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 first 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, STM32L4, STM32L4+, STM32F7, STM32H7, or STM32WB series can also be used with minor adaptations.

The second part of the document details the NN library automatically generated by X-CUBE-AI, and the embedded client inference API. It also describes the use of X-CUBE-AI for AI performance and validation, and the features used in the various DL toolboxes.

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 r3.3.0
- Embedded inference client API 1.0.0

The pre-trained Keras DL models used for the examples in this document are:

- https://github.com/Shahnawax/HAR-CNN-Keras: Human Activity Recognition using CNN in Keras
- https://github.com/Shahnawax/KWS-ANN-KERAS: KWS-ANN-KERAS - Key Word Spotting

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 the conception to the realization, among which:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards.
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions.
  - STM32CubeMonitor-Power (STM32CubeMonPwr), a monitoring tool to measure and help in the optimization of the power consumption of the MCU.
- STM32Cube™ MCU Packages, comprehensive embedded-software platforms specific to each microcontroller series (such as STM32CubeF4 for the STM32F4 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 the HW
  - A consistent set of middleware components such as RTOS, USB, TCP/IP, and graphics.
  - All embedded software utilities with full sets of peripheral and applicative examples.

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

X-CUBE-AI extends STM32CubeMX by providing an automatic neural network library generator optimized in computation and memory (RAM and Flash) that converts pre-trained Neural Networks from most used DL frameworks (such as Caffe, Keras, Lasagne, and ConvnetJS) 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 size) for a user's NN.

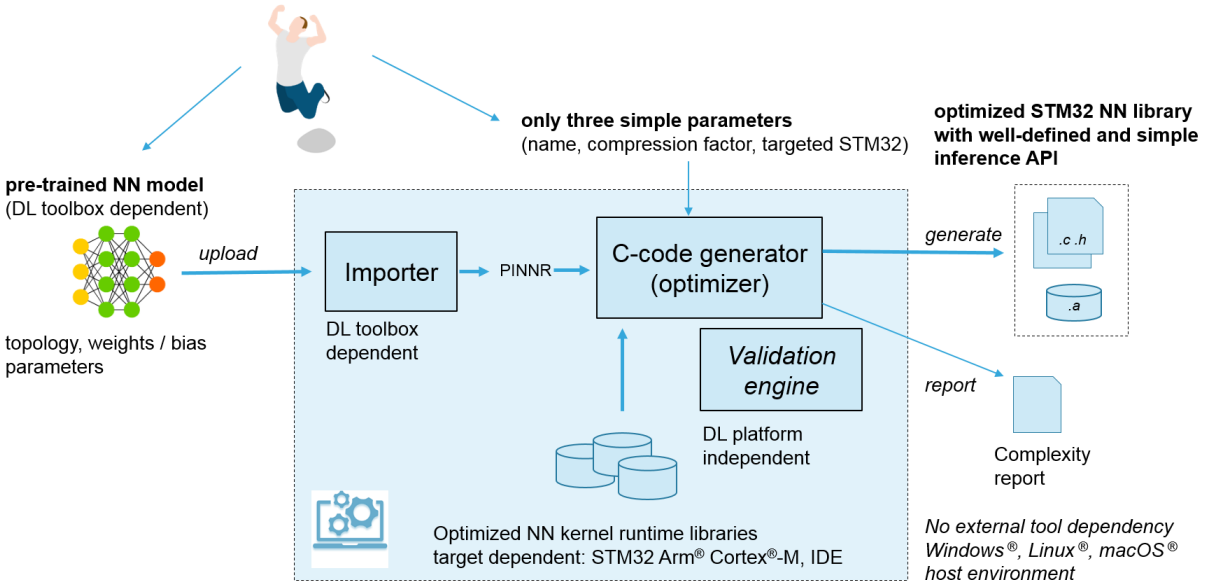
The X-CUBE-AI tool can generate three kinds of projects:

- 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 an application including multi-network support

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 1.3 . 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).

Figure 1. X-CUBE-AI core engine

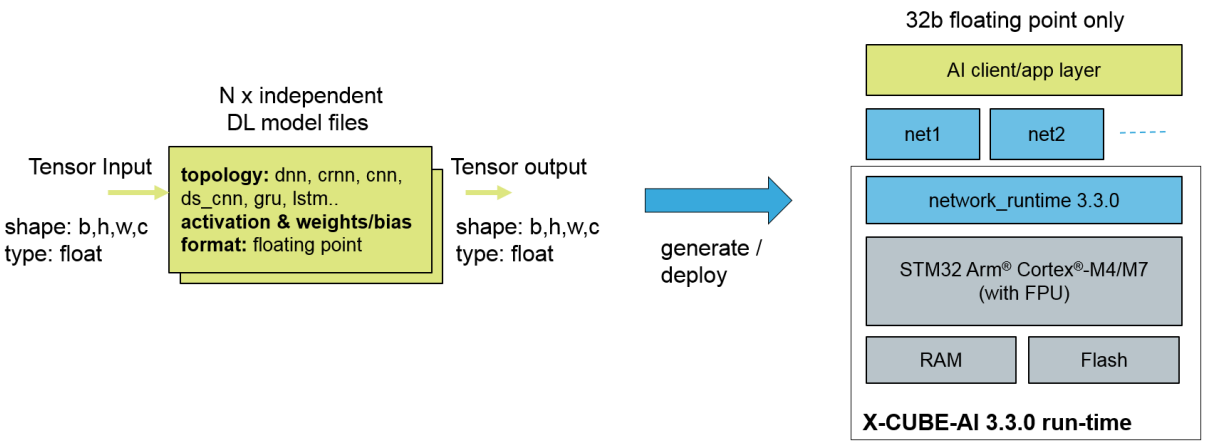


A simple configuration interface is exposed. With the pre-trained DL model file, only few parameters are requested:

- 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 3.3.0 overview

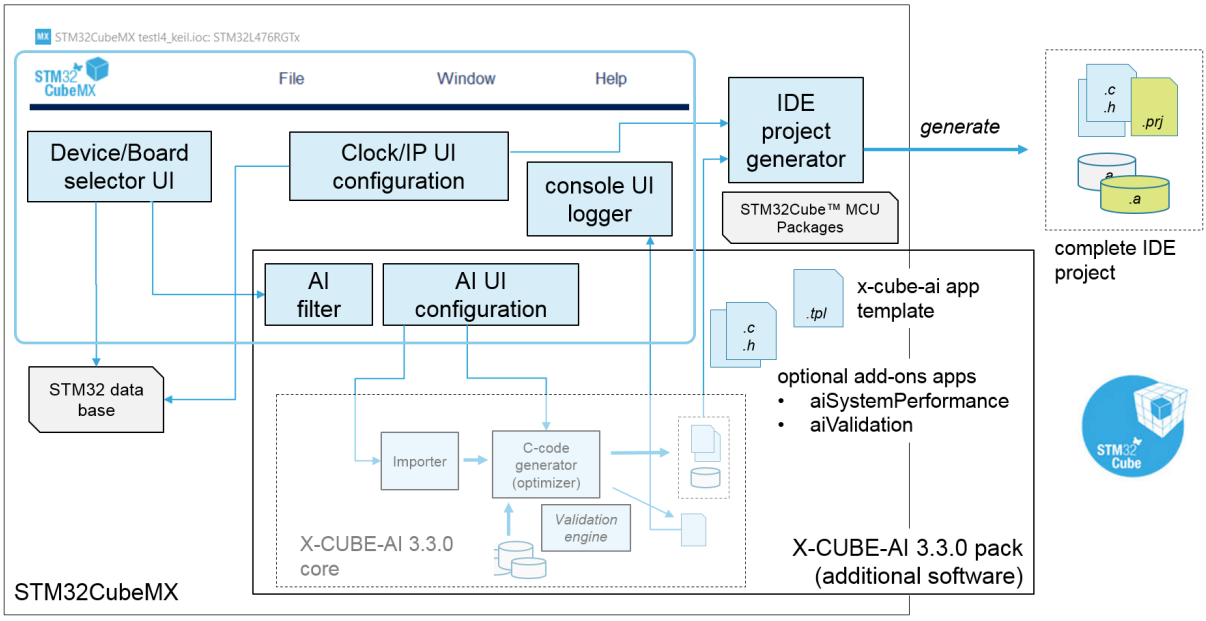


- Only simple tensor input and simple tensor output are supported
  - 4-dim shape: batch, height, width, channel (refer to Section 8.1 Input and output x-D tensor layout)
  - Floating-point type only
- Only 32-bit floating-point generated C models are supported
  - Allows quick generation process (refer to Section 6.1 Graph flow and memory layout optimizer)
  - Guarantees by construction the accuracy of the pre-trained model

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 3. 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 an IP 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 or -M7 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 flash it. No additional code or modification is requested to the end user.

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 5.0.1 or later
- Additional SW pack - STM32CubeMX AI 3.3.0 pack

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

- TrueSTUDIO® for STM32 v9.0.1 or later (atollic.com/truestudio)
- IAR Embedded Workbench™ IDE - ARM v8.x or v7.x (www.iar.com/iar-embedded-workbench)
- µVision® V5.25.2.0 - Keil® MDK-ARM Professional Version (www.keil.com)
- System Workbench for STM32 (SW4STM32)
- GNU Arm Embedded Toolchain (developer.arm.com/open-source/gnu-toolchain/gnu-rm)

All operating systems supported by STM32CubeMX can be used:

- Windows® 10 and Windows® 7
- Ubuntu® 18.4 and Ubuntu® 16.4 (or derived)
- 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.

1.7 License

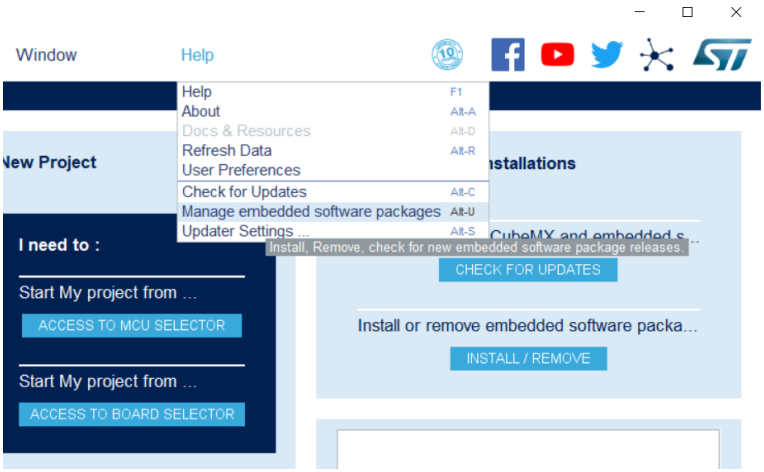
X-CUBE-AI is delivered under the *Mix Ultimate Liberty+OSS+3rd-party V1* software license agreement (SLA0048).

2 Installing X-CUBE-AI

After downloading, installing, and launching STM32CubeMX (version 5.0.1 or later), the X-CUBE-AI Software Package can be installed in a few steps.

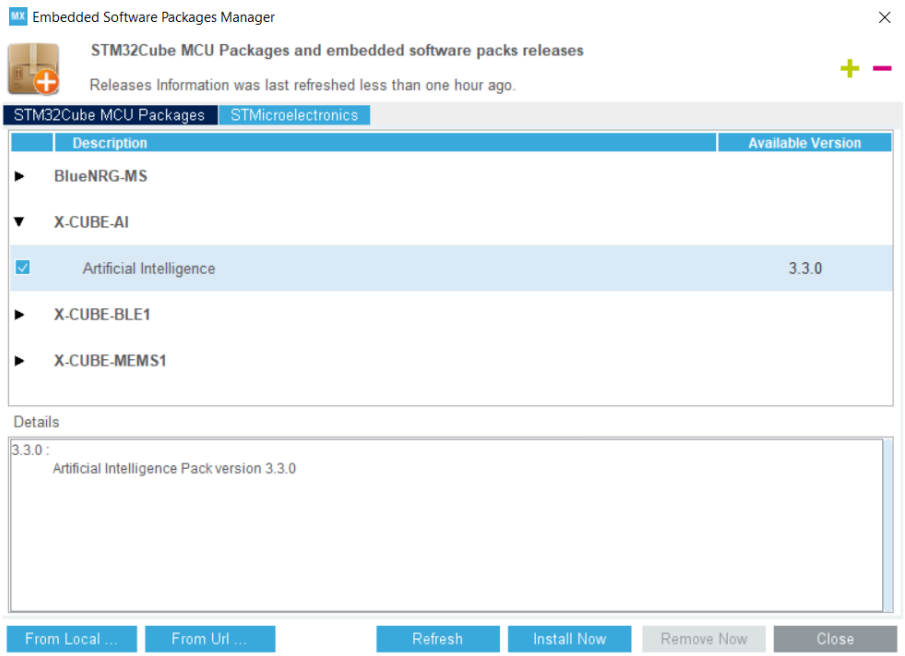
1. From the menu, select `[Help]` > `[Manage embedded software packages]` or directly click on the `[INSTALL/REMOVE]` button.

Figure 4. 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.

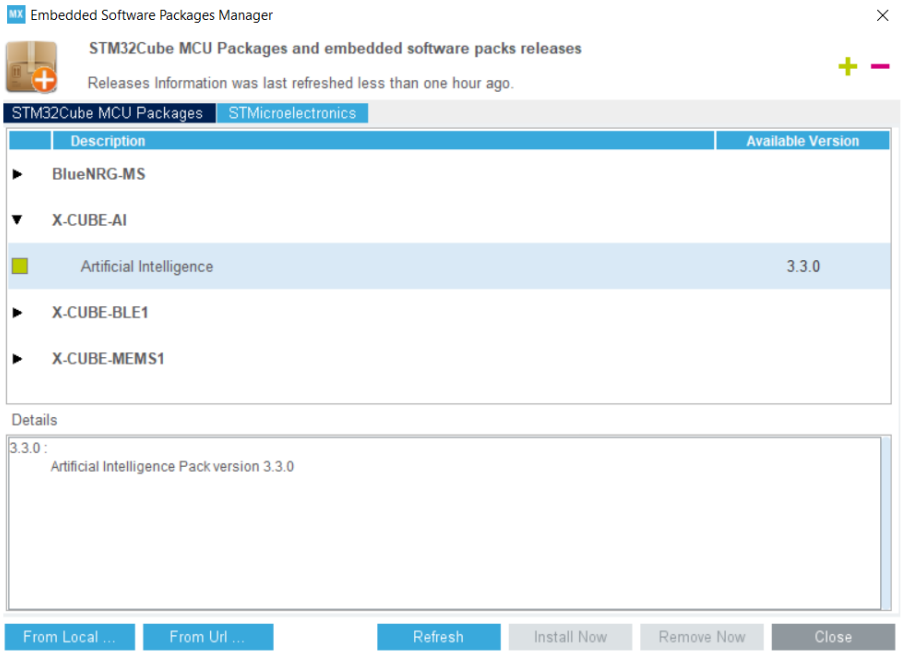
Figure 5. Installing X-CUBE-AI in STM32CubeMX



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 6. X-CUBE-AI in STM32CubeMX

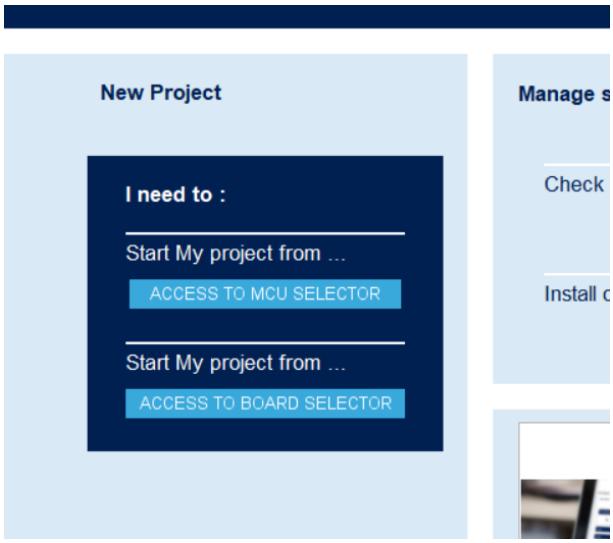


3 Starting a new STM32 AI project

3.1 MCU and board selector

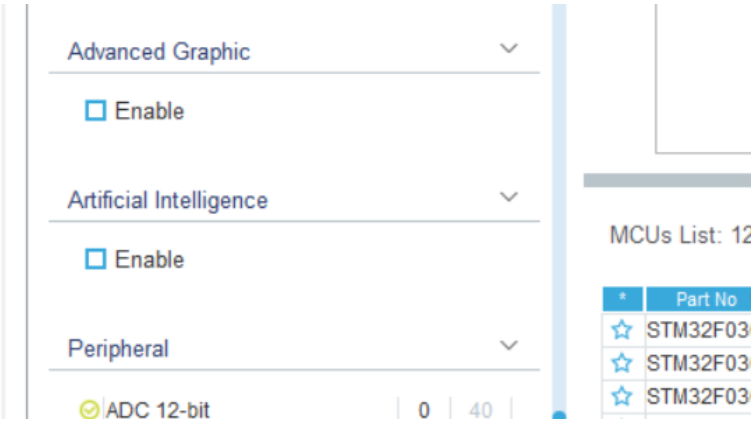
After launching the STM32CubeMX application, click on the [ACCESS TO MCU SELECTOR] or [ACCESS TO BOARD SELECTOR] button. Alternately, select [File] > [New Project...] or the CTRL-N shortcut.

