ModusToolbox™ Machine Learning user guide

ModusToolbox™ version 3.x

Machine Learning pack version 2.0

About this document

Scope and purpose

This document helps you understand all the various aspects of the ModusToolbox™ Machine Learning (ML) solution.

Document conventions

Convention	Explanation
Bold	Emphasizes heading levels, column headings, menus and sub-menus
Italics	Denotes file names and paths.
`Courier New`	Denotes APIs, functions, interrupt handlers, events, data types, error handlers, file/folder names, directories, command line inputs, code snippets
File > New	Indicates that a cascading sub-menu opens when you select a menu item

Abbreviations and definitions

The following define the abbreviations and terms used in this document:

- ML – Machine Learning
- NN –neural network
- NPZ – NumPy array in zipped format [1]
- TFLM – TensorFlow Lite for microcontrollers

References

Refer to the following documents and websites for more information as needed:

- ModusToolbox™ tools package user guide
- ModusToolbox™ Machine Learning Configurator guide
- https://github.com/infineon/ml-inference
- https://github.com/infineon/ml-middleware
- https://github.com/infineon/ml-tflite-micro
- https://www.infineon.com/cms/en/design-support/tools/sdk/modustoolbox-software/modustoolbox-machine-learning

[1] When unzipped, it provides validation data as NumPy arrays used by the tool.

Table of contents

1 Overview

ModusToolbox™ is a set of tools to help you develop applications for Infineon devices. These tools include GUIs, command-line programs, software libraries, and third-party software that you can use in just about any combination you need. For more information, refer to the ModusToolbox™ tools package user guide. One part of the ModusToolbox™ ecosystem is the ML solution.

The ML 2.0 solution is a set of tools, libraries, and middleware that will help you build, evaluate and benchmark pre-trained ML models. The ML libraries easily and efficiently run the inference on an Infineon MCU. These libraries and tools help you rapidly deploy neural network (NN)-based classification-type ML applications.

Note: Data collection and training algorithms are not part of the ModusToolbox™ ML 2.0 release.

The ML solution also provides a configurator to import pre-trained machine learning models and generate an embedded model (as C-code or binary file). This generated model can be used with the ML library along with your application code for a target device. The tool also lets you fit the pre-trained model of choice and evaluate its performance.

The ML models are often created on standard training frameworks such as TensorFlow, Keras, and PyTorch. They are kept in various formats such as H5, TFLite, and ONNX, etc. The configurator converts the model to run in the Infineon MCU, as well as to run regression to assess performance degradation in the ML model, if any, in the process of conversion. If data is not available, it can load random data to do such verification. The most popular ML models are NN-based and the ModusToolbox™ ML tooling is targeted for NN-based ML models.

During the assessment, you can validate the performance of the optimized model by checking performance against test data (accuracy), and visualize the implementation resource requirements, such as the number of cycles to run the inference engine and the memory size of the NN model and inference working memory. The report includes data on the inference engine when running on a computer (reference model with the best accuracy and higher memory requirements), and on the target device (limited hardware resources).

For more information about the ML solution, visit this website:

https://www.infineon.com/cms/en/design-support/tools/sdk/modustoolbox-software/modustoolbox-machine-learning/

2 Getting started

To begin using the ML tools, you must first install the ModusToolbox™ tools package version 3.0.0 or higher. This tool is located on the Infineon website:

https://softwaretools.infineon.com/tools/com.ifx.tb.tool.modustoolboxpackmachinelearning

Alternatively, you can use the Infineon Developer Center (IDC) to install the ModusToolbox™ tools package and Machine Learning pack. Download the IDC launcher using this link:

https://www.infineon.com/cms/en/design-support/tools/utilities/infineon-developer-center-idc-launcher/

After installing the IDC tool, open the launcher and search for the following tools and install them:

- ModusToolbox tools package 3.0.0 or higher
- ModusToolbox Machine Learning pack

You must also install the QEMU tool on your computer. Refer to the Software requirements section for more details.

After all the tools are installed, you can get started with a code example. Infineon provides ML code examples, which include *README.md* files that guide you through the process to create and configure the application. There are a few ways to do this:

2.1 Creating a ModusToolbox™ application

- If you're new to the ModusToolbox™ environment, use the Eclipse IDE for ModusToolbox™. Refer to the quick start guide as needed.
- If you prefer not to use Eclipse, use the ModusToolbox™ Project Creator as a stand-alone tool and look for starter applications whose names start with "ML." Refer to the Project Creator user guide for more details.

Note: If you created an application before installing the Machine Learning pack, you must run `make getlibs` *or use the Library Manager* ***Update*** *button to regenerate files before using the configurator for that application.*

2.2 Downloading a code example

The following are direct links to a couple of ML code examples:

https://github.com/Infineon/mtb-example-ml-profiler

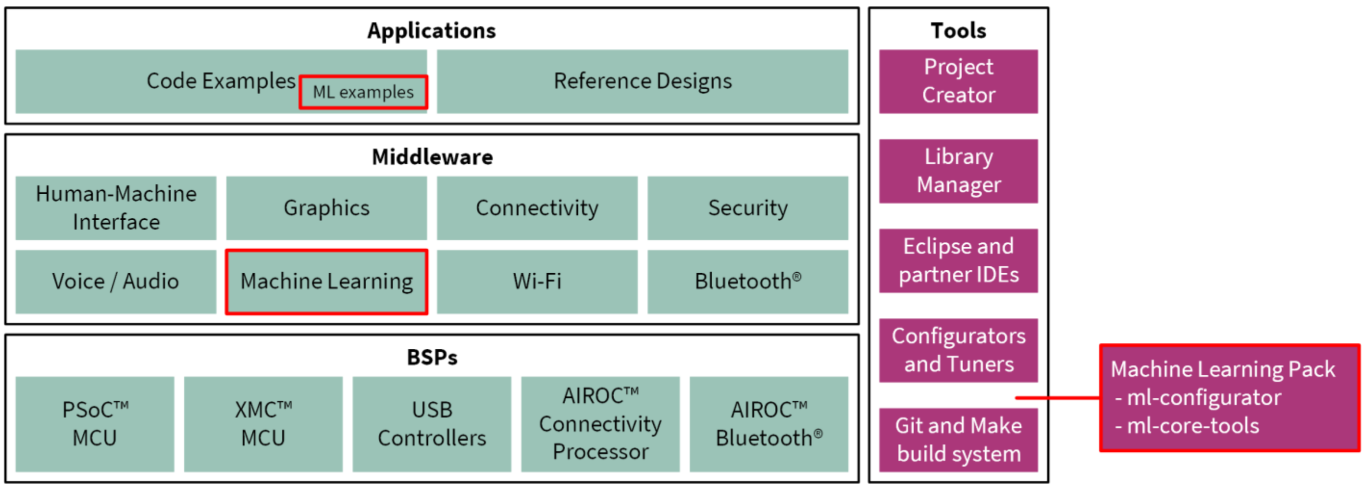
https://github.com/Infineon/mtb-example-ml-gesture-classification

As we add more code examples, you can access the complete list using the following link:

https://github.com/Infineon?q=mtb-example-ml%20NOT%20Deprecated

3 Features and architecture

The following diagram shows a very high-level view of what is available as part of ModusToolbox™ software. This is not a comprehensive list. It merely conveys the idea that there are multiple resources available to you. The ML solution is provided as a pack that you install in addition to the standard ModusToolbox™ tools package. The ML middleware is available through GitHub and used by the applicable code examples and reference designs.



