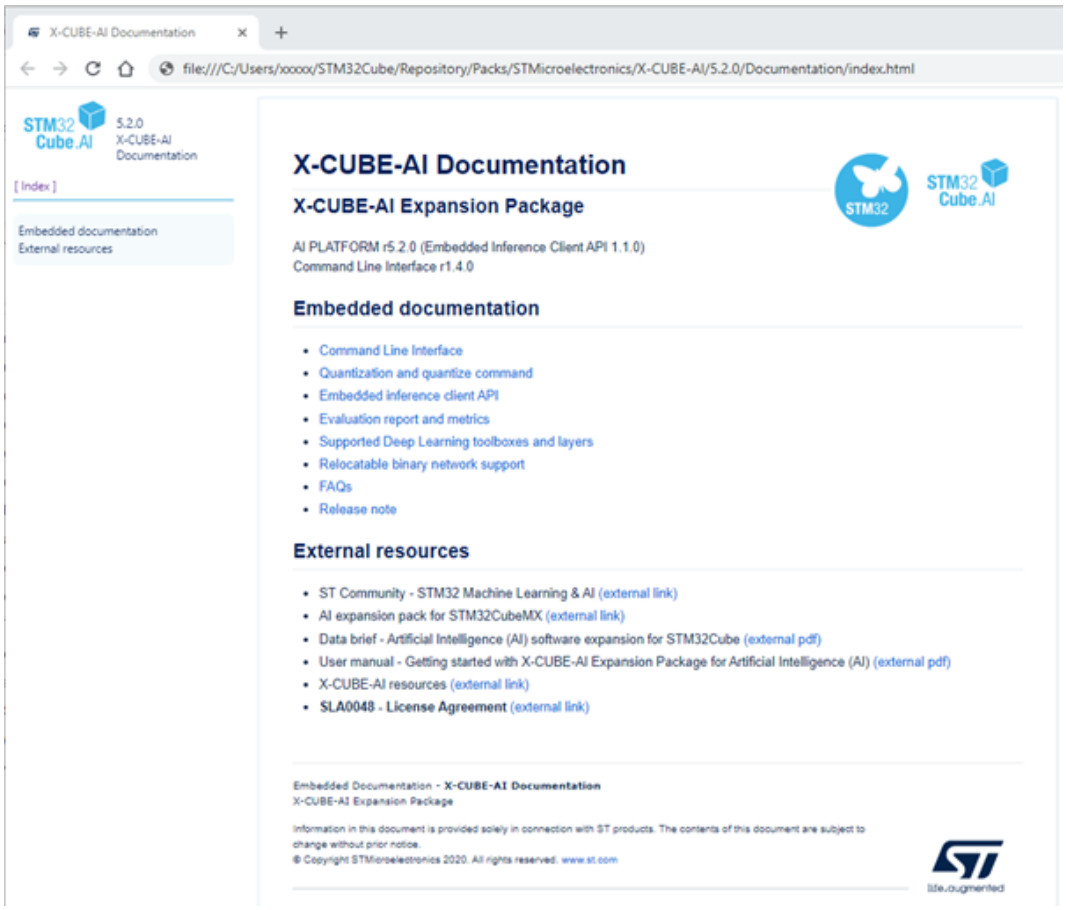

   b. Click on [**SW Pack documentation**] as shown in Figure 67.

**Figure 67. In-package documentation access through X-CUBE-AI modes (2 of 2)**




   c. A browser window opens that lists the available X-CUBE-AI documentation as shwon in Figure 68.

Figure 68. **In-package documentation index**



## Revision history

Table 6. **Document revision history**

| Date | Revision | Changes |
|---|---|---|
| 15-Jan-2019 | 1 | Initial release. |
| 19-Jul-2019 | 2 | Updated for X-CUBE-AI 4.0.0:<br>• added quantization<br>• added the command-line interface<br>• added the support of TensorFlow™ Lite<br>Simplified the user manual referring to documents in the Expansion Package for:<br>• *Section 8 Embedded inference client API*<br>• *Section 11 AI template application*<br>• *Section 12 Supported toolboxes and layers for Deep Learning*<br>• *Section 13 Error handling* |
| 11-Oct-2019 | 3 | Updated for X-CUBE-AI 4.1.0:<br>• added the support of the quantization model for TensorFlow™ Lite<br>• added the support of external memories for validation<br>Simplified the user manual:<br>• removed *Section 14.2 Custom data set file format?*<br>• added *Section 15.2* explaining how to get access to in-package documentation |
| 6-Jan-2020 | 4 | Updated for X-CUBE-AI 5.0.0:<br>• added the support of the quantization model for ONNX<br>• added the use of the activation buffer for the input buffer<br>• added the direct menu access to in-package documentation |
| 8-Jun-2020 | 5 | Updated for X-CUBE-AI 5.1.0:<br>• updated screenshots<br>• added split-weight option in *Section 4.3 Uploading a pre-trained DL model file*<br>• added memory usage graphs and layer information in:<br>– *Section 4.4.2 Generated C-model graph representation*<br>• updated toolboxes in:<br>– *Section 1.2 How does X-CUBE-AI complement STM32Cube?*<br>– *Section 14.4 Used heap or stack: disabled or not yet supported* |
| 1-Oct-2020 | 6 | Updated for X-CUBE-AI 5.2.0:<br>• updated screenshots<br>• support for the STM32L5 Series based on the Cortex®-M33 core<br>• updated the advanced settings for external memories in *Section 4.3 Uploading a pre-trained DL model file* |
| 4-Mar-2021 | 7 | Updated for X-CUBE-AI 6.0.0:<br>• updated screenshots<br>• added STM32WL Series compatibility<br>• updated Section 4.3 Uploading a pre-trained DL model file<br>• created Section 4.4 Advanced settings<br>• updated Section 6.2 Validation engine and added *Specific attention for custom data*<br>• updated Section 15.1 References used in this user manual<br>• removed *Section 14.2 Multi-network limitations?* |

# Contents

# List of tables

# List of figures