Figure 7. 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 8.

Figure 8. AI filter



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

Figure 9 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 9. AI filter with default option

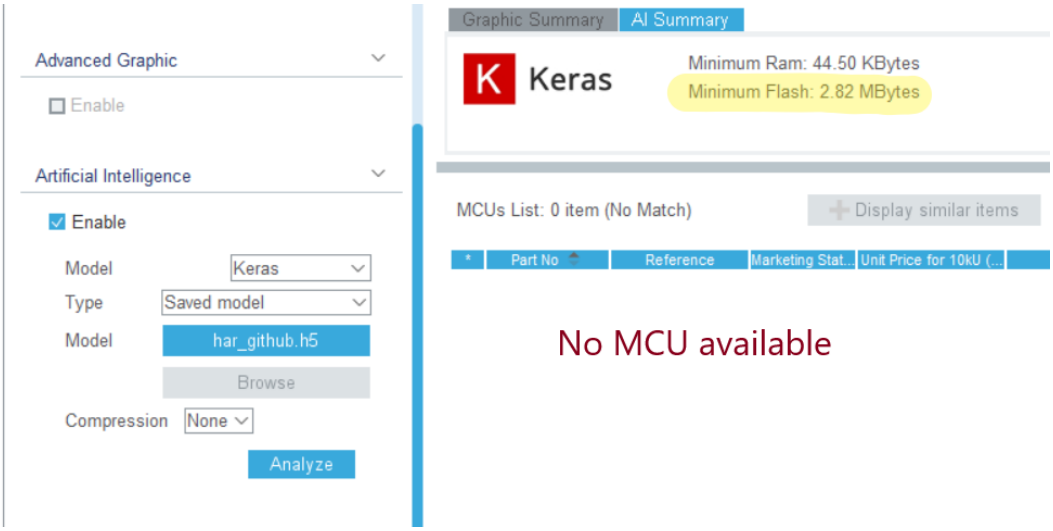
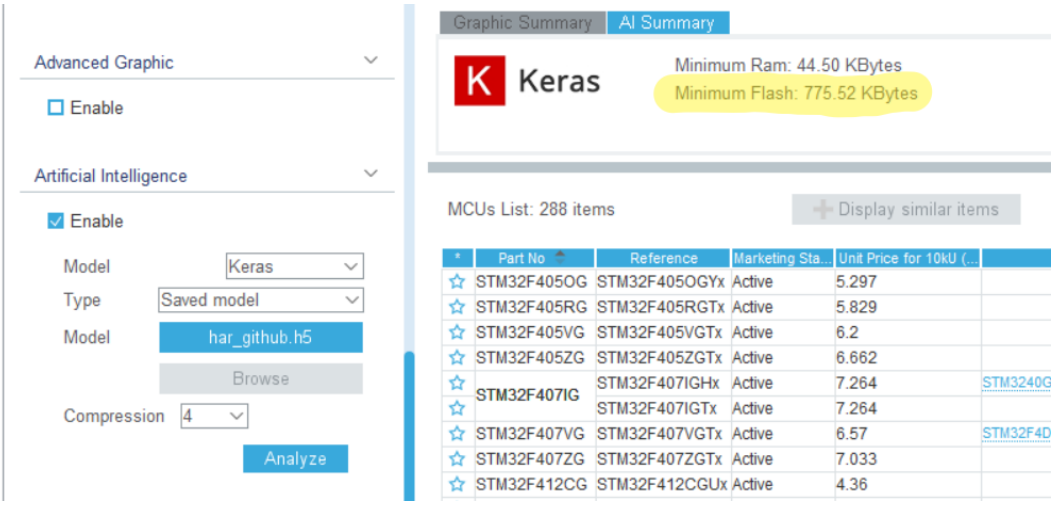


Figure 10 illustrates the case where a compression factor of 4 is applied.

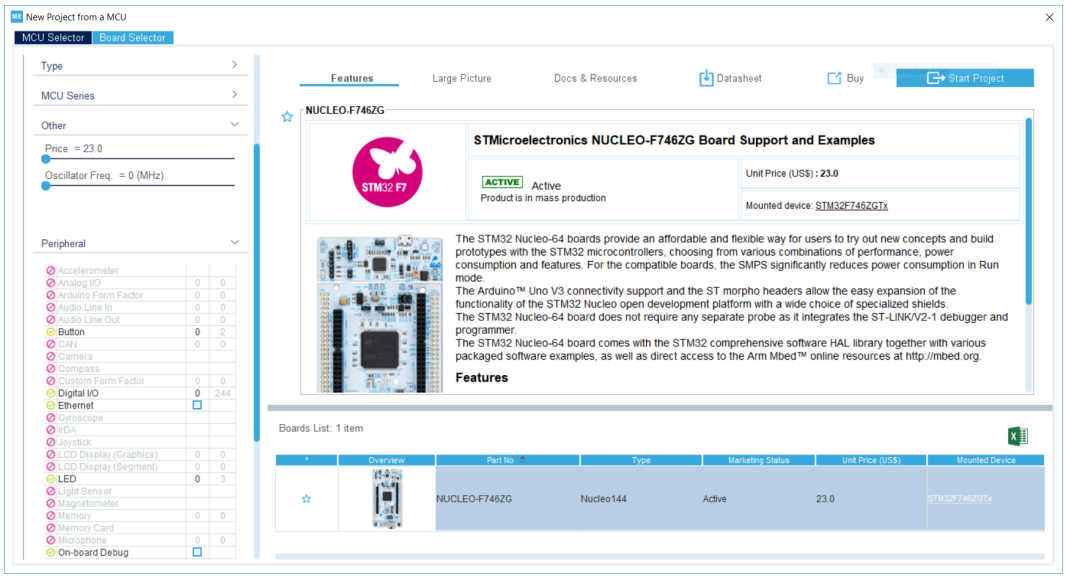
Figure 10. AI filter with compression x4



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.

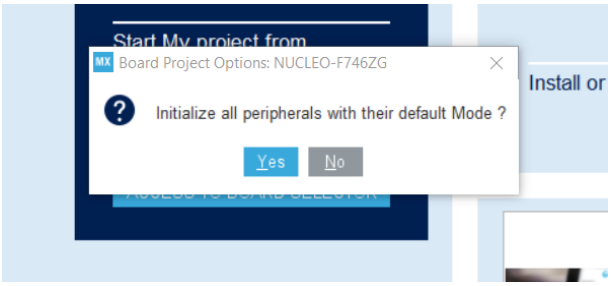
To continue, a NUCLEO-F746ZG development kit is selected as shown in Figure 11.

Figure 11. **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 12. **Initialize all peripherals**

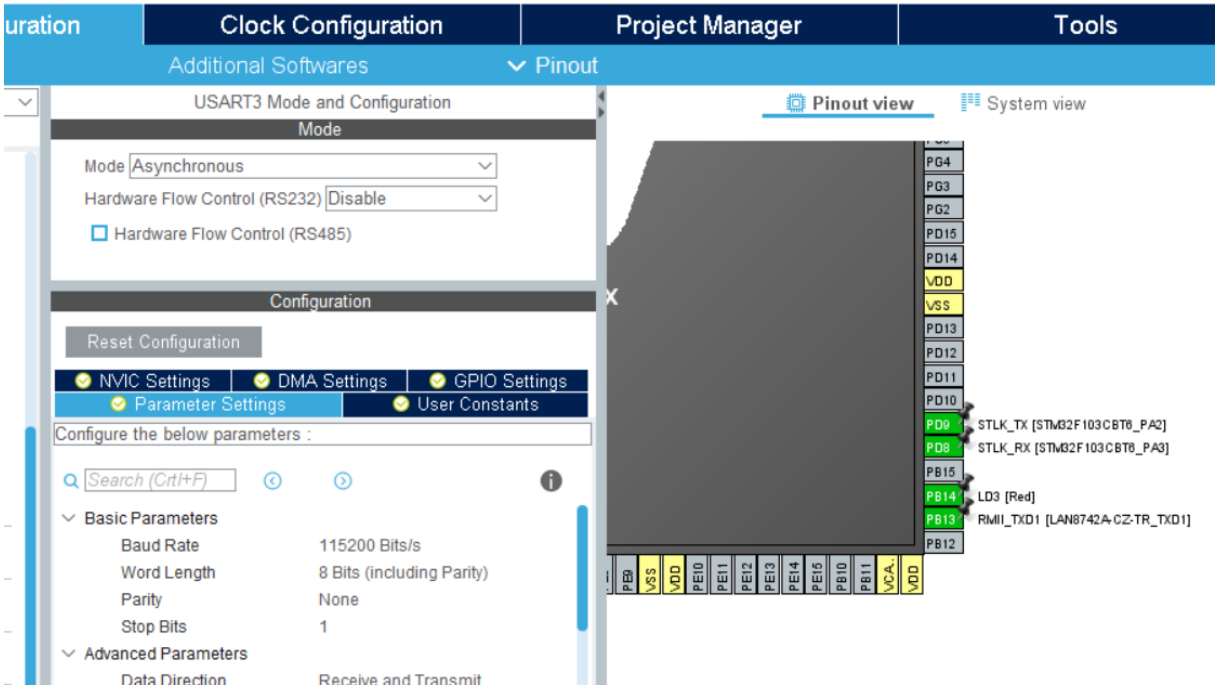


3.2 HW and SW platform settings

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 UART-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 IP to support the Virtual COM port (refer to Figure 13).

Figure 13. **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 15) to call the clock wizard to configure automatically the PLL IP (and associated clock tree). If the the clock wizard pop-up appears as shown in Figure 14, click on the `[OK]` button to continue.

Figure 14. **Clock wizard pop-up**

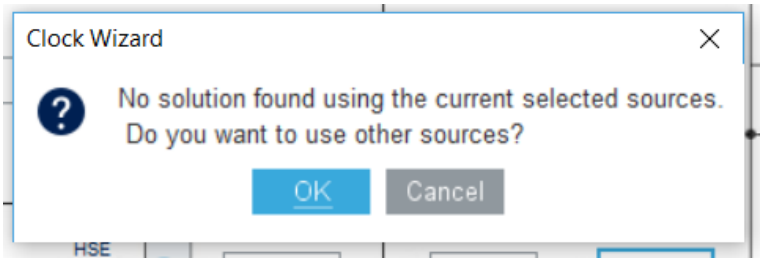
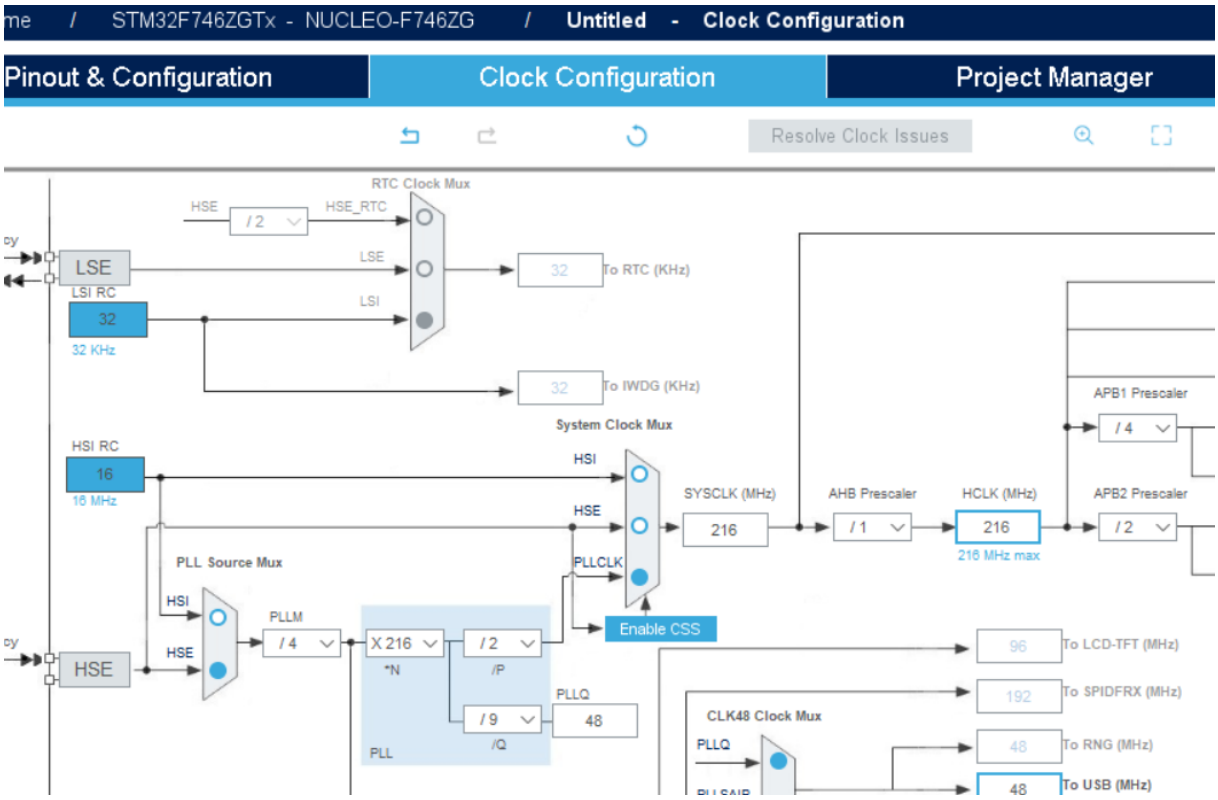


Figure 15. **System clock settings**

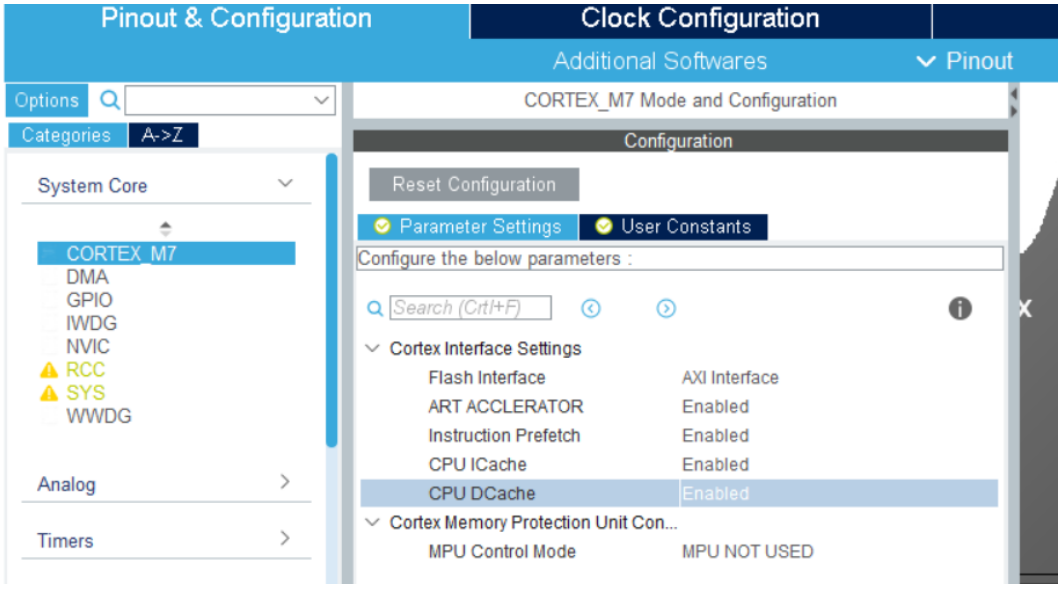


3.2.2 Set the MCU memory sub-system

- From the `[Pinout & Configuration]` tab (refer to Figure 16), 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 16. **MCU memory sub-system (parameter settings)**



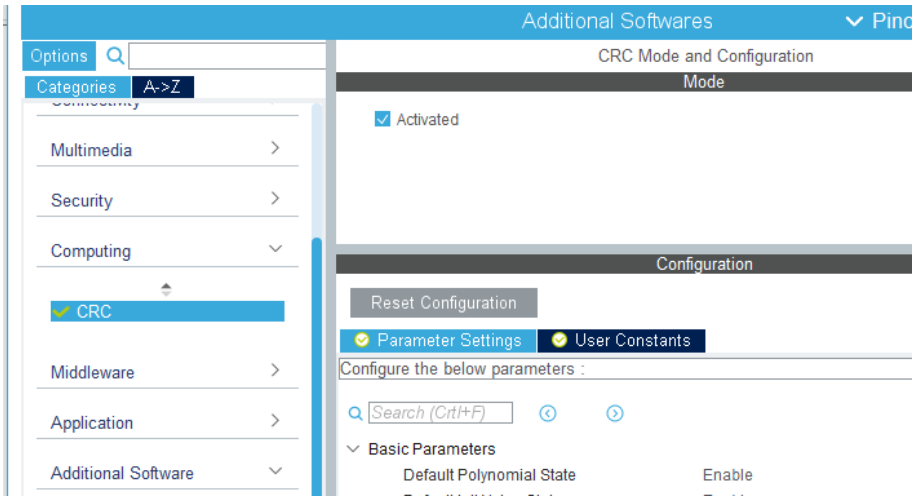
Note: Setting the maximum value for the MCU clock is not mandatory. It must be aligned with the configuration that is used in the final design. The setting of the wait-state for the Flash is automatically adjusted by the STM32CubeMX platform code generator.