3.1 Features

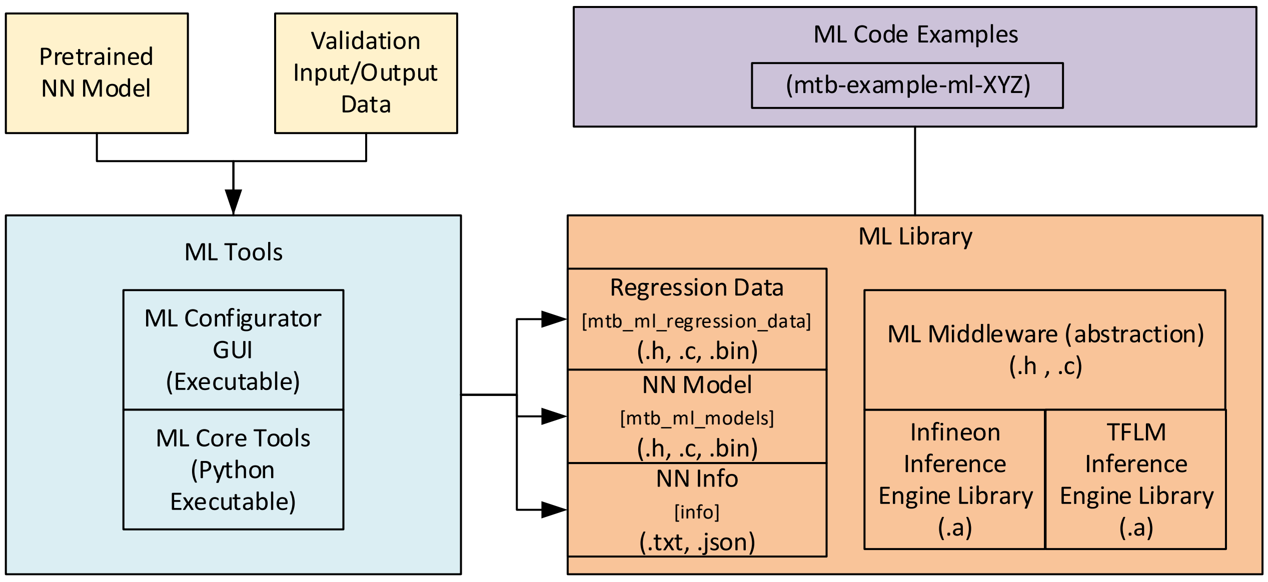
- Supports the TFLite and H5 Model format
- Supports two types of inference engines:
  - TensorFlow Lite for microcontrollers (TFLM) inference engine
  - Infineon inference engine
- Supports the following characteristics of NNs:
  - Core NN Kernels: MLP, GRU, Conv1d, Conv2d, LSTM
  - Support NN Kernels: flatten, dropout, reshape, input layer
  - Activations: relu, softmax, sigmoid, linear, tanh
- Input Data Quantization Level:
  - 32-bit float
  - 16/8-bit integer
- NN Weights Quantization Level:
  - 32-bit float
  - 16/8-bit integer
- Regression Data Evaluation
- Cycle and memory estimation
- PC based inference engine
- Target device-based inference engine (optimized)

3.2 Components

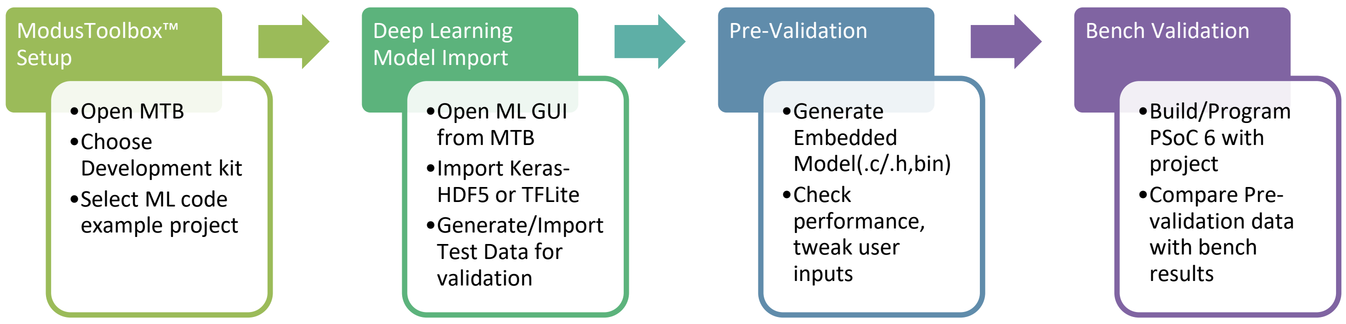
The ModusToolbox™ ML solution consists of four major components:

- ML Configurator: a GUI tool with options to load pre-trained models and test data. It also provides graphical plots and accuracy data when validating the optimized models against test data.
- ML core tool: a collection of Python scripts (provided as an executable) to perform parsing, regression and conversion of NNs.
- ML library: includes middleware helper functions, inference engine libraries to work with an NN model and input data.
- ML code examples: a collection of applications on how to use the ML libraries.

The following figure shows how these four assets interconnect with each other.


Pretrained NN Model
Validation Input/Output Data
ML Code Examples
(mtb-example-ml-XYZ)
ML Tools
ML Configurator GUI (Executable)
ML Core Tools (Python Executable)
ML Library
Regression Data [mtb_ml_regression_data] (.h, .c, .bin)
NN Model [mtb_ml_models] (.h, .c, .bin)
NN Info [info] (.txt, .json)
ML Middleware (abstraction) (.h , .c)
Infineon Inference Engine Library (.a)
TFLM Inference Engine Library (.a)

3.3 Development flow


ModusToolbox™ Setup
•Open MTB
•Choose Development kit
•Select ML code example project
Deep Learning Model Import
•Open ML GUI from MTB
•Import Keras-HDF5 or TFLite
•Generate/Import Test Data for validation
Pre-Validation
•Generate Embedded Model(.c/.h,bin)
•Check performance, tweak user inputs
Bench Validation
•Build/Program PSoC 6 with project
•Compare Pre-validation data with bench results

3.4 Hardware requirements

This solution can run on any PSoC™ 6 MCU. When it comes to designing the NN, pay special attention to the memory size and performance requirements to run the inference engine on the target device. The recommended hardware/kit platform is the CY8CKIT-062S2-43012. Other PSoC™ 6-based platforms can be targeted as well, but please note the documentation and code examples may not work as expected.

To better demonstrate the capability to run ML applications on the PSoC™ 6 MCU, sensor data needs to be sampled by the device. The CY8CKIT-028-SENSE IoT Sense Expansion kit provides a collection of sensors that are compatible with all Arduino-based PSoC™ 6 MCUs. For more details about this kit, visit this webpage: https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-028-sense/.

3.5 Software requirements

In addition to installing the ModusToolbox™ tools package and Machine Learning pack, as described under Getting started, this ML solution also requires the open source software QEMU. You can download it at this page:

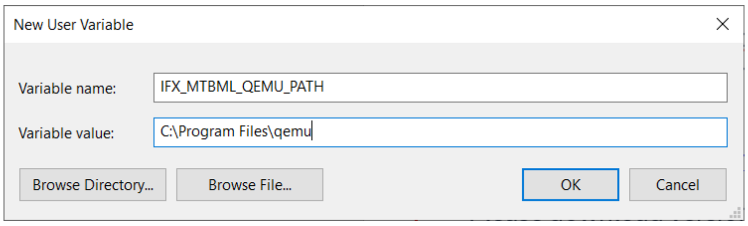
https://www.qemu.org/download/

Please download and install version **6.2.0** based on your applicable operating system. If you have already installed a newer version of QEMU, we recommend that you uninstall it first.