3.2.3 CRC

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

Note: this IP is automatically enabled when the X-CUBE-AI component is added to the current project.

Figure 17. Enabling the CRC IP



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 18).

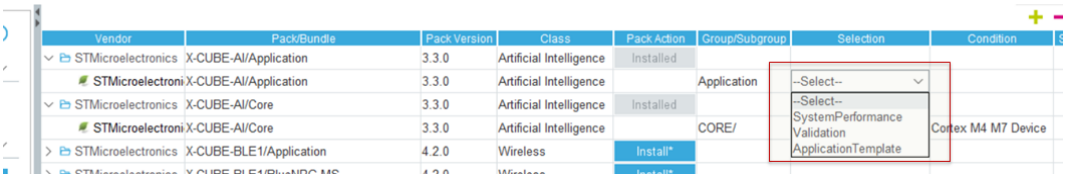
Figure 18. Additional software button

2. From the [Additional Software Component Selection] window, the X-CUBE-AI/core bundle (refer to Figure 19) must be checked to be able to upload the NN models and generate the associated STM32 NN library, In this case, the library is fully integrated as a static library, theonly needs to implement his AI-based application/middleware on top of the generated well-defined NN API [6].

Figure 19. Adding the X-CUBE-AI core component

3. Optionally, one of the add-on X-CUBE-AI applications (refer to Figure 20) 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 20. Add-on X-CUBE-AI applications



4. Click on [OK] to finalize the selection

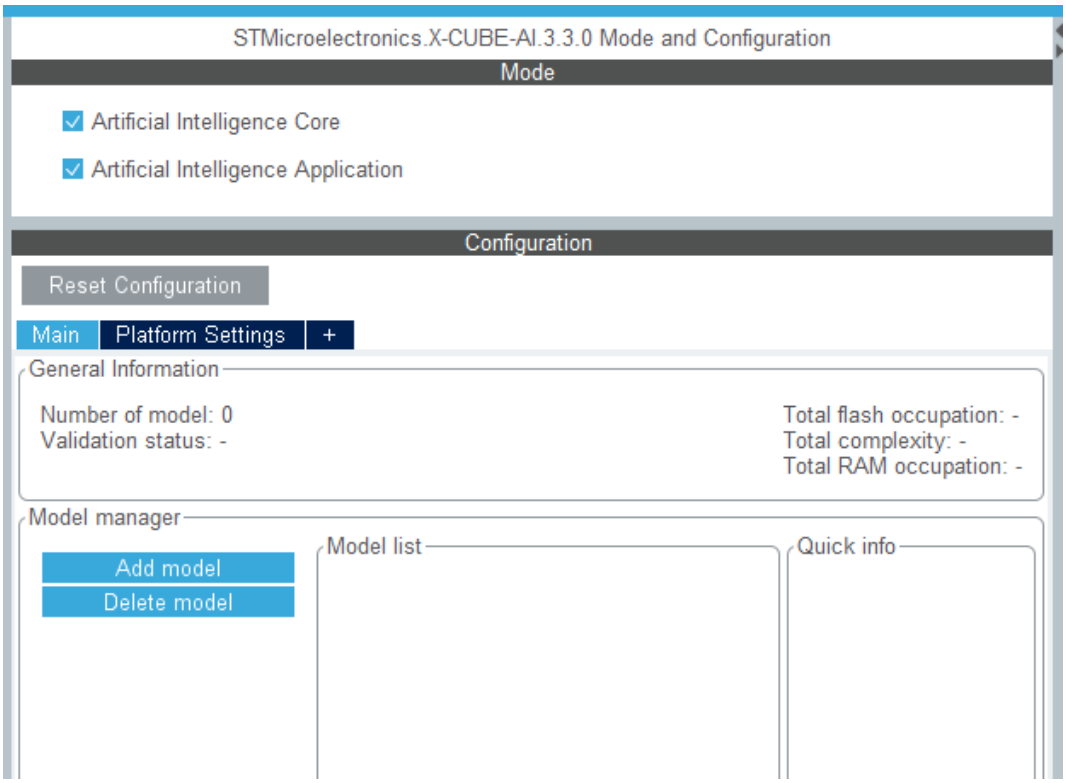
4.2 Enabling the X-CUBE-AI component

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 softwares. Click on [STMicroelectronics X-CUBE-AI 3.3.0] 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.

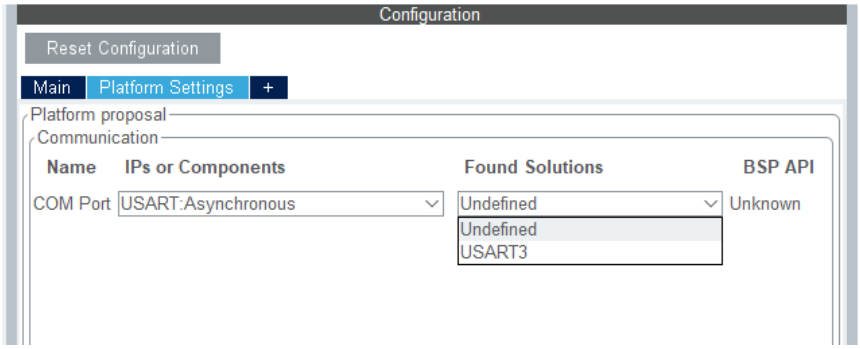
The AI application that is selected here corresponds to the application enabled during the previous step (refer to Figure 20).

Figure 21. 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 UART IP used to report the information (AI System Performance application) or communicate with the host (AI validation application).

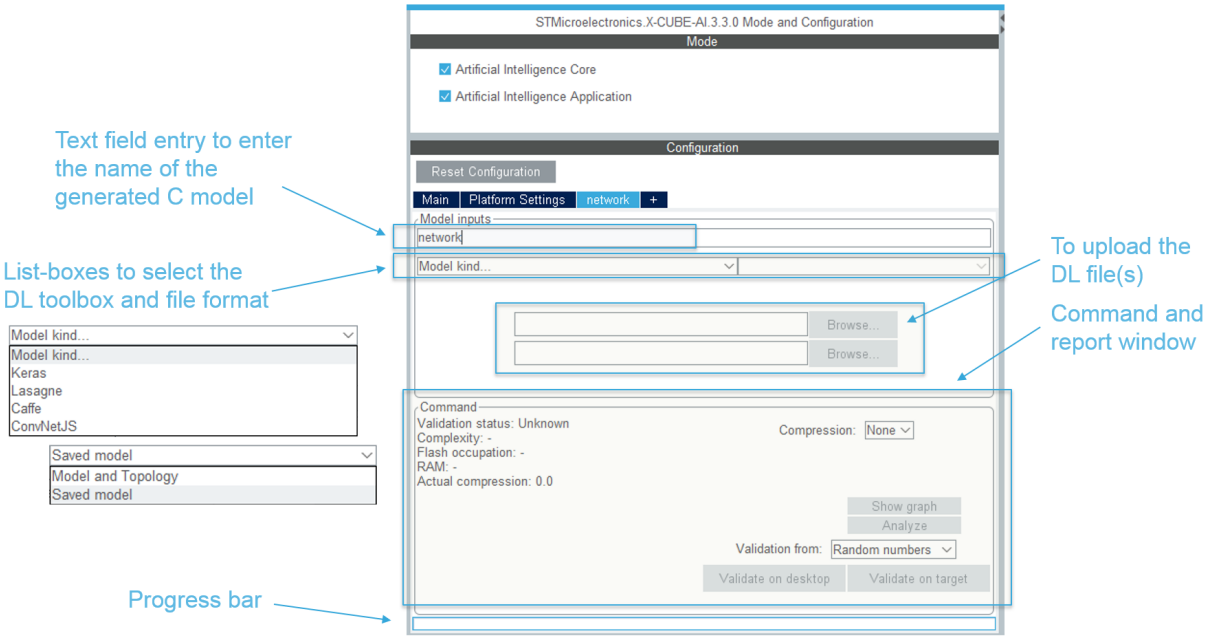
Figure 22. X-CUBE-AI platform setting panel



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 23. 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 [6]). 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 24. 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 4.4).

Figure 24. Insufficient RAM/Flash message box

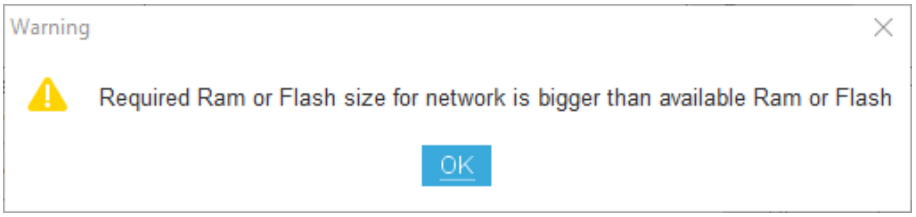
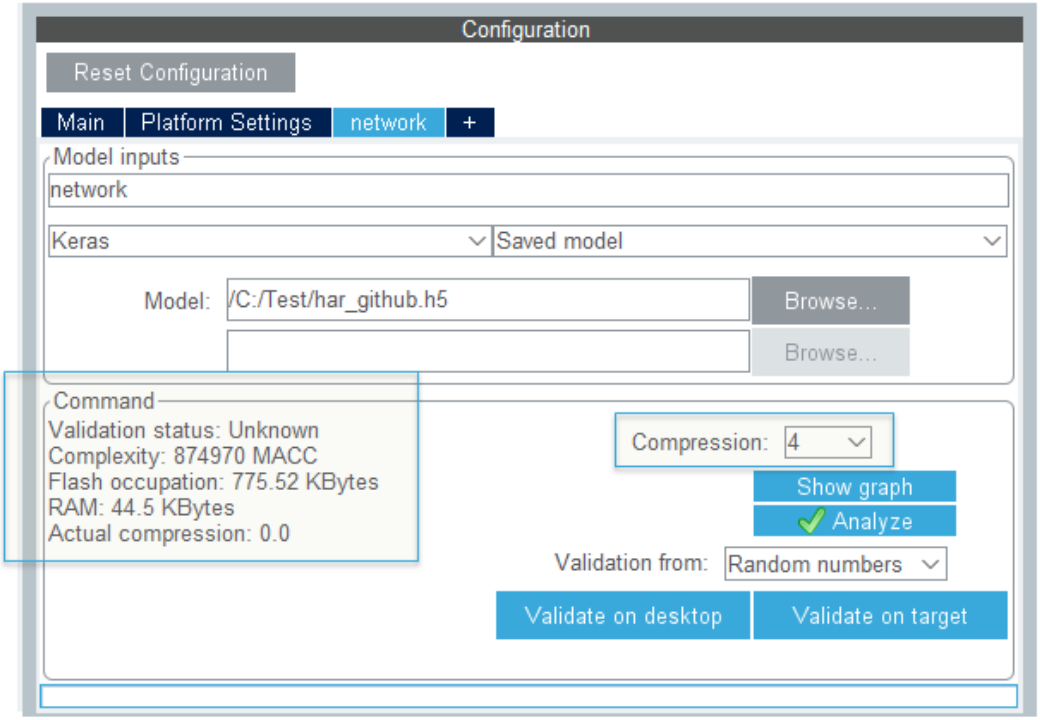


Figure 25. **Uploaded and analyzed DL model**



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

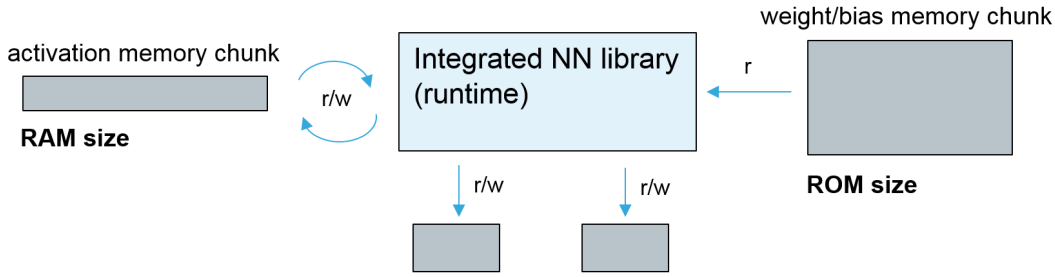
4.4 Dimensioning information report

When a DL model is processed, the dimensioning system informations presented in Table 2 and Figure 26 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 26. **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.

4.4.1 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):

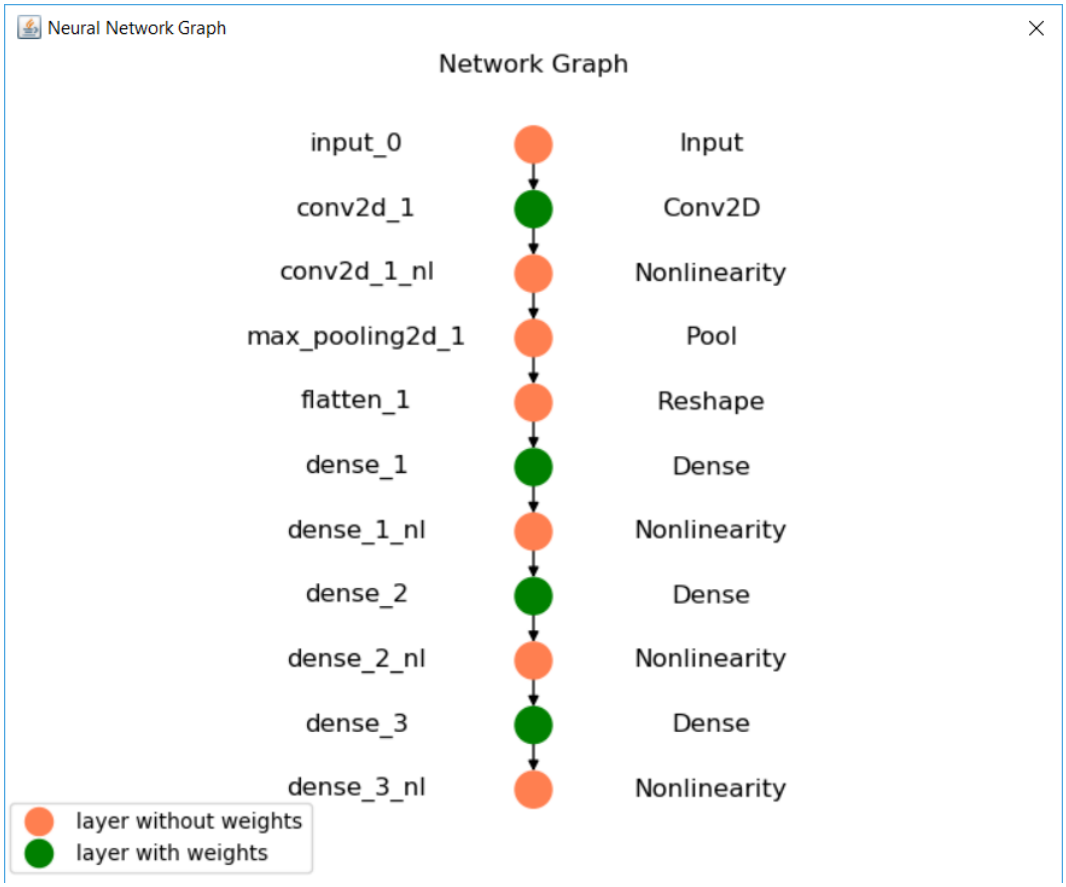
- STM32 Arm® Cortex®-M4: ~9 cycles/MACC
- STM32 Arm® Cortex®-M7:- ~6 cycles/MACC

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).

4.4.2 Generated C-model graph representation

Click on the `[Show graph]` button to show the main structural information of the uploaded DL model which are considered by the C-code generator (Figure 27).

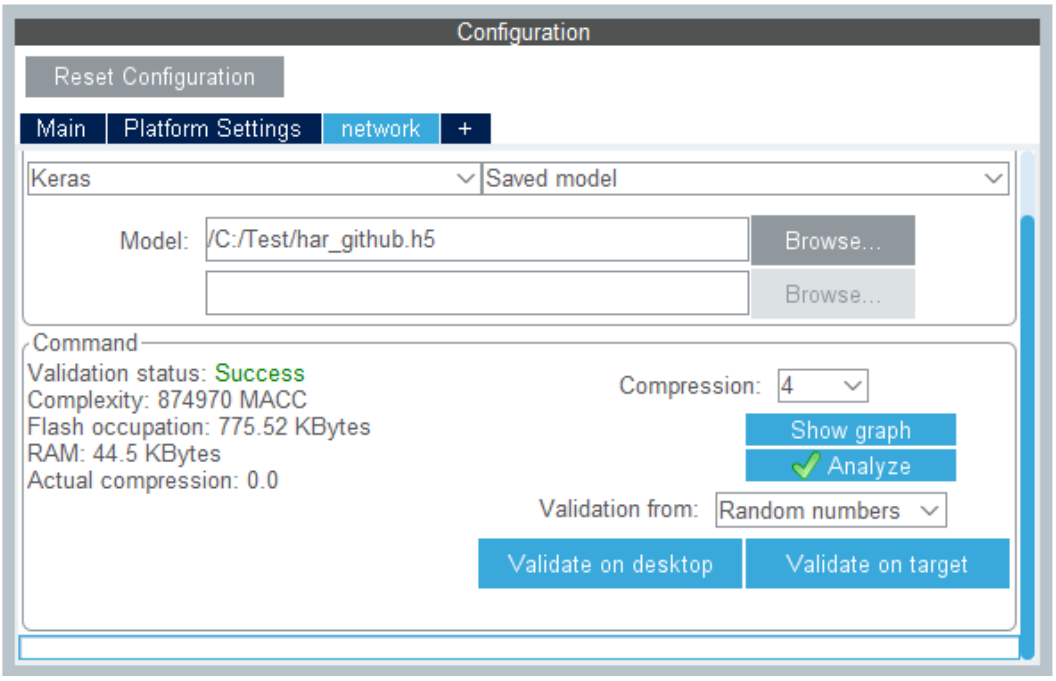
Figure 27. **Generated C-model graph**



4.5 Validating the generated C model

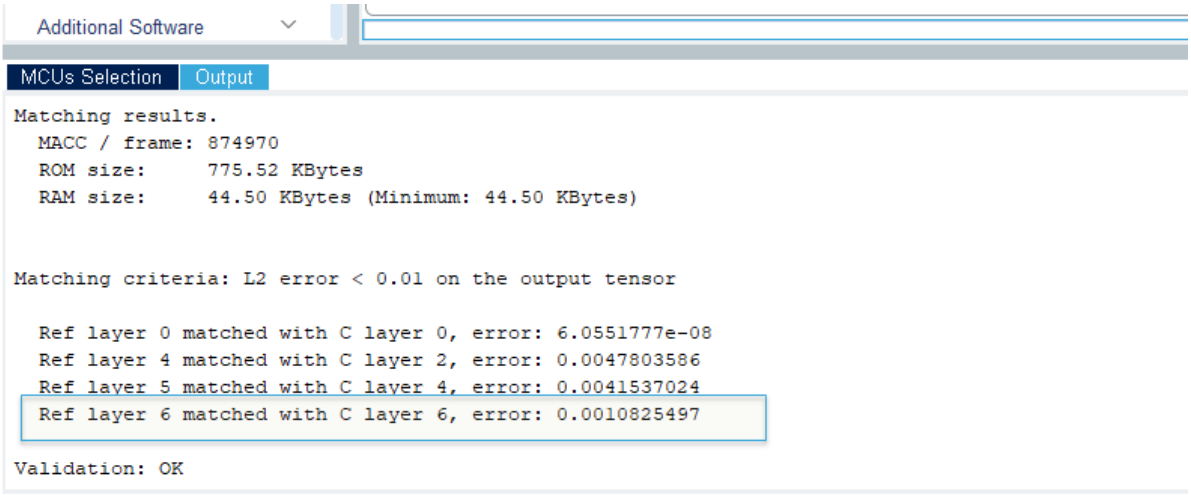
Click on the `[ Validate on desktop ]` button to launch a validation process of the generated C model. The `[Validation status]` field (refer to Figure 28) is updated according to the reported value of the L2 relative error (refer to Section 6.2 Validation engine). Note that this step is optional but preferable in particular when a compression factor is requested (refer to Section 6.1 Graph flow and memory layout optimizer).

Figure 28. **Validation status field**



More detailed information is reported in the UI log console (`[Outputs]` window) as shown in Figure 29. In particular the L2 error is also reported for each generated C layer.

Figure 29. **Validate on desktop - log report**



Note: Due to the optimization process, the report "Ref layer X matched with C layer Y, error: 0.0..." provides the link between the original layer of the DL model (X) and the optimized C-layer (Y). Refer to the "operation fusing" optimization in Section 6.1 Graph flow and memory layout optimizer).

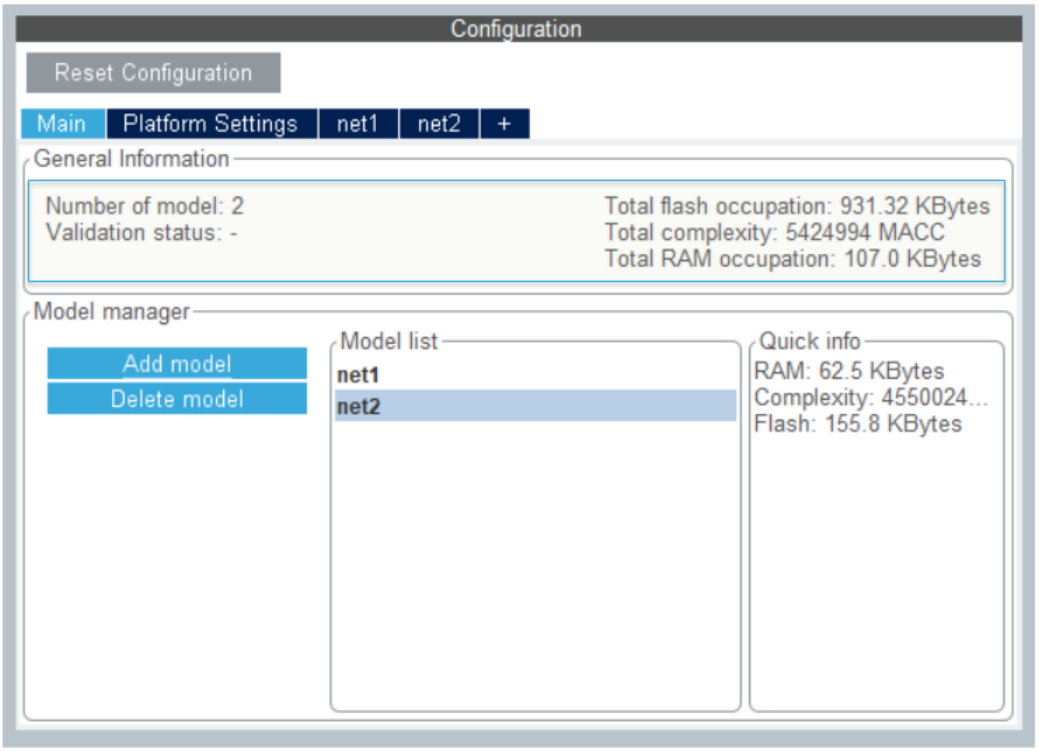
At this step, the uploaded DL model is ready to be integrated in the generated IDE project.

The `[Validate on target]` option must be used only later when the targeted device is flashed with 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.

4.6 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 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 occupations.

Figure 30. Main view with multiple networks



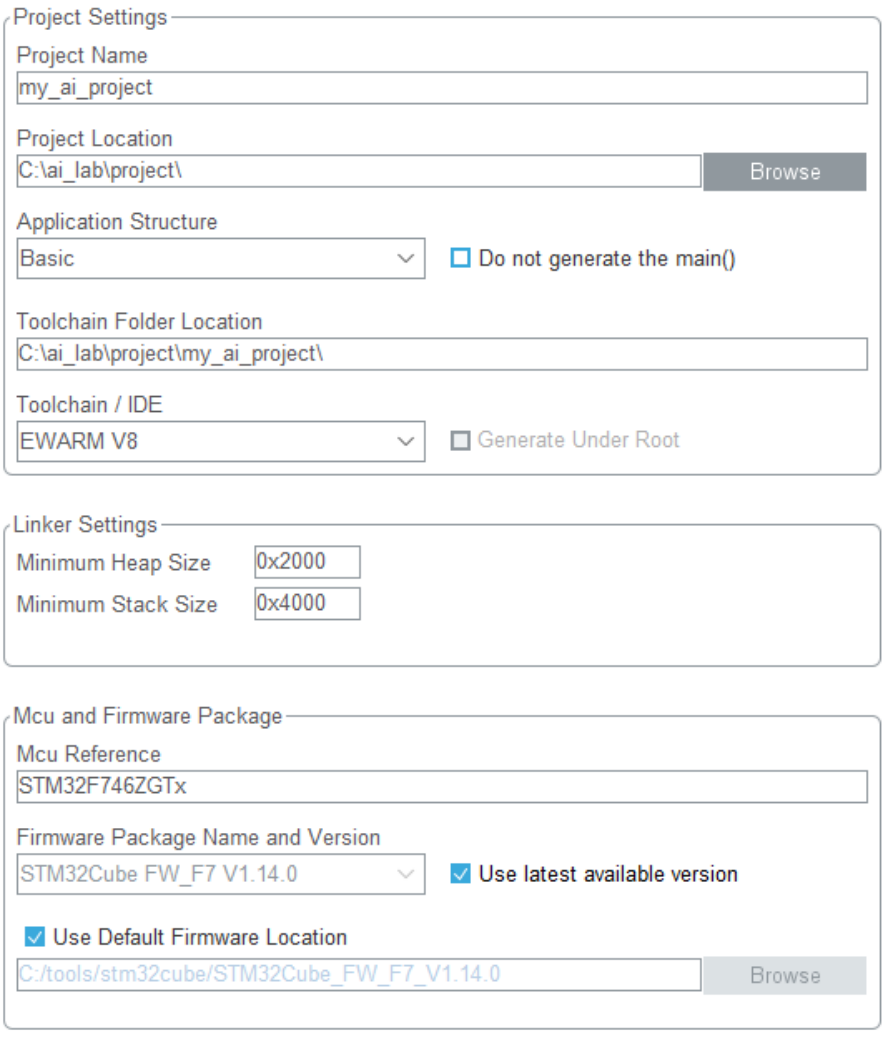
5 Generating, building and flashing

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™, TrueSTUDIO® for Atollic IDE, or others)
4. Update the minimum heap/stack size to minimize in a first time the possible overflow (minimum 2 Kbytes of heap is expected for the *"AI Validation"* test application)

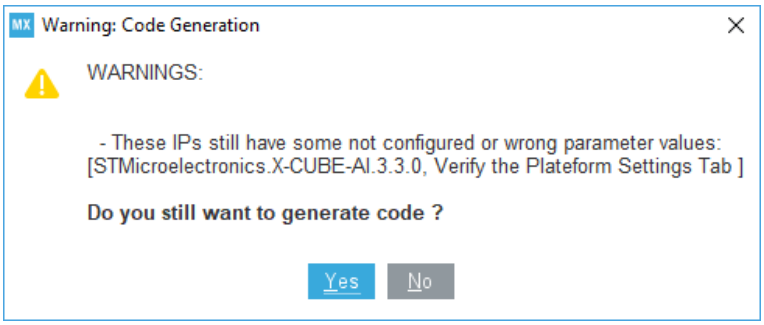
Figure 31. Project settings view for IDE Code generator



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 32 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 UART handle). Refer to Section 4.1 Adding the X-CUBE-AI componentfor details.

Figure 32. AI IP 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 IP, a middleware component, or both, or perform the `[Validation on target]` process.

5.2 Building and flashing

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.

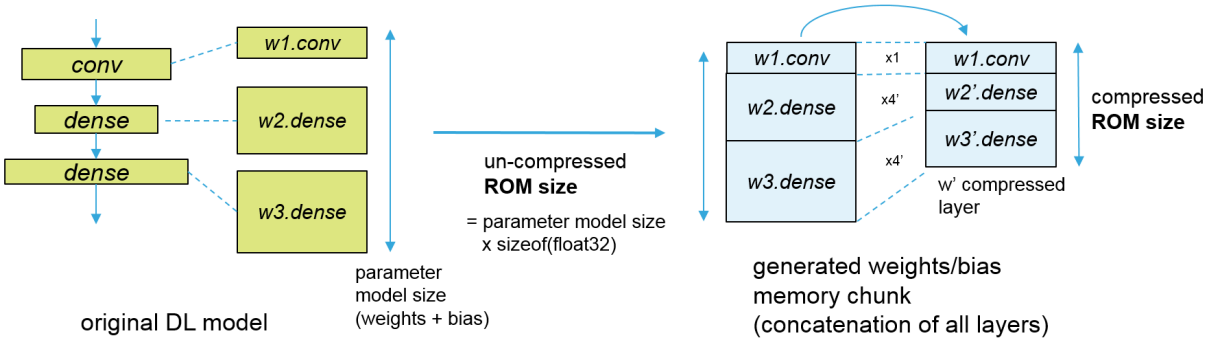
6 X-CUBE-AI internals

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 data-set 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.

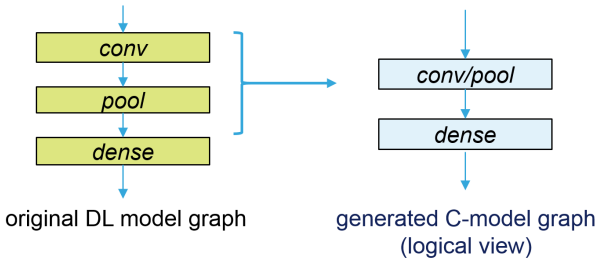
Figure 33. Weight/bias compression



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.