3.5.1 Windows

On Windows, use the executable provided from the QEMU website (link). After installing, create a new system variable called `IFX_MTBML_QEMU_PATH` and set it to the folder path of the *qemu-system-arm* executable. For example:

C:\Program Files\qemu (or whatever path you installed QEMU)


New User Variable

Variable name: IFX_MTBML_QEMU_PATH

Variable value: C:\Program Files\qemu

Note: *Ensure that the ML Configurator tool and Eclipse IDE for ModusToolbox™ 3.0 are closed when creating/updating the system variable. You might need to restart your computer.*

3.5.2 macOS

On MacOS, install QEMU using these instructions for version 6.2.0:

```
> wget https://raw.githubusercontent.com/Homebrew/homebrew-
core/71cdf768e447b09cf607adcd100060347678dc84/Formula/qemu.rb
> export HOMEBREW_NO_INSTALL_CLEANUP=1
> export HOMEBREW_NO_INSTALLED_DEPENDENTS_CHECK=1
> brew install ./qemu.rb
```

Note: *You might need to install some other dependency packages for this installation. Please follow instructions from this link:*

https://brew.sh/

Once QEMU has been installed, you need to create a new system variable `IFX_MTBML_QEMU_PATH` and set it to the path where the QEMU was installed. For example, include the following command in your *.bash_profile*:

```
> export IFX_MTBML_QEMU_PATH="/usr/local/opt/qemu/bin"
```

3.5.3 Linux

On Linux, if your distribution provides qemu-6.2.0 (e.g. Ubuntu 22.04) you can install via your package manager. Otherwise, build locally from the source code (version 6.2.0) downloaded from the QEMU website. Follow these build instructions:

```
> wget https://download.qemu.org/qemu-6.2.0.tar.xz
> tar xvJf qemu-6.2.0.tar.xz
> cd qemu-6.2.0
> ./configure --cc=gcc --cxx=g++ --target-list=arm-softmmu,aarch64-softmmu --
static --prefix=/usr/local/qemu-6.2.0
> make
> make install
```

Note: *You might need to install some other dependency packages for the build. Here is a list of commands to run first:*

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install ninja-build
sudo apt-get install libglib2.0-dev
sudo apt-get install -y libpixman-1-dev
```

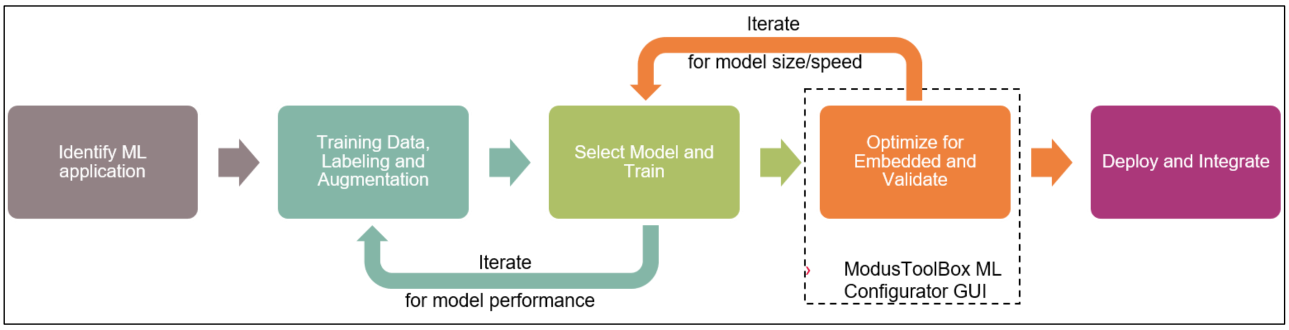
Once QEMU has been installed, you need to create a new system variable `IFX_MTBML_QEMU_PATH` and set it to the path provided to the **./configure --prefix** argument. For example, include the following command in your **".profile"** or **".bashrc"**:

```
> export IFX_MTBML_QEMU_PATH="/usr/local/qemu-6.2.0/bin "
```

4 ModusToolbox™ ML Configurator

The ML Configurator accepts a pretrained ML model and generates an embedded model (as a C header or binary file), which can be used along with your application code for a target device. The tool lets you fit the pretrained model of choice to the target device with a set of optimization parameters. The tool is provided as a GUI and a command line tool that provide the same functionality. The GUI includes its own user guide, available from the **Help** menu. The command line tool includes a `-h` switch to display the various options available.

The following shows the design flow for a typical application. The ML Configurator forms a vital part in fitting the model to the target platform.



The ML Configurator has two options for inference engine:

- TFLM inference engine
- Infineon inference engine

Each option comes with a different set of configurations, as explained in the next sections.

Note: *The labeling, augmentation and training steps are handled by a third-party tool or machine learning framework, such as TensorFlow.*

Note: *When generating code or validating a model with the ML Configurator, you might see some warning messages related to the TensorFlow AutoGraph module. These warnings should not cause any problem and can be safely ignored. For more information about this, access the link provided in the warning message.*

4.1 Choosing a model

The first step when using the ML Configurator tool is to choose a pre-trained NN model.

Note: *There are many formats to define a NN model. The format of the NN depends on the platform used for training among other things.*

Note: *The TensorFlow framework supports Keras-H5 and TFLite model format. When generating Keras-H5 format with TensorFlow framework, you must use the TF v2.4.0 or later; otherwise, the model might fail when loading to the ML Configurator.*

The following table shows some popular formats:

Format	Supported in v2.x?	File extension	Link
Keras-H5	Yes	.h5, .hdf5	https://keras.io/
Tensorflow-protobuf	No	.pb	https://www.tensorflow.org/
TFLM	Yes	.tflite	https://www.tensorflow.org/lite
Caffe	No	.caffemodel .prototxt	https://caffe.berkeleyvision.org/
PyTorch	No	.pkl	https://pytorch.org/
Open NN Exchange	No	.onnx	https://onnx.ai/

If using the TFLM inference engine with a Keras-H5 model and quantization enabled, you must also provide calibration data. The following table shows coverage items by each supported format.

Format	Coverage
Keras-H5 (Layers)	Activation / Add / AveragePooling1D / AveragePooling2D / BatchNormalization / Clipped RELU / Concatenate / Conv1D / Conv2D / Dense / DepthwiseConv2D / Dropout / Flatten / GlobalAveragePolling1D / GlobalAveragePooling2D / GlobalMaxPooling1D / GlobalMaxPooling2D / GRU[2] / InputLayer / LeakyReLU / LSTM[2] / MaxPooling1D / MaxPooling2D / ReLU / Reshape / SeparableConv2D / Softmax / Transpose / Upsampling
TFLM (Operators)	Abs / Add / Add_N / Arg_Max/ Arg_Min / Assign_Variable / Average_Pool_2D / Batch_To_Space_Nd / Call_Onc / Ceil / Concatenation / Conv_2D / Cos / CumSum / Depth_To_Space / Depthwise_Conv_2D / Dequantize / DetectionPostprocess / Elu / Equal / EthosU / Expand_Dims / Floor / Floor_Div / Floor_Mod / Fully_Connected / Greater / Greater_Equal / Hard_Swish / L2_Normalization / L2_Pool_2D / Leaky_Relu / Less / Less_Equal/ Log / Logical_And / Logical_Not / Logical_Or / Logistic / Max_Pool_2D / Maximum / Mean / Minimum / Mul / Neg / Not_Equal / Pack / Pad / PadV2 / Prelu / Quantize / Read_Variable / Reduce_Max / Relu / Relu6 / Reshape / Resize_Bilinear / Resize_Nearest_Neighbor / Round / Rsqrt / Shape / Sin /Softmax / Space_To_Batch_Nd / Space_To_Depth / Split / Split_V / Sqrt / Square / Squeeze / Strided_Slice / Sub / Svdf / Tanh / Transpose / Transpose_Conv / Unpack / Var_Handle / Cast / CircularBuffer / Exp / Fill / Gather / GatherNd / If / MirrorPad / Slice / ZerosLike

4.2 Importing test data

You can input your own test data for verification. Alternatively, you can use the ML Configurator to generate random test data for validation. To input the test data, click on the **Validate in Desktop** or **Validate on Target** tab and choose the desired settings. The data can be stored with the following dataset structures:

4.2.1 In a single CSV file (ML format)

The CSV file must contain only numeric data, no header and no sample ID columns. You must specify the first feature column and the number of features in the ML Configurator. You also must specify the first target column and the number of targets. The tool supports 1D feature data only. If your model requires 2D or 3D feature data, use the NPZ format instead.

2 The GRU and LSTM layers from the Keras-H5 are not supported by the TFLM inference engine; only with the Infineon inference engine.

4.2.2 In a single NPZ format file (uses NumPy)

Features and labels can be loaded from a single NumPy compressed file. The feature and label must have been saved in the following order when creating the NumPy file:

```
import numpy as np
x = np.ones((n_samples, n_features))
y = np.ones((n_samples, n_labels))
np.savez('saved_features', x , y)
```

In this example, we populate x and y with "ones". In a real scenario, you should populate with actual data, where `n_samples` is the number of samples; `n_features` is the number of features; and `n_labels` is the number of labels. 2D and 3D feature data are also supported, represented as:

- 2D: (n_samples, a, b)
- 3D: (n_samples, a, b, c)

4.2.3 In a folder format

This format separates the data by label. A base directory contains subfolders named by label, containing data files (nodes) with data for each label. The only supported node file formats are JPEG and CSV.

4.3 Evaluating a model

Based on the imported test data, you can feed the input test data to the inference engine emulated on the computer or in the target, then analyze the result from the inference engine with the output test data. You may also choose the quantization in which the output model needs to be generated. There are three options:

- 8-bit
- 16-bit [not available with TFLM]
- Floating number

The model conversion takes the reference model imported to the tool and converts it to C code for floating-point and integer. The floating-point mode uses input and model weights that are 32-bit precision and the regression on this converted model yields the highest accuracy. Although floating-point is more accurate than integer mode, it requires the most memory, and is not always desirable to embed on the target device. Integer has the following fixed-point modes:

- 16x16 bit precision (16-bit input, 16-bit weights)
- 16x8 bit precision (16-bit input, 8-bit weights)
- 8x8 bit precision (8-bit input, 8-bit weights)

As a rough rule of thumb, the 16x16 format gives the best trade-off between model accuracy and size. Specifically for the PSoC™ 6 platform, 16-bit operations are more computationally efficient when compared to 8-bit. A small degradation of accuracy can be expected when converting the floating-point model to integer modes. The ML Configurator converts and performs verification tests on both, and you can select which model best suits your specifications.

In summary, take the following into consideration when selecting the model for the PSoC™ 6 platform.

Model on Target Device	Model Size	Working Memory	Cycles Requirements	Accuracy	Dynamic Range
Floating-point	🟥 High	🟥 High	🟥 High	🟩 High	🟩 Excellent
Fixed-point 16x16	🟨 Medium	🟨 Medium	🟩 Low	🟨 Medium	🟨 Good
Fixed-point 16x8	🟩 Small	🟨 Medium	🟨 Medium	🟨 Medium	🟨 Good

Model on Target Device	Model Size	Working Memory	Cycles Requirements	Accuracy	Dynamic Range
Fixed-point 8x8	Small	Low	Medium	Low	Low

When validating the device on the PC, a report tells if the model passed or failed. It considers as a pass when the accuracy is equal or greater than 98%.

In the ML Configurator, if you choose the Infineon Inference Engine , there is an "Advanced scratch memory optimization" check box, which is enabled by default. This feature reduces the amount of scratch memory required by the Infineon inference engine; however, it might affect the accuracy and/or the number of cycles of the inference engine.

If you choose the TFLM inference engine and provide a Keras-H5 model, a model calibration group box appears. You can input some calibration data to improve the accuracy of the model after quantization. The format options of this calibration data are the same as the one used for validation. If you provide a **.tflite* model with the TFLM inference engine, there is no option to enable model quantization. However, if a **.tflite* model was created using int8x8 quantization, you must indicate the imported model has such quantization in the ML Configurator. By default, it assumes the **.tflite* model uses floating point.

Another feature provided when using the TFLM inference engine is to run it without an interpreter (TFLM Interpreter-less), which can reduce the amount of memory required by the inference engine. However, this option doesn't allow run the validation in desktop option, only in the target. For more information about this feature, refer to the TFLM inference engine library section.

And last, the TFLM inference engine has also a feature to take advantage of sparsity if the trained model utilizes pruning. Note that this document does not cover any guidelines on how to prune a model during the training process. It assumes the developer provides a pre-trained pruned model, which then can be further optimized by packing sparse weights present in the model, saving memory when deploying the model to the target device.

4.4 Exporting model on target device

When the model has been converted with an acceptable accuracy and satisfied the memory requirements for the target device, it can now be exported on the target device. The ML Configurator generates a header file or binary file containing the simplified model with the desired quantization. It can also export regression data to profile the model in the target device.

All this data can be exported to a directory you specify, which usually is part of the application in development. The following shows a summary of files exported by the tool. These files depend on the inference engine and quantization selected.

Assume the name of the model is "NN" and the root folder is "mtb_ml_gen". The "<type>" can assume *float*, *int8x8, int16x8*, or *int16x16*, and the "<inference>" can assume *tflm* (runtime interpreter), *tflm_less* (interpreter-less) or *ifx* (Infineon inference engine).

Folder / File Name	Description
/mtb_ml_gen/	Root folder. Name is chosen by the user
/mtb_ml_gen/info/	General information folder about the model (for internal use only)
/mtb_ml_gen/model_gen_dir/	Folder that stores modified/converted model files (for internal use only)
/mtb_ml_gen/mtb_ml_models/	Folder that contains the model weights/bias and parameters
NN_<inference>_model_<type>.h	Header file with model array [ifx,tflm] or function [tflm_less] declaration
NN_<inference>_model_<type>.c	C file containing the flat-buffer array definition for weights/bias [ifx, tflm]
NN_<inference>_model_<type>.cpp	C++ file containing the implementation of TFLM functions [tflm_less]

Folder / File Name	Description
NN_<inference>_model_<type>.bin	Weights/bias array in binary format [ifx, tflm]
NN_<inference>_model_prms.bin	Parameters array in binary format [ifx only]
/mtb_ml_gen/mtb_ml_regression_data/	Folder that contains regression data
NN_<inference>_x_data_<type>.h	Header file with input data array declaration
NN_<inference>_x_data_<type>.c	C file containing the input data array definition
NN_<inference>_x_data_<type>.bin	Input data array in binary format
NN_<inference>_y_data_<type>.h	Header file with output data array declaration
NN_<inference>_y_data_<type>.c	C file containing the output data array definition
NN_<inference>_y_data_<type>.bin	Output dt array in binary format

Note: *Multiple models can be stored in the same folder, since the name of the MODEL is used as a prefix for all the files.*

Note: *Regression data generated for the "tflm" and "tflm_less" is the same, so the <inference> is set to "tflm" for both.*

The binary files are useful when storing the data in a file system in the target device. If using header files, the data is available as C arrays with the name format shown in the following table. The header file also provides some `#define` about the model:

File name: ***NN_<inference>_model_<type.h>***	
Array name	**Description**
`NN_model_prms_bin[]`	Model parameters [ifx only]
`NN_model_bin[]`	Weights/bias [tfm, ifx]
DEFINE name	**Description**
`NN_MODEL_PRMS_BIN_LEN`	Length of the parameters array in bytes [ifx only]
`NN_MODEL_BIN_LEN`	Length of the weights/bias array in bytes [ifx, tflm]
`NN_ARENA_SIZE`	Size of the TFLM arena in bytes [tflm only]
`NN_MODEL_NUM_OF_LAYERS`	Number of layers [ifx only]
`NN_MODEL_NUM_OF_RESIDUAL_CONN`	Number of residual connections [ifx only]
`NN_MODEL_INPUT_DATA_SIZE`	Size of the input layer [ifx only]
`NN_MODEL_OUTPUT_DATA_SIZE`	Size of the output layer [ifx only]
`NN_MODEL_SIZE_OF_STATE_OUT`	Size of the state output [ifx only]
`NN_MODEL_SCRATCH_MEM_SIZE`	Size of the scratch memory in bytes [ifx only]

File name: ***NN_<inference>_x/y_data_<type>.h***	
Array name	**Description**
`NN_x_data_bin[]`	Input data
`NN_y_data_bin[]`	Output data
DEFINE name	**Description**
`NN_X_DATA_BIN_LEN`	Length of the input data in bytes
`NN_Y_DATA_BIN_LEN`	Length of the output data in bytes

Note: *The NN_<inference>_model_<type.h> file contains an array with the model weights. By default, it stores it in the flash and loads automatically to the RAM. There is an option to place this array in a specific section of the memory. Simply define the `CY_ML_MODEL_MEM` macro in the Makefile to a section available in the linker script. In terms of performance, the inference engine runs faster when placing the model weights in RAM, and slower if placing in the external memory.*

To place to the internal flash only, you can add:

```
DEFINES+=CY_ML_MODEL_MEM=".constdata"
```

To place to external memory, you can add:

```
DEFINES+=CY_ML_MODEL_MEM=".cy_xip"
```

Refer to the https://github.com/Infineon/mtb-example-psoc6-qspi-xip code example to properly setup the XIP mode using QSPI.

This approach does not work with the PSoC™ 64 family, due to the protection settings to work in XIP mode.

The following table shows the main differences between using C header files and binary files.

File Format	Requires File System	Easy to Upgrade	Access	Scalability
Header File	Optional	No. Part of the firmware image	Easy. Directly access a variable in firmware	Low. Usually placed on internal memory.
Binary	Recommended	Yes. Update a file in the filesystem	Hard. Need to open and load a file	High. Usually placed on external memory

When using the TFLM Interpreter-less feature, the model file generated contains functions instead of arrays. The following table shows a summary of the functions and defines.

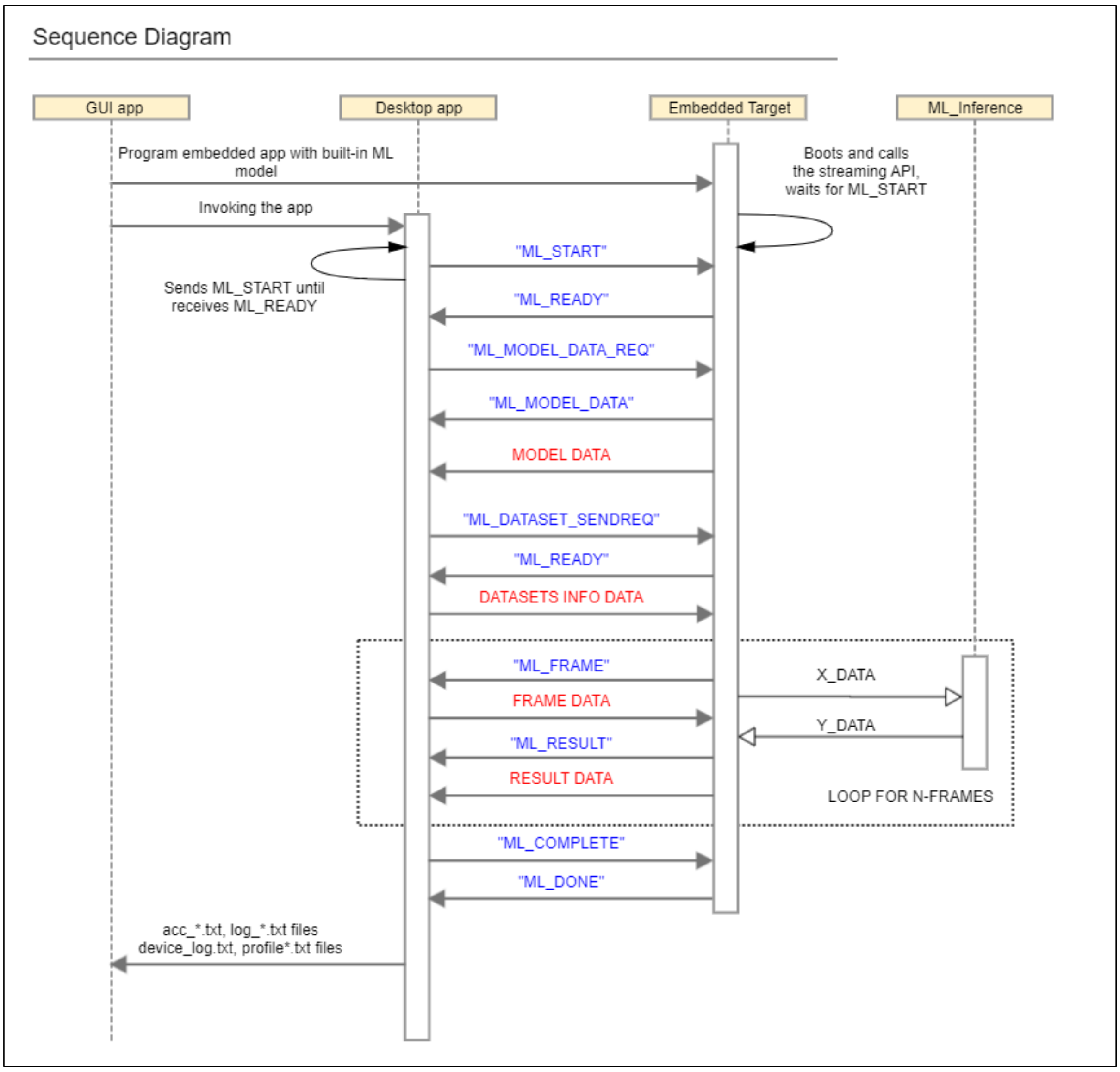
File name: *NN_tflm_less_model_<type.h>*	
Function name	**Description**
`NN_init()`	Sets up the model with init and prepare steps
`NN_input(index)`	Returns the input tensor with the given index
`NN_output(index)`	Returns the output tensor with the given index
`NN_invoke()`	Runs inference for the model
`NN_inputs()`	Returns the number of input tensors
`NN_outputs()`	Returns the number of output tensors
`NN_input_ptr(index)`	Return the buffer pointer of input tensor
`NN_input_size(index)`	Return the buffer size of input tensor
`NN_input_dims_len(index)`	Return the dimension size of input tensor
`NN_input_dims(index)`	Return the dimension buffer pointer of input tensor
`NN_output_ptr(index)`	Return the buffer point of output tensor
`NN_output_size(index)`	Return the buffer size of the output tensor
`NN_output_dims_len(index)`	Return the dimension size of output tensor
`NN_output_dims(index)`	Return the dimension buffer pointer of output sensor
DEFINE name	**Description**
NN_MODEL_CONST_DATA_SIZE	Non-volatile data size used by this model
NN_MODEL_INIT_DATA_SIZE	Initialized volatile data size used by this model
NN_MODEL_UNINIT_DATA_SIZE	Uninitialized volatile data size used by this model

The input regression data (*x_data*) contains not only the input data, but also the data input type, number of inputs in the first layer of the NN, the number of samples and the Q-format. Here is how to parse the data:

Location	Description
byte 0	Data input type: 0 → Unknown 1 → Float 32-bit 2 → Integer 8-bit 3 → Integer 16-bit
byte 1	
byte 2	
byte 3	
byte 4	Number of samples in the dataset
byte 5	
byte 6	
byte 7	
byte 8	Number of nodes in the first layer (input) or frame size of the sample dataset
byte 9	
byte 10	
byte 11	
byte 12	Q-format of the sample dataset. Indicates the number of fraction bits [Only used for Infineon inference]
byte 13	
byte 14	
byte 15	
byte 16	Data samples
byte …	

4.5 Streaming regression data to the Device

The ML 2.0 solution comes with a feature that enables streaming the regression data to the target device through the UART. Instead of storing the regression data locally, as explained in the previous section, the ML Configurator tool streams the exact same data to the target device. Here is the overall sequence diagram of the protocol in place.



The simplest way to leverage the streaming capability is to use the mtb-example-ml-profiler code example. Ensure the operating system used to build this code example and run the ML configurator tool are the same. If they are different, the UART baud rate configured in the firmware needs to change to match the following speed:

Operating System	Windows	Linux	macOS
UART baud rate	1 Mbps	1 Mbps	115.2 Kbps

To change the baud rate in the code example, open the *main.c* file and set the correct speed to the `cy_retarget_io_init()` function.

4.6 Using command line

The ML Configurator comes with a command line executable, named as *ml-configurator-cli*. To see all the options available, you can type:

```
ml-configurator-cli -h
```

This command prints the following message:

```
Options:
  -?, -h, --help              Displays help on commandline options.
  --help-all                  Displays help including Qt specific options.
  -v, --version               Displays version information.
  -c, --config <config_file>  Path to the
                              configuration file.
  -o, --output-dir <dir>      The path to the
                              generated source directory. It is either an
                              absolute path or a path relative to the
                              configuration file parent directory.
  --convert                   Analyze and convert
  --evaluate                  Evaluate
```

The option `--config` requires a configuration file with extension MTBML, which is generated by the ML Configurator GUI.

If the configurator is not used to generate a *.mtbml* file, then one must be generated before using the ml-configurator-cli tool. The following example should be used as a template. The JSON file must have an extension of .mtbml and use UTF-8 encoding.

Formatted JSON example:

```
{
    "app": "ML",
    "calibration_data": {
        "active_state": true,
        "feat_col_count": 784,
        "feat_col_first": 1,
        "input_calibration_type": "ML",
        "input_format": "JPEG",
        "path": "test_data/test_data.csv",
        "target_col_count": 1,
        "target_col_first": 0
    },
    "filetype": "modustoolbox-ml-configurator",
    "formatVersion": "3",
    "lastSavedWith": "ML Configurator",
    "lastSavedWithVersion": "2.0.0",
    "model": {
        "framework": "TFLITE",
        "optimization_ifx": "SIZE",
        "optimization_tflm": false,
        "path": "pretrained_models/small_mlp_mnist.h5",
        "quantization": {
            "float32": true,
            "int16x16": false,
            "int16x8": false,
            "int8x8": true
        },
        "sparsity_tflm": false,
        "tflm_model_quantization": "FLOAT"
    },
    "name": "TEST_MODEL",
    "output_dir": "mtb_ml_gen",
    "target": "PSoC6",
    "toolsPackage": "ModusToolbox Machine Learning Pack 2.0.0"
    "validation": {
        "feat_col_count": 784,
        "feat_col_first": 1,
        "input_format": "JPEG",
        "input_type": "ML",
        "max_samples": 1000,
        "path": "test_data/test_data.csv",
        "quantization": {
            "float32": true,
            "int16x16": true,
            "int16x8": true,
            "int8x8": true
        },
        "target": {
            "target_quantization": "int8x8"
        },
        "target_col_count": 1,
        "target_col_first": 0
    }
}
```

The JSON breakdown table gives a description of each object and can be used to edit the example JSON file. All fields need to be included, even if not used.

JSON Breakdown:

Objects	Description	Value
calibration_data		
active_state	Enables or disables model calibration section	true false
feat_col_count	Number of feature columns	Integer
feat_col_first	Index of first feature column	Integer
input_calibration_type	Input data type	NPZ FOLDER ML
input_format	Only used if input_calibration_type=ML. Format type for the input data.	JPEG CSV
path	Path to calibration data	String path
target_col_count	Number of target columns	Integer
target_col_first	Index of first target column	Integer
Filetype		
filetype	Identifies the file type	modustoolbox-ml-configurator
Model		
framework	Inference engine	KERAS TFLITE

Objects	Description	Value
optimization_ifx	Enables or disables the "advanced scratch memory optimization" setting for "ifx" inference engine. • "SIZE" (default) – enables "advanced scratch memory optimization" setting • "SPEED" – disables "advanced scratch memory optimization" setting	SIZE SPEED
optimization_tflm	Enables or disables the "tflm_interpreter_less" setting for "tflm" inference engine. • false (default) – disables "tflm interpreter less" setting, • true – enables "tflm interpreter less" setting.	true false
path	Path to model file. If the path is relative, it is evaluated with respect to the location of the configuration file.	String path
sparsity_tflm	Enable use of a memory-efficient packed format for any sparse weights present in the model	true false
tflm_model_ quantization	Define if the tflite model was created using fixed-point or floating point.	FLOAT INT8X8
model.quantization		
float32	Generate source for quantization type	true false
int16x16	Generate source for quantization type	
int16x8	Generate source for quantization type	
int8x8	Generate source for quantization type	
name		
name	Project name (used as output file prefix)	User defined
output_dir		
output_dir	Output directory. If the path is relative, it is evaluated with respect to the location of the configuration file.	String path
target		
target	Specifies the device, always set to PSoC6	PSoC6
validation		
feat_col_count	Number of feature columns	Integer
feat_col_first	Index of first feature column (0-based)	Integer
input_format	Format type for the input data	CSV BIN JPEG
input_type	Input data type, can be PRNG, NPZ, FOLDER, or ML	PRNG NPZ FOLDER ML
max_samples	Number of random samples to generate	Integer [0~1000]
path	Path to validation input data file or folder. If the path is relative, it is evaluated with respect to the location of the configuration file.	Path
validation.quantization		
float32	Validate quantized model type (true or false)	true false
int16x16	Validate quantized model type (true or false)	

Objects	Description	Value
int16x8	Validate quantized model type (true or false)	
int8x8	Validate quantized model type (true or false)	
validation.target		
target_quantization	Which quantization type to use for "Evaluate on Target"	int8x8 int16x8 int16x16 float32
target_col_count	Number of feature columns	Integer
target_col_first	Index of first target column (0-based).	Integer
app		
app	Application type. Currently, it only accepts “ML”.	ML
formatVersion		
formatVersion	File format version. The version described here is 3. The tool should refuse to load files with a higher version that the tool’s maximum supported version. The file format version is incremented when a breaking change is made to the format. Adding an optional field is not a breaking change. Changing the type of an existing field is a breaking change.	3
lastSavedWith		
lastSavedWidth	Tool that was last saved with. Currently, it only accepts “ML Configurator”	ML Configurator
lastSavedWithVersion		
lastSavedWidthVersion	Tool version that was last saved with. The version descried here is 2.0.0. The tool version is incremented when a new release is publicly available.	2.0.0
toolsPackage		
toolsPackage	Name of the package. Currently, it only accepts “ModusToolbox Machine Learning Pack 2.0.0”	ModusToolbox Machine Learning Pack 2.0.0

5 ModusToolbox™ ML embedded libraries

For the ML 2.0 solution, there are three middleware assets deployed as ModusToolbox™ libraries:

- ML Infineon inference engine
- TFLM inference engine
- ML middleware

5.1 ML Infineon inference engine library

5.1.1 Adding the library

The ML-inference engine library is available as an ModusToolbox™ asset. Use the following GitHub link:

https://github.com/infineon/ml-inference

You can add a dependency file (MTB format) under the deps folder or use the Library Manager to add it in your project. It is available under **Library > Machine Learning > ml-inference**

In the Makefile of the project, you need to define the quantization to be deployed. Note that you can only choose one type of quantization. In the `COMPONENTS` parameter, add one of the following:

- `ML_FLOAT32`: use 32-bit floating-point for the weights and output/input data
- `ML_INT16x16`: use 16-bit fixed-point for the weights and output/input data
- `ML_INT16x8`: use 16-bit fixed-point for the output/input data and 8-bit for the weights
- `ML_INT8x8`: use 8-bit fixed-point for the weights and output/input data

5.1.2 Using the library

There are four steps to use the ML-inference engine library.

5.1.2.1 Step 1: Get required memory for the inference engine

One of the data arrays generated by the ML Configurator Tool contains the model parameters. If using binary file, it is placed in the *NN_model_prms.bin*. If using C array header, it is placed in *NN_model_all.h* > *NN_model_prms_bin[]*.

Use the `Cy_ML_Model_Parse()` function to extract information from the NN model. It tells how much memory is required by the persistent and scratch memory, the input layer size and the number inference classification output size.

5.1.2.2 Step 2: Allocate memory

Once you know how much memory is required by the persistent and scratch memory, allocate it in your application. Here is an example in C on how to perform Steps 1 and 2.