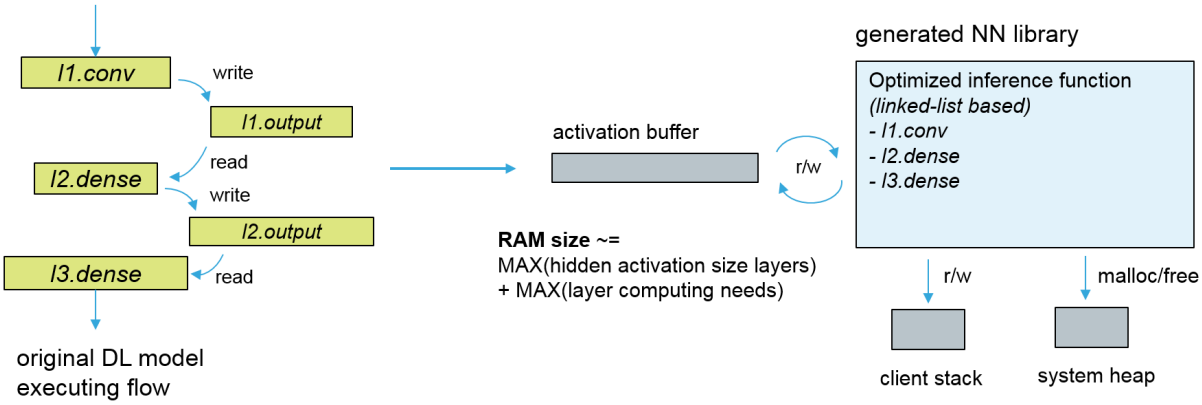
Figure 34. Operation fusing



- **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.

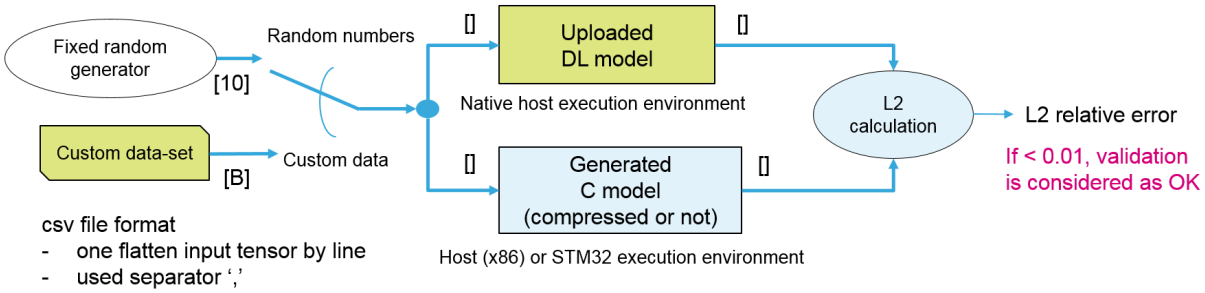
Figure 35. Optimal activation/working buffer



6.2 Validation engine

A simple and quick validation mechanism is provided to compare the accuracy of a generated C-model and uploaded DL model from a numerical standpoint (refer to Figure 36). Both models are fed with the same input tensors (fixed random inputs or custom data-set; refer to Section 14.2 Custom data set file format?). The L2 relative error is then calculated for all inferences. Only the L2 relative error of the output layer is used to indicate if the generated C-model is valid or not (the validity threshold is set to below 0.01). 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 data set and NN output tracing.

Figure 36. Validation flow overview



The L2 relative error is computed as shown in Figure 37 with:

- F_j: flattened arrays of the C code layer output j
- f_i: flattened arrays of the corresponding original layer output i

Figure 37. L2 computation

$$e_i = \frac{\|F_j - f_i\|}{\|F_j\|}$$

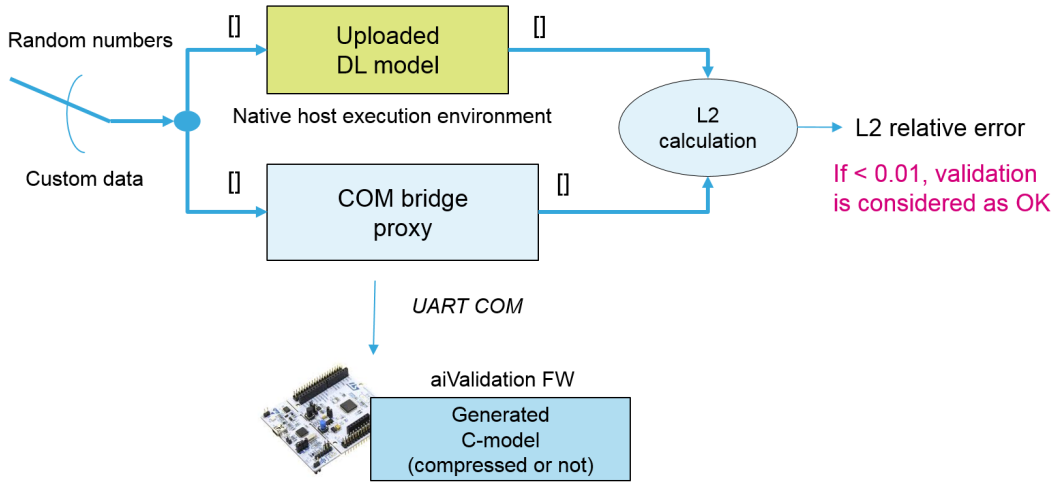
Two executing modes are provided:

- `[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 4.5 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 38. Validation on target



Note: All messages (including the data) exchanged with the target are stored in the " C:\Users\<username>\.stm32cubemx\ai_stm32_msg.log" dedicated log file.

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.

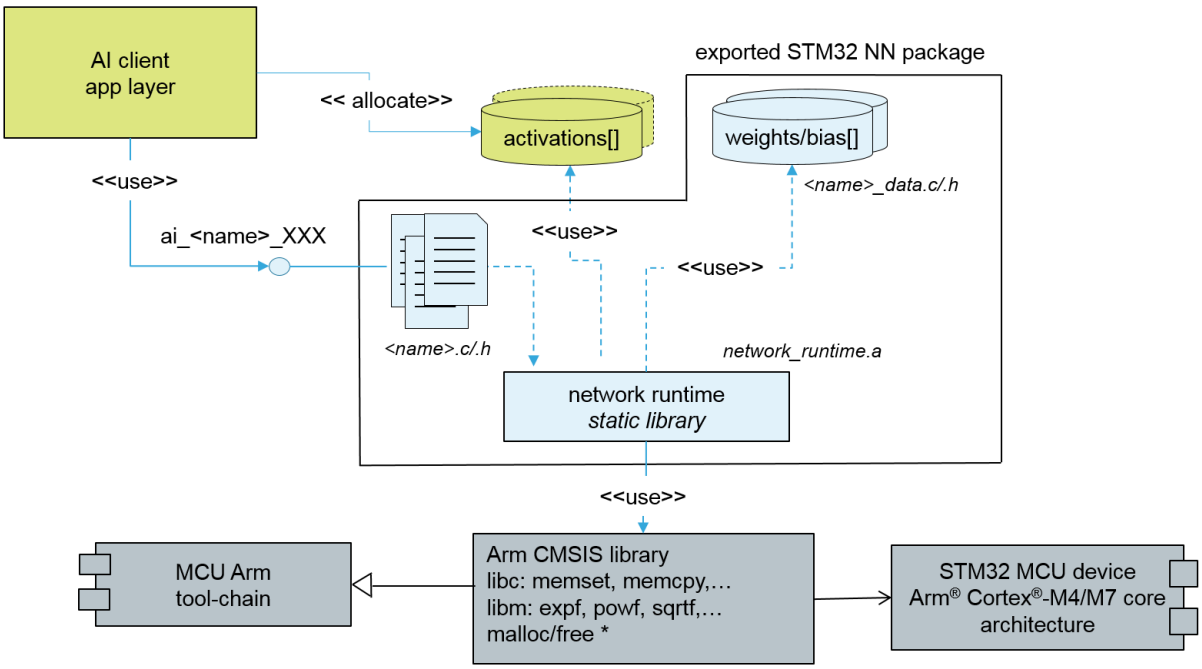
The *network_runtime.a* library or NN computing kernel library is provided as a static library:

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

7.1 Firmware integration

Figure 39 illustrates the MCU integration model and view (including run-time dependencies) of the generated STM32 NN package.

Figure 39. 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 generated NN objects are exported into sub-folder *<project_name>/Middlewares/ST/AI/*.

The *arm_dot_prod_f32.c* file from the CMSIS-DSP library is also added.

```
<project_name>
    |- Drivers\CMSIS\DSP_Lib\Source\BasicMathFunctions
    |                                     \_ arm_dot_prod_f32.c
    |- Inc
    |    |- bsp_ai.h                    /* generated files by STMCubeMX */
    |    |- constants_ai.h              /* for AI sample application */
    | ...
    |--Middlewares
        \-- ST/AI
            | /*=====================================================================*/
            |-- AI         /* generated NN library by the X-CUBE-AI core */
            |     |-- data
            |     |     |-- <name_1>_data.c     /* specialized NN weight/bias */
            |     |     |-- <name_1>_data.h
            |     |     |-- <name_2>_data.c
            |     |     \-- <name_2>_data.h
            |     |-- include
            |     |     |-- <name_1>.h          /* specialized public/client API */
            |     |     |-- <name_2>.h
            |     |     \- *.h                  /* generic and private header files */
            |     |-- lib
            |     |     \-- network_runtime.a   /* generic run-time library */
            |     \-- src
            |         |-- <name_1>.c            /* specialized NN implementation*/
            |         \-- <name_2>.c
            | /*=====================================================================*/
            |
            \-- Application
                    |-- SystemPerformance       /* generic sample application */
                        |-- Inc
                        | \-- aiSystemPerformance.h
                        \-- Src
                            \-- aiSystemPerformance.c
```

Note: Except for the" <name>.h" file, all header files from the "include" folder are the private header files ("network_runtime.a" library).

Note: "bsp_ai.h", "constants_ai.h", and "SystemPerformance." files are generated or copied only if the add-on AI application is added and enabled.*

The following specific system build options are automatically added by the STM32CubeMX generator in the IDE project files:

```
# Sub-makefile to build the AI-related files

# AI sub-folder location
NN_LIB ?= ./Middlewares/ST/AI
```

```
STM32_HAL_DIR ?= ./Drivers

# specialized AI sources
SRC += $(NN_LIB)/AI/data/*_data.c
SRC += $(NN_LIB)/AI/src/*.c

# C/C++ compiler options
CPPFLAGS += -I$(NN_LIB)/AI/Include
CPPFLAGS += -I$(NN_LIB)/AI/data

# to compile the "arm_dot_prod_f32.c" file
CPPFLAGS += -DARM_MATH_CM7 -D_FPU_PRESENT=1U
CPPFLAGS += -D$(STM32_HAL_DIR)/CMSIS/Include
SRC += $(STM32_HAL_DIR)/CMSIS/DSP_Lib/Source/BasicMathFunctions/arm_dot_prod_f32.c

# Linker options to the generic NN runtime library
LDFLAGS += -L$(NN_LIB)/AI/lib -l:network_runtime.a

# for the AI System Performance application
SRC += $(NN_LIB)/Application/SystemPerformance/Src/aiSystemPerformance.c
CPPFLAGS += -I$(NN_LIB)/Application/SystemPerformance/Inc
```

7.3 Specific AI platform files

For the AI applications, specific files are automatically generated by the STM32CubeMX code generator. They provide for the application code a simple way to be portable through different STM32 HW platforms and settings thanks to the STM32 HAL layer and associated well-defined macro definitions (refer for instance to the *aiSystemPerformance.c* and *aiSystemPerformance.h* files as example).

```
/* @file - bsp_ai.h - GENERATED CODE by STM32Cube MX */

...
#include "stm32f7xx.h"
#include "stm32f7xx_hal.h"
#include "app_x-cube-ai.h"
#include "constants_ai.h"
#define UartHandle huart3
#define MX_UARTx_Init MX_USART3_UART_Init
...


/* @file - app_x-cube-ai.c - GENERATED CODE by STM32Cube MX */

...
#include "bsp_ai.h"
#include "aiXXX.h"
...
void MX_X_CUBE_AI_Init(void)
{
    MX_UARTx_Init();
    aiXXXInit(); /* Entry point to initialize the application */
    /* USER CODE BEGIN 0 */
    /* USER CODE END 0 */
}

/*
* AI background task
*/
void MX_X_CUBE_AI_Process(void)
{
    aiXXXProcess(); /* Entry point to execute the core of application */
    HAL_Delay(1000); /* delay 1s */
    /* USER CODE BEGIN 1 */
    /* USER CODE END 1 */
}
...
...
```

7.4 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 Section 8 Embedded inference client API); 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_co
nfig);

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);
```

7.5 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.

7.6 Code and data placement considerations

For the current STM32 memory architecture (L4/F4/F3-based and F7/H7-based), no specific data or code placement is expected for performance reason. The Flash ART IP 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.

7.7 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

The embedded inference client API is part of the *<project_name>/Middlewares/ST/AI/src/<name>.h* file. All functions and macros are generated according to the provided C-network name.

8.1 Input and output x-D tensor layout

Input and output buffers are defined as a tensor with a maximum of 3 dimensions (HWC layout format, standing for Height, Width, and Channels). A batch dimension can be added to handle an array of tensors. The buffers are completely defined through the `struct ai_buffer` C structure definition.

```
/* @file: ai_platform.h */

typedef struct ai_buffer_ {
    ai_buffer_format    format;     /*!< buffer format */
    ai_u16        n_batches;  /*!< number of batches in the buffer */
    ai_u16        height;     /*!< buffer height dimension */
    ai_u16        width;      /*!< buffer width dimension */
    ai_u32        channels;   /*!< buffer number of channels */
    ai_handle          data;       /*!< pointer to buffer data */
} ai_buffer;
```

Note: Currently, only the `AI_BUFFER_FORMAT_FLOAT` format (32-bit floating-point) is supported for the input and output tensors.

This C structure can be also used to handle an opaque and specific data buffer.

8.1.1 1-D tensor

For a 1-D tensor, standard C-array type is expected to handle the input and output tensors.

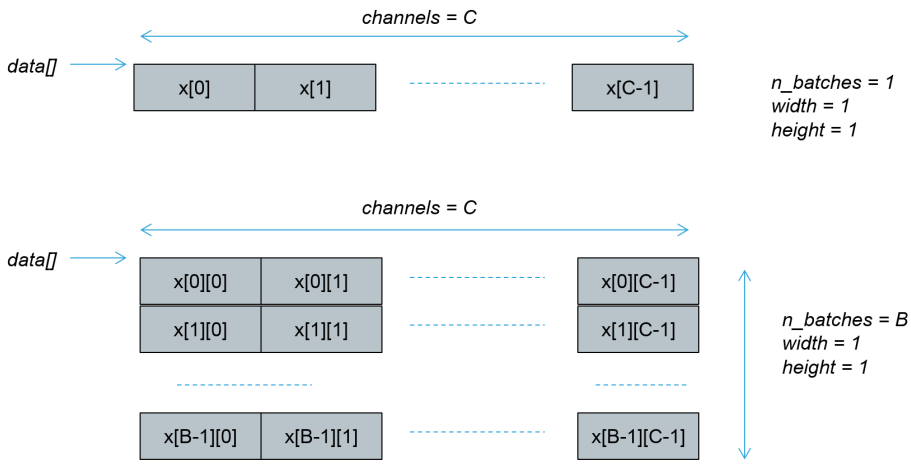
```
#include "network.h"

#define xx_SIZE VAL                 /* = H * W * C = C */

ai_float xx_data[xx_SIZE];          /* n_batch = 1, height = 1, width = 1, channels = C */

ai_float xx_data[B * xx_SIZE];      /* n_batch = B, height = 1, width = 1, channels = C */
ai_float xx_data[B][xx_SIZE];
```

Figure 40. 1-D Tensor data layout



8.1.2 2-D tensor

For a 2-D tensor, standard C-array-of-array memory arrangement is used to handle the input and output tensors. Two dimensions are mapped to each of two first dimensions of the tensor in the original toolbox representation: H and C in Keras / TensorFlow™, H and W in Lasagne.

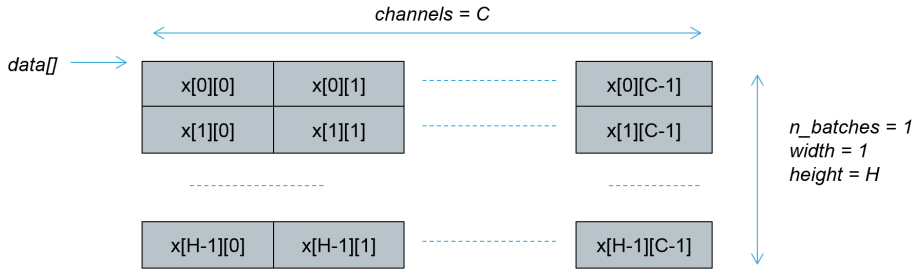
```
#include "network.h"

#define xx_SIZE VAL              /* = H * W * C = H * C */

ai_float xx_data[xx_SIZE];       /* n_batch = 1, height = H, width = 1, channels = C */
ai_float xx_data[H][C];

ai_float xx_data[B * xx_SIZE];   /* n_batch = B, height = H, width = 1, channels = C */
ai_float xx_data[B][H][C];
```

Figure 41. 2-D Tensor data layout



Note: If the dimension order in the original toolbox is different from HWC (such as Lasagne: CHW), it is the user's responsibility to properly re-arrange the elements.

8.1.3 3-D tensor

For a 3D-tensor, standard C-array-of-array-of-array memory arrangement is used to handle the input and output tensors.

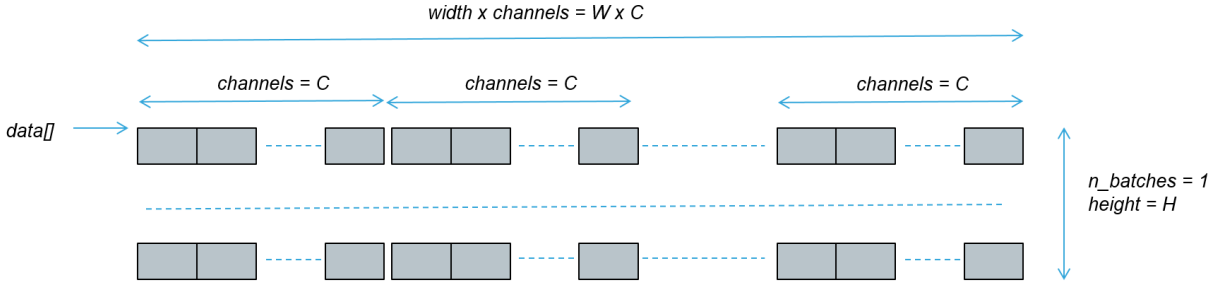
```
#include "network.h"

#define xx_SIZE VAL              /* = H * W * C */

ai_float xx_data[xx_SIZE];       /* n_batch = 1, height = H, width = W, channels = C */
ai_float xx_data[H][W][C];

ai_float xx_data[B * xx_SIZE];   /* n_batch = B, height = H, width = W, channels = C */
ai_float xx_data[B][H][W][C];
```

Figure 42. 3-D Tensor data layout



8.2 create() / destroy()

```
ai_error ai_<name>_create(ai_handle* network, const ai_buffer* network_config);
ai_handle ai_<name>_destroy(ai_handle network);
```

This mandatory function is the early function that must be called by the AI client to instantiate and initialize an NN instance. `ai_handle` references an opaque context that must be passed to the other functions.

- The `<name>_config` parameter is a specific network configuration buffer coded as an `ai_buffer`. It is generated (`AI_<NAME>_DATA_CONFIG` C-define) by the AI code generator (see the *<name>.h* file)
- When the instance is no more used by the application, the `ai_<name>_destroy()` function must be called to release the allocated resources

Typical usage

```
#include <stdio.h>
#include "network.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;
...
int aiInit(void) {
    ai_error err;
    ...
    err = ai_network_create(&network, AI_NETWORK_DATA_CONFIG);
    if (err.type != AI_ERROR_NONE) {
        /* manage the error */
        printf("E: AI error - type=%d code=%d\r\n", err.type, err.code);
        ...
    }
    ...
}
```

8.3 get_error()

```
ai_error ai_<name>_get_error(ai_handle network);
```

This function can be used by the client to retrieve the first error reported during the execution of an `ai_<name>_xxx()` function. Refer to the *ai_platform.h* file to have the list of the returned error types (`ai_error_type`) and associated codes (`ai_error_code`) .

Typical AI error function handler (for debug and log purpose)

```
#include "network.h"
...
void aiLogErr(const ai_error err, const char *fct)
{
    if (fct)
        printf("E: AI error (%s) - type=%d code=%d\r\n", fct, err.type, err.code);
    else
        printf("E: AI error - type=%d code=%d\r\n", err.type, err.code);
}
```

8.4 get_info()

```
ai_bool ai_<name>_get_info(ai_handle network, ai_network_report* report);
```

This optional function can be used by the client to retrieve the informations of the instantiated NN (for debug and log purpose).

The `network` handle must be a valid handle. Refer to the `ai_<name>_create()` function.

Typical usage

```
#include "network.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;
...
int aiInit(void) {
    ai_network_report report;
    ai_bool res;
...
    res = ai_network_get_info(network, &report)
    if (res) {
        /* display/use the reported data */
        ...
    }
...
}
```

8.5 init()

```
ai_bool ai_<name>_init(ai_handle network, const ai_network_params* params);
```

This mandatory function must be used by the client to initialize the internal run-time structure of the instantiated NN:

- The `params` parameter is a structure (`ai_network_params` type) that passes the references of the generated weights (`params` attribute), and the activation/crash memory buffer (`activations` attribute).
- The `network` handle must be a valid handle. Refer to the `ai_<name>_create()` function.

```
/* @file: ai_platform.h */
typedef struct ai_network_params_ {
  ai_buffer params;           /*! info about params buffer(required!) */
  ai_buffer activations;      /*! info about activations buffer (required!) */
} ai_network_params;
```

Multiple C-macro helpers and specific `AI_<NAME>_XX` C defines must be used to initialize this parameter:

- The `params` attribute handles the weights/bias memory buffer
- The `activations` attribute handles the activation/crash memory buffer, which is used by the forward process (refer to the `ai_<name>_run()` function)
- The sizes of associated memory blocks are respectively defined by the following C defines (refer to file *<name>_data.h*). The memory layouts of these buffers depend on the Neural Network implemented.
  - `AI_<NAME>_DATA_WEIGHTS_SIZE`
  - `AI_<NAME>_DATA_ACTIVATIONS_SIZE`

Typical usage

```
#include <stdio.h>
#include "network.h"
#include "network_data.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;

/* Global buffer to handle the activations data buffer - R/W data */
AI_ALIGNED(4)
static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];
...

int aiInit(void) {
    ai_error err;
```

```
...
    /* initialize network */
    const ai_network_params params = {
        AI_NETWORK_DATA_WEIGHTS(ai_network_data_weights_get()),
        AI_NETWORK_DATA_ACTIVATIONS(activations) };
    if (!ai_network_init(network, &params)) {
        err = ai_network_get_error(network);
        /* manage the error */
        ...
        }
...
}
```

8.6 run()

```
ai_i32 ai_<name>_run(ai_handle network, const ai_buffer* input, ai_buffer* output);
```

This function is the main function to feed the NN. The input and output buffer parameters (`ai_buffer` type) provide the input tensors and store the predicted output tensors respectively (refer to Section 8.1 Input and output x-D tensor layout):

- The returned value is the number of the input tensors processed (when `n_batches` > 1). If the returned value is negative or null, use the `ai_<name>_get_error()` function to know the error.
- `AI_<NAME>_IN_1` (respectively `AI_<NAME>_OUT_1`) must be used to initialize the input (respectively output) buffer handle
- `AI_<NAME>_IN_1_SIZE` (respectively `AI_<NAME>_OUT_1_SIZE`) is used to initialize the input data (respectively output data) buffers

Note: Two separate lists of input and output `ai_buffer` can be passed. This allows the future support of neural network with multiple inputs, outputs or both. `AI_<NAME>_IN_NUM` and `AI_<NAME>_OUT_NUM` respectively are used to know at compile-time the number of inputs and outputs. These values are also returned by the `ai_network_report` structure (refer to the `ai_<name>_get_info()` function).

Typical usage

```
#include <stdio.h>
#include "network.h"
...
/* Global handle to reference the instantiated NN */
static ai_handle network = AI_HANDLE_NULL;
...
static ai_buffer ai_input[AI_NETWORK_IN_NUM] = { AI_NETWORK_IN_1 };
static ai_buffer ai_output[AI_NETWORK_OUT_NUM] = { AI_NETWORK_OUT_1 };
...
int aiRun(const ai_float *in_data, ai_float *out_data,
          const ai_u16 batch_size)
{
    ai_i32 nbatch;
    ...
    /* initialize input/output buffer handlers */
    ai_input[0].n_batches = batch_size;
    ai_input[0].data = AI_HANDLE_PTR(in_data);
    ai_output[0].n_batches = batch_size;
    ai_output[0].data = AI_HANDLE_PTR(out_data);

    nbatch = ai_network_run(network, &ai_input[0], &ai_output[0]);
    if (nbatch != batch_size) {
        err = ai_network_get_error(network);
        /* manage the error */
        ...
        }
    ...
}
```

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 UART (refer to Section 3.2 HW and SW 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/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 43 and Figure 44 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 43. System run-time information - Keil® IDE

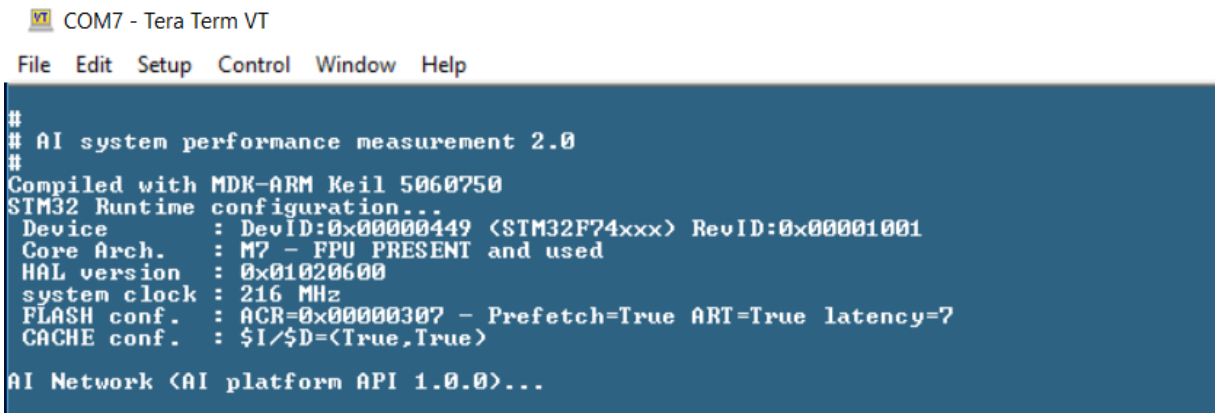
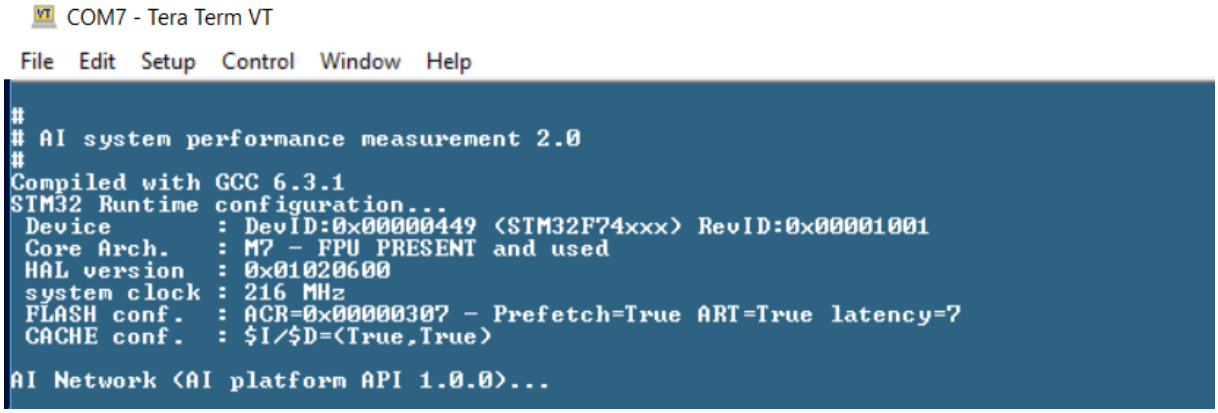


Figure 44. **System run-time information - Atollic IDE**



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

9.2 Embedded C-model network information

This second part shown in Figure 45 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 45. **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/R]` in the console during the execution of the main loop.

9.3 Embedded C-model run-time performance

As illustrated in Figure 46 and Figure 47, 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 46. 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 47. 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`.

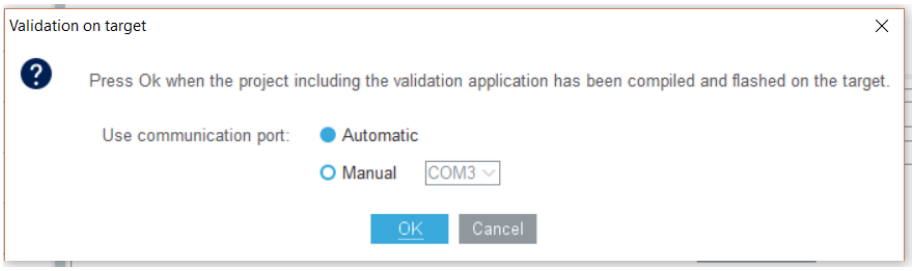
Caution: Today, the heap monitor is only supported for a GCC-based environment.

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 flexible and complete UART-based interface with the host to export a complete inference API. This application must be used through the X-CUBE-AI UI.

1. When the board is flashed, reset or restart it.
2. Reopen 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 48. Otherwise, all available COM ports are discovered to detect a valid connected STM32 board (the first board found is used).

Figure 48. Host COM port selector for validation on device

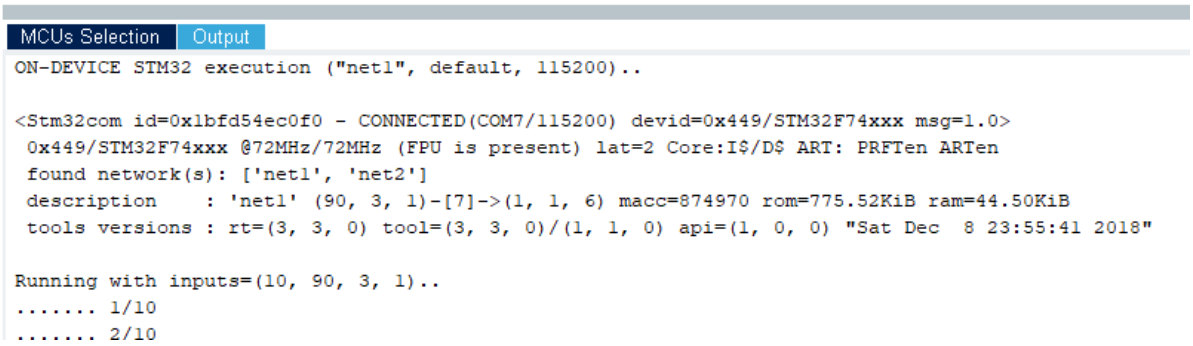


As for `[Validation on desktop]`, the final result is reported in the `[Validation status]` field. More detailed information is reported in the UI log console (`[Outputs]` window).

10.1 System run-time information

The first part of the reported log shown in Figure 49 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 49. System run-time information



10.2 Embedded C-model run-time performance

The second part of the log shown in Figure 50 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 50. 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 51 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 51. 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 52 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 52. 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

Only the following UART setting is supported:

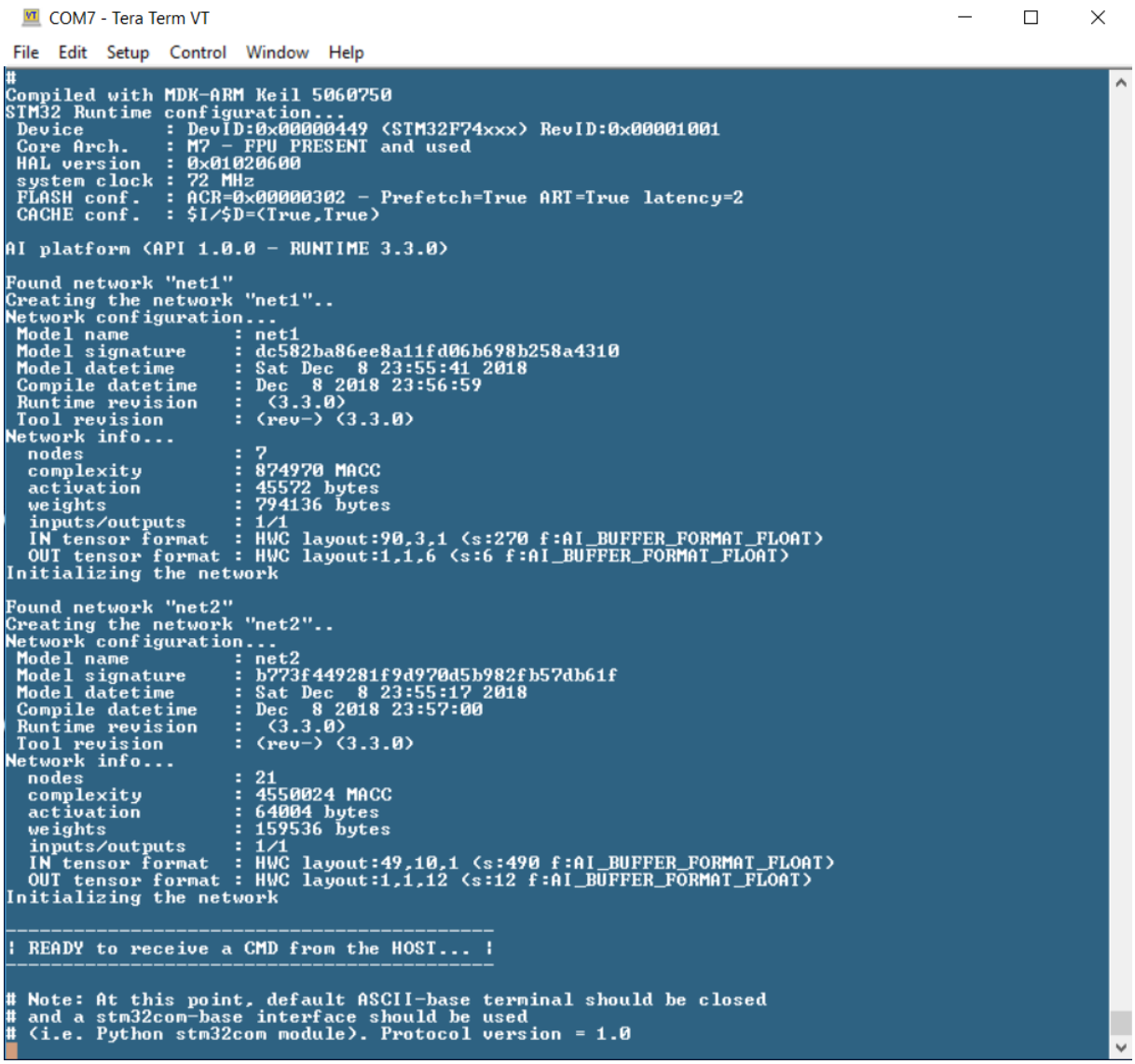
- 8 bits
- 1 stop bit
- No parity
- 115200 bauds (default value).
  If redefined, the new value is used (refer to Section 3.2 HW and SW platform settings).

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 flashed, 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 `[Valid on target]` process again.

Figure 53. 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 (refer to Section 7.3 Specific AI platform files) are generated to provide a basic framework with a multi-network API support and generic UART handle. This can be used as started point to develop an initial bare-metal application with two simple entry points (`init` and `process` functions).

12 Supported toolboxes and layers for Deep Learning

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

- Keras: //keras.io/
- Lasagne: //lasagne.readthedocs.io/en/latest/
- Caffe: //caffe.berkeleyvision.org/
- ConvNetJs: //cs.stanford.edu/people/karpathy/convnetjs/

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. The following sections describe in detail the configurations supported by the current version of the tool for each toolbox in terms of layers supported (using the toolbox naming conventions) and the attributes (parameters) supported for each layer. Same functionalities supported by multiple layers are listed together.

12.1 Keras

In Keras, X-CUBE-AI supports the TensorFlow™ backend with channels-last dimension ordering. Keras 2.0 up to version 2.2.4 is supported, while networks defined in Keras 1.x are not officially supported.

A model can be loaded in two different ways:

- From a single file with both model and weights (*.h5*, *.hdf5*)
- From the model configuration and weights in separate files. In this case, the weights are loaded from an HDF5 file (*.h5*, *.hdf5*) and the model configuration is loaded from a text file, either JSON (*.json*) or YAML (*.yml*, *.yaml*).

The following layers and attributes are supported:

- **Conv2D**, **Conv1D**: convolutional layers; 1D convolutions are supported by adding a singleton dimension on *x*. The following attributes are supported:
  - *padding*: *"SAME"* and *"VALID"* strategies
  - *kernel_size*: arbitrary filter kernel sizes, provided that they are smaller than the input size
  - *stride*: arbitrary strides, provided that they are smaller than the input size
  - *filters*: number of output channels
  - *use_bias*: do not use bias if set to *"False"*
- **DepthwiseConv2D**: depthwise convolutional layer; supports the same attributes as 2D convolution (*Conv2D*) with the following additional attribute:
  - *depth_multiplier*: number of channels per feature group; the number of output groups and channels is automatically inferred.
- **SeparableConv2D**: concatenation of a depthwise convolution without bias and a 1x1 convolution. The attributes of *DepthwiseConv2D* and *Conv2D* are supported.
- **MaxPooling1D**, **MaxPooling2D**: maximum pooling layer; the 1D versions are supported by adding a singleton dimension on *x*. The following attributes are supported:
  - *padding*: *"SAME"* and *"VALID"* strategies
  - *pool_size*: arbitrary pool sizes, provided they are smaller than the input size
  - *strides*: arbitrary strides, provided they are smaller than the input size
- **AveragePooling1D**, **AveragePooling2D**: average pooling layers; same attributes as the maximum pooling layers.
- **GlobalMaxPooling1D**, **GlobalMaxPooling2D**, **GlobalAveragePooling1D**, **GlobalAveragePooling2D**: global maximum and average pooling layers; supported by setting *pool_size* and *strides* to the input size of the layer.
- **Dense**: dense (fully connected) layer. The following attributes are supported:
  - *units*: number of output features
  - *use_bias*: do not use bias if set to *"False"*

- **Activation**: nonlinear activation layer, decoded also when part of *Conv2D*, *DepthwiseConv2D*, *SeparableConv2D*, or *Dense*. The following attributes are supported:
  - *nonlinearity*: type of nonlinear activation; the following functions are supported: `linear`, `relu`, `relu6`, `softmax`, `tanh`, `sigmoid`, and `hard_sigmoid`
- **BatchNormalization**: batch normalization layer. The following attribute is supported:
  - *axis*: input dimension on which the normalization is performed. Only normalization on the last axis (channels) is supported.
- **Permute**: dimension permutation layer. Only 3D tensors are supported. The following attribute is supported:
  - *dims*: target dimension order given the input dimension order, only for non-batch dimensions.
- **Flatten**: flattens the non-batch input dimensions to a vector; mapped as a special *Reshape* layer.
- **Reshape**: changes the shape of the input tensor without changing the number of elements. The following attribute is supported:
  - *shape*: target shape. Infer (-1) and keep same (0) values are also supported.
- **ZeroPadding1D**, **ZeroPadding2D**: padding layer, supported only as part of a subsequent *Conv1D* or *Conv2D* layer. The following attribute is supported:
  - *padding*: list of padding values, symmetric and asymmetric padding supported
- **LSTM**: Long-Short Term Memory layer. The following attributes are supported:
  - *units*: number of cell / hidden units
  - *activation*: hidden-to-output activation, the values from *Activation* are supported
  - *recurrent_activation*: hidden-to-hidden activation; the following functions are supported: `linear`, `relu`, `tanh`, `sigmoid`, and `hard_sigmoid`
  - *return_sequences*: if *"True"*, returns all the hidden states instead of the last one only
  - *use_bias*: do not use bias if set to *"False"*
- **GRU**: Gated Recurrent Unit layer; supports all the *LSTM* attributes and additionally supports:
  - *reset_after*: *GRU* only; uses a slightly different formula to align to the *CuDNN* implementation. Available only in Keras 2.1.0 and later.
- **Input**: optional placeholder for network input, dropped during the conversion
- **Dropout**: training-only layer, ignored in the conversion.

12.2 Lasagne

Lasagne is based on the Theano computational library. It is one of the first user-friendly toolboxes for Deep Learning. Theano is on end-of-life support, but Lasagne is supported for historical reasons as several projects are using it. Only the last version for GitHub (*0.2-dev*) is supported, while the stable *0.1* version from *pip* is not supported.

12.2.1 Model format

Lasagne had no file format for the network structure, while the weights can be saved and loaded from NUMPY (*.npz*) files. For the network structure, the tool expects a specifically crafted Python™ module building the network.

The module must have the following elements:

- **INPUT**: input of the network declared as a Theano tensor.
- **INPUT_SHAPE**: shape of the input tensor; the first dimension is considered as batch size and ignored.
- **network**: function building the network; the single input parameter must match the *INPUT* variable declared above.

The loading fails if an element is not defined or the weights in the NUMPY arrays do not match the model.

A module example is:

```
INPUT = T.tensor4('inputs')
INPUT_SHAPE = (None, 1, 60, 63)

def network(input_var):
    net = InputLayer(input_var=input_var, ..)
    net = Conv2DLayer(net, ..)
```

```
return net
```

12.2.2 Layer support

The following layers and attributes are supported:

- **Conv2DLayer**: convolutional layer, only 2D convolutions are supported. The following attributes are supported:
  - *pad*: only explicit (list of integers) padding supported
  - *filter_size*: arbitrary filter kernel sizes, provided that they are smaller than the input size
  - *stride*: arbitrary strides, provided that they are smaller than the input size
  - *num_filters*: number of output channels
  - *use_bias*: do not use bias if set to *"False"*
- **MaxPool2DLayer**: maximum pooling layer. Only the 2D version is supported. The following attributes are supported:
  - *pad*: only explicit (list of integers) padding supported
  - *filter_size*: arbitrary filter kernel sizes, provided that they are smaller than the input size
  - *stride*: arbitrary strides, provided that they are smaller than the input size
  - *ignore_border*: if *"True"*, ignores partial pooling regions in computing the output
- **AvgPool2DLayer**: average pooling layer, same attributes as the max pooling layer.
- **GlobalPoolLayer**: global max and average pooling layers, supported by setting *pool_size* and *strides* to the input size of the layer.
- **Dense**: dense (fully connected) layer. The following attributes are supported:
  - *num_units*: number of output features
  - *use_bias*: do not use bias if set to *"False"*
- **NonlinearityLayer**: nonlinear activation layer, decoded also when part of Conv2D and Dense. The following attribute is supported:
  - *nonlinearity*: type of nonlinear activation; the following functions are supported: *identity*, *linear*, *rectify*, *softmax*, *tanh*, and *sigmoid*.
- **BatchNormLayer**: batch normalization layer. The following attribute is supported:
  - *axes*: input dimensions on which the normalization is performed. Only normalization on the channel axis is supported.
- **LocalResponseNormalization2DLayer**: local response normalization layer within channels. The following attributes are supported:
  - *alpha*, *k*, *beta*, *n*: parameters of the normalization equation
- **ReshapeLayer**: changes the shape of the input tensor without changing the number of elements. The following attribute is supported:
  - *shape*: target shape. Infer (-1) values are also supported.
- **DimshuffleLayer**: dimension permutation layer; the following attribute is supported:
  - *pattern*: target dimension order given the input dimension order, only for non-batch dimensions.
- **InputLayer**: optional placeholder for network input, dropped during conversion
- **DropoutLayer**: training-only layer, ignored in the conversion
-

12.3 Caffe

X-CUBE-AI supports only the 1.0 official distribution of Caffe with the following layers:

- **Convolution**: convolutional layer. The following attributes are supported:
  - *pad*: arbitrary pad values, provided they are smaller than the filter kernel size
  - *kernel_size*: arbitrary filter kernel sizes, provided that they are smaller than the input size
  - *stride*: arbitrary strides, provided that they are smaller than the input size
  - *num_output*: number of output channels
  - *group*: number of feature groups in the output (such as for AlexNet architectures)
  - *bias_term*: do not use bias if set to *"False"*

- **Pooling**: pooling layer. The following attributes are supported:
  - *pool_size*: arbitrary pool sizes, provided they are smaller than the input size
  - *stride*: arbitrary strides, provided that they are smaller than the input size
  - *pad*: arbitrary pad values, provided they are smaller than the pool kernel size
  - *pool_function*: function used for pooling. MAX (maximum pooling) and AVE (average pooling) are supported.
  - *global_pooling*: if *"True"*, use global pooling; supported by setting *kernel_size* and *stride* to the input size of the layer.
- **InnerProduct**: dense (inner product) layer. The following attributes are supported:
  - *num_units*: number of output features
  - *use_bias*: do not use bias if set to *"False"*
- **ReLU**: ReLU nonlinear activation layer. The following attribute is supported:
  - *negative_slope*: slope in the negative *x* if used as leaky ReLU
- **Softmax**: softmax nonlinear activation layer. The following attribute is supported:
  - *axis*: input dimension on which softmax is computed. Only normalization on the channel axis is supported.
- **Scale**: dimension-wise scaling layer, it can be used as an inference-only batch normalization layer. The following attributes are supported:
  - *axes*: set of dimensions to scale
  - *use_bias*: also uses a bias term if set to *"True"*
- **LRN**: local response normalization layer, only normalization within channels is supported. The following attributes are supported:
  - *alpha*, *k*, *beta*, *local_size*: parameters of the normalization equation
  - *region*: type of normalization region. Only *"ACROSS_CHANNELS"* is supported.
- **Flatten**: flattens input dimensions to a vector; mapped as a special *Reshape* layer. The following attributes are supported:
  - *axis*, *end_axis*: range of dimensions to flatten
- **Reshape**: changes the shape of the input tensor without changing the number of elements. The following attribute is supported:
  - *shape*: target shape. Keep same (None) values are also supported.
- **Input**: optional placeholder for network input, dropped during conversion
- **Dropout**: training-only layer, ignored in the conversion

12.4 ConvNetJs

ConvNetJs is a simple DL toolbox written in JavaScript. Only the 0.3 version from *npm* is supported with the following layers:

- **conv**: 2D convolutional layer. The following attributes are supported:
  - *sx*, *sy*: filter size; arbitrary values are supported, provided they are smaller than the input size.
  - *stride*: arbitrary strides, provided they are smaller than the input size
  - *pad*: arbitrary pad values, provided they are smaller than the filter size
- **pool**: maximum pooling layer. The following attributes are supported:
  - *sx*, *sy*: pooling size,; arbitrary values are supported, provided they are smaller than the input size.
  - *stride*: arbitrary strides, provided they are smaller than the input size
  - *pad*: arbitrary pad values, provided they are smaller than the filter size
- **fc**: dense (fully connected) layer; optional bias is supported. The following attribute is also supported:
  - *out_depth*: number of output features
- **relu**, **softmax**, **sigmoid**, **tanh**: nonlinear activation layers; *softmax* is applied only on the channel dimension.
- **input**: optional placeholder for network input, dropped during conversion

13 Error handling

The X-CUBE-AI core handles a range of different errors and reports them to the user:

- Load error: due to missing or unreadable files passed to the tool
- Invalid model: the network model file is corrupted or cannot be parsed
- Invalid options: options passed to the command-line interface are not valid
- Not implemented: missing a functionality that may be introduced in future releases
- Tool error: errors due to wrongly-configured tool environment
- Internal error: unexpected errors, may indicate bugs

13.1 Load error

This error is raised if the file with the network model description does not exist or is invalid (such as a directory instead of a file). Moreover, the following condition is handled:

- *"Unable to build the network library."*: the validation failed due to a compilation error; as a result, the compiled C library is not available.

13.2 Invalid model

This error is raised if the NN model file cannot be loaded correctly. Depending on the DL toolbox chosen, different messages are reported.

13.2.1 Lasagne

- *"Couldn't load Lasagne model , error: "*: error while loading the Python™ module containing the model code; the message includes error details.
- *"Wrong parameters: , error: "*: the parameters passed are incompatible with the model; the message gives the exact reason.

13.2.2 Keras

- *"Model saved with Keras "V1" but <= "V2" is supported"*: the model has been saved with a newer version of Keras and may not be parsed correctly; loading is thus stopped.
- *"Model configuration is not a text file: "*, *"Unknown format for model configuration file: "*, *"Model file is not a HDF5 file: "*: passing the wrong file type for model configuration, weights, or both.
- *"Couldn't load Keras model | model configuration | weight , error: "*: error parsing the model files; the message gives more details such as shape mismatch information.

13.2.3 Caffe

- *"Not a Caffe model file: "*: the network structure file is not a valid *.prototxt* file.
- *"Invalid file or using the wrong Caffe version: , error: "*: error while parsing the *.prototxt* file, due either to an invalid syntax or unrecognized attributes, for instance because of a different or newer Caffe version.

13.2.4 ConvNetJs

- *"ConvNetJs model is not a valid JSON file , error: "*; the file passed is not recognized as a valid JSON file and cannot be loaded.
- *"Invalid ConvNetJs model: "*: the model has not the structure expected for a ConvNetJs NN.

13.3 Invalid options

This error is raised if wrong or meaningless options are passed to a command-line interface.

- *"unrecognized network type: "*: the type of the network to convert is not among the ones supported by the tool.

13.4 Not implemented

The tool currently supports only a subset of all the functionalities in the DL toolboxes. If a functionality not yet supported is encountered, this error is returned:

- *"Unsupported layer type: "*: the combination of layer type and toolbox is not currently supported and the layer cannot be parsed.
- *"Unsupported wrapper: "*: the current layer wrapper (meta-layer) is not supported.
- *"TimeDistributed non supported on shape "*: the *TimeDistributed* wrapper is supported only on layers with vector outputs, otherwise this error is raised.
- *"scale / bias are supported only on the input channel; used"*: the scale and bias layer is applied on an arbitrary dimensions, which is not currently handled.
- *"Unsupported cOde generation for layer "*: the generation of C code for the layer is not yet supported.
- *"unsupported nonlinearity: "*: this type of nonlinearity is not recognized or not yet supported.
- *"unsupported pool function: "*: this type of pooling function is not recognized or not yet supported; currently only *average* and *max* are handled.
- *"unsupported pool padding strategy: "*: different toolboxes use different padding methods for Pooling; the current method is not supported.
- *"Unsupported LRN region type: "*: only normalization across channels is supported for Local Response Normalization; spatial normalization raises this error.
- *"RNN nonlinearity not supported: "*: this type of nonlinearity is not supported inside a recurrent neural network.
- *"only 4b or 8b elements supported for compression, found "*: only compression to elements using 4 bits or 8 bits are handled; other element sizes raise this error.
- *"Network type cannot be validated: "*: this type of network cannot be validated automatically by the tool.

13.4.1 Keras

- *"Unsupported Keras backend: "*: only the TensorFlow™ backend is currently supported; for models saved using one of the other backends, this error is returned.
- *"Pad type not supported: "*: this type of padding strategy is not supported in Keras.
- *"RNN 'go_backwards | return_state' argument not supported"*: these two options are still not supported in recurrent layers.

13.4.2 Caffe

- *"Running statistics in Caffe BatchNorm are not supported."*: the layer is using batch statistics computed online during execution, which is not supported.

13.5 Tool error

This error is raised if an assumption on the system setup or in the results is violated. For instance, it is raised if a dependency is missing, or if an unexpected condition is encountered during the conversion of an NN . If the user's environment is correctly configured, such an error may be indicative of a bug.

13.6 Internal error

This error is raised when an unexpected or abnormal condition is realized, which is most likely the result of a bug. The tool still captures the error and reports it to the user.

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
```

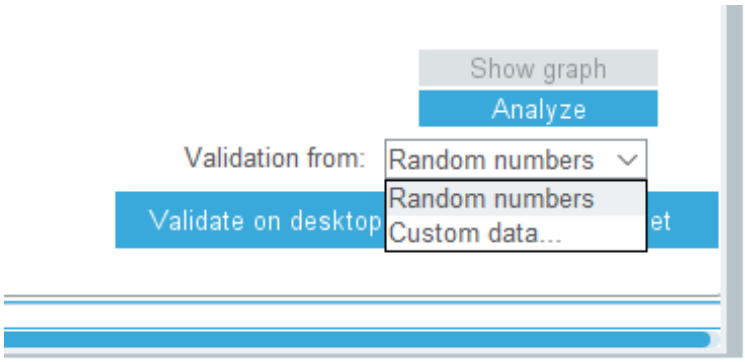
14.2 Custom data set file format?

For the validation process (refer to Section 6.2 Validation engine) a custom data set can be provided instead of the random input data (the L2 relative error is calculated in both cases). The expected file is a simple text file in *csv* format with one flattened input tensor by line. The comma is used as a separator. The dash indicates the start of a comment. Empty lines are skipped.

```
# This is a comment line
-1.0760075, 6.2789803, 3.9499009, ... , 9.112013, 0.08172209
-1.920469, 7.8589406, 1.3075534, ... , 14.818938, -1.8387469
1.719910651445388794e-03,1.286244078073650599e-04, ... ,9.708247184753417969e-01
```

Note: the `numpy.genfromtxt()` Python™ function is used to upload the data.

Figure 54. Custom data selector



14.3 Multi-network limitations?

There is no real limitation to use multiple networks other than RAM and Flash availability. Each network has its own set of `ai_<name>_XXX()` inference functions (refer to Section 8 Embedded inference client API). section). If the activation memory chunk is shared between multiple networks, care must be taken. No preemption is allowed (refer to Section 7.5 Re-entrance and thread safety considerations).

14.4 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_M
```

```
ATH_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
```

14.5 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 3. Heap and stack monitoring support

Toolchain	Stack monitor	Heap monitor	Note
GCC	Supported	Supported	SW4STM32
IAR™ 8.x and 7.x	Supported	Not implemented	-
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
...
```

14.6 Why is *“used heap”* always zero?

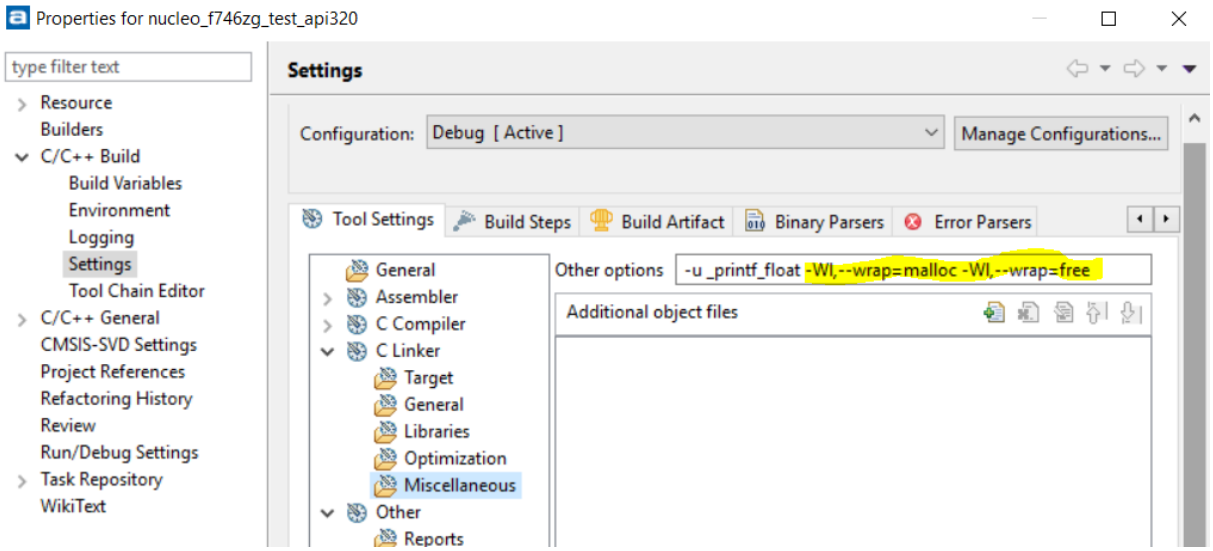
For GCC-based projects only:

```
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 55.

Figure 55. **Linker options to enable the heap monitor**



14.7 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
```

14.8 CPU cycles/MACC?

Refer to Section 4.4.1 CPU cycles/MACC? and Section 9 AI system performance application.

14.9 Is it necessary to enable or configure a TIMER IP?

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).

14.10 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.

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

The answer is twofold:

- Because the current integration of STM32 NN mapping tools is UI oriented with no CLI support yet (refer to Section 14.12), there is no built-in library export mechanism available.
- 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.

14.12 Command-line interface?

CLI is not considered in the current release of X-CUBE-AI.

15 References

Table 4. References

ID	Description	Link
[1]	NUCLEO-F746ZG development kit	www.st.com/en/product/nucleo-f746zg
[2]	Atollic TrueSTUDIO® for STM32 v9.0.1	atollic.com/truestudio
[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 - STM32Cube™ initialization code generator	www.st.com/en/product/stm32cubemx
[6]	System Workbench for STM32 (SW4STM32)	www.st.com/en/product/sw4stm32

Revision history

Table 5. **Document revision history**

Date	Version	Changes
15-Jan-2019	1	Initial release.

Contents

1 **General information** 2
1.1 What is STM32Cube™? 2
1.2 How does X-CUBE-AI complement STM32Cube™? 2
1.3 X-CUBE-AI core engine 2
1.4 STM32CubeMX extension 4
1.5 Acronyms, abbreviations and definitions 4
1.6 Prerequisites 5
1.7 License 5
2 **Installing X-CUBE-AI** 6
3 **Starting a new STM32 AI project** 8
3.1 MCU and board selector 8
3.2 HW and SW platform settings 10
3.2.1 Increase or set the CPU and system clock frequency 11
3.2.2 Set the MCU memory sub-system 12
3.2.3 CRC 13
4 **X-CUBE-AI configuration wizard** 14
4.1 Adding the X-CUBE-AI component 14
4.2 Enabling the X-CUBE-AI component 14
4.3 Uploading a pre-trained DL model file 15
4.4 Dimensioning information report 17
4.4.1 CPU cycles/MACC? 18
4.4.2 Generated C-model graph representation 18
4.5 Validating the generated C model 19
4.6 Adding a new DL model 20
5 **Generating, building and flashing** 22
5.1 Generating the IDE project 22
5.2 Building and flashing 23
6 **X-CUBE-AI internals** 24
6.1 Graph flow and memory layout optimizer 24

6.2 Validation engine . . . 25
7 Generated STM32 NN library . . . 27
7.1 Firmware integration . . . 27
7.2 Library source tree view . . . 28
7.3 Specific AI platform files . . . 29
7.4 Multi-network inference API . . . 29
7.5 Re-entrance and thread safety considerations . . . 30
7.6 Code and data placement considerations . . . 30
7.7 Debug considerations . . . 30
8 Embedded inference client API . . . 31
8.1 Input and output x-D tensor layout . . . 31
8.1.1 1-D tensor . . . 31
8.1.2 2-D tensor . . . 31
8.1.3 3-D tensor . . . 32
8.2 `create()` / `destroy()` . . . 32
8.3 `get_error()` . . . 33
8.4 `get_info()` . . . 33
8.5 `init()` . . . 34
8.6 `run()` . . . 35
9 AI system performance application . . . 36
9.1 System run-time information . . . 36
9.2 Embedded C-model network information . . . 37
9.3 Embedded C-model run-time performance . . . 38
10 AI validation application . . . 39
10.1 System run-time information . . . 39
10.2 Embedded C-model run-time performance . . . 39
10.3 Layer-by-layer run-time performance . . . 40
10.4 Final result for validation on target . . . 40
10.5 Returned error during the connection . . . 40
10.5.1 Error: no connected board, invalid firmware, or board restart needed . . . 41

10.5.2 Error: `network_name` is not a valid network . . . 42
10.5.3 Error: the embedded STM32 model does not match the C model . . . 42
11 AI template application . . . 43
12 Supported toolboxes and layers for Deep Learning . . . 44
12.1 Keras . . . 44
12.2 Lasagne . . . 45
12.2.1 Model format . . . 45
12.2.2 Layer support . . . 46
12.3 Caffe . . . 46
12.4 ConvNetJs . . . 47
13 Error handling . . . 48
13.1 Load error . . . 48
13.2 Invalid model . . . 48
13.2.1 Lasagne . . . 48
13.2.2 Keras . . . 48
13.2.3 Caffe . . . 48
13.2.4 ConvNetJs . . . 48
13.3 Invalid options . . . 48
13.4 Not implemented . . . 48
13.4.1 Keras . . . 49
13.4.2 Caffe . . . 49
13.5 Tool error . . . 49
13.6 Internal error . . . 49
14 FAQs . . . 50
14.1 Log files for debug purpose? . . . 50
14.2 Custom data set file format? . . . 50
14.3 Multi-network limitations? . . . 50
14.4 Unable to compile file *arm_dot_prod_f32.c* . . . 50
14.5 Used heap or stack: disabled or not yet supported . . . 51
14.6 Why is *“used heap”* always zero? . . . 51
14.7 Formatted floating-point numbers are empty for a GCC-based project . . . 52

14.8 CPU cycles/MACC? 52
14.9 Is it necessary to enable or configure a TIMER IP? 52
14.10 How to update only the exported NN library in my generated project? 52
14.11 Is it possible to export an NN library for a non-STM32CubeMX-based project? 52
14.12 Command-line interface? 52
15 References 53
Revision history 54
Contents 55
List of tables 59
List of figures 60

List of tables

Table 1. Definition of terms used in this document 5
Table 2. System informations reporting 17
Table 3. Heap and stack monitoring support 51
Table 4. References 53
Table 5. Document revision history 54

List of figures

Figure 1. X-CUBE-AI core engine . . . 3
Figure 2. X-CUBE-AI 3.3.0 overview . . . 3
Figure 3. X-CUBE-AI core in STM32CubeMX . . . 4
Figure 4. Managing embedded software packs in STM32CubeMX . . . 6
Figure 5. Installing X-CUBE-AI in STM32CubeMX . . . 6
Figure 6. X-CUBE-AI in STM32CubeMX . . . 7
Figure 7. Creating a new project . . . 8
Figure 8. AI filter . . . 8
Figure 9. AI filter with default option . . . 9
Figure 10. AI filter with compression x4 . . . 9
Figure 11. NUCLEO-F746ZG board selection . . . 10
Figure 12. Initialize all peripherals . . . 10
Figure 13. USART3 configuration . . . 11
Figure 14. Clock wizard pop-up . . . 11
Figure 15. System clock settings . . . 12
Figure 16. MCU memory sub-system (parameter settings) . . . 12
Figure 17. Enabling the CRC IP . . . 13
Figure 18. Additional software button . . . 14
Figure 19. Adding the X-CUBE-AI core component . . . 14
Figure 20. Add-on X-CUBE-AI applications . . . 14
Figure 21. Main X-CUBE-AI configuration panel . . . 15
Figure 22. X-CUBE-AI platform setting panel . . . 15
Figure 23. NN configuration wizard . . . 16
Figure 24. Insufficient RAM/Flash message box . . . 16
Figure 25. Uploaded and analyzed DL model . . . 17
Figure 26. Integrated C-model (runtime-view) . . . 18
Figure 27. Generated C-model graph . . . 19
Figure 28. Validation status field . . . 20
Figure 29. Validate on desktop - log report . . . 20
Figure 30. Main view with multiple networks . . . 21
Figure 31. Project settings view for IDE Code generator . . . 22
Figure 32. AI IP not fully configured . . . 23
Figure 33. Weight/bias compression . . . 24
Figure 34. Operation fusing . . . 24
Figure 35. Optimal activation/working buffer . . . 25
Figure 36. Validation flow overview . . . 25
Figure 37. L2 computation . . . 25
Figure 38. Validation on target . . . 26
Figure 39. MCU integration model and view . . . 27
Figure 40. 1-D Tensor data layout . . . 31
Figure 41. 2-D Tensor data layout . . . 32
Figure 42. 3-D Tensor data layout . . . 32
Figure 43. System run-time information - Keil® IDE . . . 36
Figure 44. System run-time information - Atollic IDE . . . 37
Figure 45. C-model network information . . . 37
Figure 46. C-model run-time performance . . . 38
Figure 47. C-model run-time performance with heap and stack checking . . . 38
Figure 48. Host COM port selector for validation on device . . . 39
Figure 49. System run-time information . . . 39
Figure 50. C-model run-time performance . . . 40
Figure 51. Layer-by-layer results - Validation on target . . . 40

Figure 52. Final report for validation on target 40
Figure 53. AI valid - Initial log. 41
Figure 54. Custom data selector. 50
Figure 55. Linker options to enable the heap monitor 52