```
#include "NN_model_all.h"
...
cy_stc_ml_model_info_t model_xx_info;

count = Cy_ML_Model_Parse(NN_model_prms_bin, &model_xx_info);
if (count > 0) /* Model parsing successful */
{
   persistent_mem = (char*) malloc(model_xx_info.persistent_mem*sizeof(char));
   scratch_mem = (char*) malloc(model_xx_info.scratch_mem*sizeof(char));
}
```

Alternatively, you can allocate it statically based on the MACROs provided by the ML middleware. Refer to the ML middleware library section.

5.1.2.3 Step 3: Initialize model and get model container/object

After allocating memory, you can initialize the interference engine by providing pointers to the memory and the model weights. Here is an example in C using float quantization:

```
#include "NN_model_all.h"
...
void *model_xx_obj;

result = Cy_ML_Model_Init(&model_xx_obj,       // NN model data container pointer
                          &NN_model_flt_bin,   // NN model parameter buffer pointer
                          persistent_mem,      // Pointer to allocated persistent
                          scratch_mem,         // Pointer to allocated scratch mem
                          &model_xx_info);     // Pointer to model info structure
```

5.1.2.4 Step 4: Run the inference engine

The last step is to run the inference engine. In this step, the input data can come from sensors or from the regression data. Here is an example in C using float quantization.

```
result = Cy_ML_Model_Inference(&model_xx_obj, // NN model data container pointer
                               in_buffer,      // Input buffer
                               out_buffer,     // Output buffer
                               NULL);          // Not used in floating-point
```

Note: *The last argument of the function above is only used in fixed-point. This argument is the pointer to input data fixed-point Q factor. The function overwrites this argument with the output data fixed-point Q factor.*

5.1.3 Verifying the inference

The ML inference engine library integrates a profiling mechanism to obtain the number of cycles the library takes to execute. To make this work, the application needs to implement the following function:

```
int Cy_ML_Profile_Get_Tsc(uint32_t *val)
```

This function needs to return a counter value that shows how many CPU cycles have passed. One way to implement this is to run a timer continuously that triggers an interrupt every second to increment a variable to represent how many seconds passed. On calling `Cy_ML_Profile_Get_Tsc`, it gets the number of seconds passed and combine with the current counter. Then it translates the result in CPU cycles.

To actually enable the profiling from the ML-inference engine library, a few functions need to be called. Here is the flow:

1. Call `Cy_ML_Profile_Init()` to initialize the profiling or disable all profiling features enabled.
2. Call `Cy_ML_Profile_Print()` after feeding all the regression data.

An example using the regression data generated by the ML Configurator tool and the profiling integrated in the ML-inference engine library is available in this link (this example uses the ML middleware):

https://github.com/infineon/mtb-example-ml-profiler

There are a few options for printing profiling and debugging information based on the configuration provided to the `Cy_ML_Profile_Init()` function, as shown in the following list:

Constants	Description
`CY_ML_PROFILE_LAYER`	Enable layer profiling. It prints general information of each layer of the ML model, such as average cycle, peak cycle and peak frame.
`CY_ML_PROFILE_MODEL`	Enable the ML model profiling. It prints general information about the ML model, such as average cycle, peak cycle and peak frame.
`CY_ML_PROFILE_FRAME`	Enable the per frame profiling. It prints general information for each frame, such as number of cycles per frame.
`CY_ML_LOG_LAYER_OUTPUT`	Enable the layer output logging. It prints the output values generated by each layer.
`CY_ML_LOG_MODEL_OUTPUT`	Enable the model output logging. It prints the output values generated by the inference engine.

Note: *Although you can derive any of the combinations above by ORing them, certain combinations might affect the profiling results due to additional printing activity. The recommended profile settings that work without affecting the results are listed below.*

Constants	Description
`CY_ML_PROFILE_DISABLE`	`0`
`CY_ML_PROFILE_ENABLE_MODEL`	`CY_ML_PROFILE_MODEL`
`CY_ML_PROFILE_ENABLE_LAYER`	`CY_ML_PROFILE_LAYER`
`CY_ML_PROFILE_ENABLE_MODEL_PER_FRAME`	`CY_ML_PROFILE_MODEL | CY_ML_PROFILE_FRAME`
`CY_ML_PROFILE_ENABLE_LAYER_PER_FRAME`	`CY_ML_LOG_LAYER_OUTPUT | CY_ML_PROFILE_FRAME`
`CY_ML_LOG_ENABLE_MODEL_LOG`	`CY_ML_LOG_MODEL_OUTPUT`

When using the regression data, the output of the inference engine needs to be compared against a reference. If the result matches between the regression data and the inference engine output, it contributes to the final calculation of the model accuracy, which is calculated as:

$$Accuracy = \frac{Number\ of\ successful\ results}{Number\ of\ interactions}$$

5.2 TFLM inference engine library

The TFLM library is runs machine learning models on Infineon microcontrollers. The TFLM library is available as a ModusToolbox™ asset. Use the following GitHub link:

https://github.com/infineon/ml-tflite-micro

You can add a dependency file (mtb format) under the *deps* folder or use the Library Manager to add it in your project. It is available under **Library > Machine Learning > ml-tflite-micro.**

This document does not provide details on how to use the TFLM APIs written in C++. It is recommended to use our machine learning abstraction middleware, explained in the ML middleware library section. For more general information about the TFLM, including examples and documentation, refer to this link:

https://www.tensorflow.org/lite/microcontrollers

Infineon provides the following methods to deploy machine learning models using the TFLM inference engine:

- **TFLM runtime interpreter**: Uses an interpreter to process a machine learning model deployed as binary data. This allows easy updates on the deployed model or the need to inference multiple models in the application.
- **TFLM interpreter-less**: Does not require a run-time interpreter; instead uses pre-generated code to execute the inference. This allows smaller binary and less overhead on inference execution.

In both cases, this version of the library provides two types of quantization – *floating point* and *8-bit integer*. To use this library, the following `COMPONENTS` and `DEFINES` are required:

- If using TFLM runtime interpreter:
  ```
  DEFINES+=TF_LITE_STATIC_MEMORY
  COMPONENTS+=ML_TFLM_INTERPRETER IFX_CMSIS_NN
  ```
- If using TFLM interpreter-less:
  ```
  DEFINES+=TF_LITE_STATIC_MEMORY TF_LITE_MICRO_USE_OFFLINE_OP_USER_DATA
  COMPONENTS+=ML_TFLM_INTERPRETER_LESS IFX_CMSIS_NN
  ```
- If using floating point:
  ```
  COMPONENTS+=ML_FLOAT32
  ```
- If using 8-bit integer:
  ```
  COMPONENTS+=ML_INT8x8
  ```

5.3 ML middleware library

This library works as an abstraction layer to the Infineon and TFLM inference engine libraries. It implements all the steps described in the previous sections in two main functions:

```
mtb_ml_model_init() : to initialize the model
mtb_ml_model_run()  : to run the inference engine
```

The library also provides helper MACROs to include the model files generated by the ML Configurator tool. For example, if a model was generated using the output file prefix "test_model", you can use:

```
#include MTB_ML_INCLUDE_MODEL_FILE(test_model)          // Model file
#include MTB_ML_INCLUDE_MODEL_X_DATA_FILE(test_model)    // Regression data X
#include MTB_ML_INCLUDE_MODEX_Y_DATA_FILE(test_model)    // Regression data Y
```

The `mtb_ml_model_init()` function has an option to skip the parsing and memory allocation by providing pointers for the scratch and persistent memories (for Infineon inference engine only). The application can use the following helper MACROs to know how much memory to allocate:

```
// Infineon inference engine
MTB_ML_MODEL_SCRATCH_MEM_SIZE(test_model)
MTB_ML_MODEL_PERSISTENT_MEM_SIZE(test_model)
// TFLM inference engine
MTB_ML_MODEL_ARENA_SIZE(test_model)
```

To access the data arrays generated for the model, use the following MACROs:

```
MTB_ML_MODEL_NAME_STR(test_model)    // String for the model name
MTB_ML_MODEL_BIN_DATA(test_model)    // Populates the model binary structure
MTB_ML_MODEL_X_DATA_BIN(test_model)  // Regression data input array
MTB_ML_MODEL_Y_DATA_BIN(test_model)  // Regression data output array
```

The following table shows the steps for the two initialization methods:

Method 1: Using internal memory allocation

```
mtb_ml_model_t *model_object;

/* NN model data */
mtb_ml_model_bin_t model_bin = {MTB_ML_MODEL_BIN_DATA(model_test)};

/* Initialize the model */
mtb_ml_model_init(&model_bin, NULL, &model_object);
```

Method 2: Using external memory allocation

```
mtb_ml_model_t *model_object;

#if defined(COMPONENT_MTB_ML_IFX)
/* Allocate the persistent and scratch memories */
   uint8_t persistent_mem[MTB_ML_MODEL_PERSISTENT_MEM_SIZE(test_model)];
   uint8_t scratch_mem[MTB_ML_MODEL_SCRATCH_MEM_SIZE(test_model)];
   mtb_ml_model_buffer_t mem_buf = {persistent_mem, scratch_mem};
#elif defined(COMPONENT_ML_TFLM_INTERPRETER))
   uint8_t tensor_arena[MTB_ML_MODEL_ARENA_SIZE(test_model)];
   mtb_ml_model_buffer_t mem_buf = {tensor_arena,
                                    MTB_ML_MODEL_ARENA_SIZE(test_model)};
#endif

/* NN model data */
mtb_ml_model_bin_t model_bin = {MTB_ML_MODEL_BIN_DATA(model_test)};

/* Initialize the model */
mtb_ml_model_init(&model_bin, &mem_buf, &model_object);
```

Note: *Using `ML_TFLM_INTERPRETER_LESS`, only method 1 applies.*

This library also comes with a set of APIs to handle streaming data from the ML Configurator tool. It handles the UART connection and what profiling and debugging features to enable. There are two main functions to call:

```
mtb_ml_stream_init() : execute mtb_ml_model_profile_config() and
mtb_ml_model_init()

mtb_ml_stream_task() : execute mtb_ml_model_run() and mtb_ml_model_log()
```

The following link provides an example using the streaming feature of this solution:

https://github.com/infineon/mtb-example-ml-profiler

This library also comes with helper functions that:

- convert floating-point to fixed-point and vice versa
- quantize the inputs
- dequantize outputs
- return the index of the maximum value in an array

These functions leverage some of the Arm DSP instructions to improve performance. To enable DSP, add the following component/define to the Makefile:

```
COMPONENTS+=CMSIS_DSP
DEFINES+=MTB_ML_HAVING_CMSIS_DSP
```

6 ModusToolbox™ ML core tools

The ML core tools are the backend utilities which the ML Configurator tool interacts with to perform a variety of features:

- Parsing a JSON file for details on the model, data and configuration
- Creating random data for regression
- Alternative to using random data, imported data can be created [3]
- Loading pretrained Keras H5 model and parsing them for ModusToolbox™ ML relevant information
- Performing model reference evaluation on the specified dataset to determine a base floating-point collection for reference metrics
- Converting the pretrained or trained model to a library C code for embedding and use on the PSoC™ 6 MCU and as part of the ModusToolbox™ ecosystem
- Saving log files for output and debugging purposes during and after their operation
- Regression data generation, tests and cross-domain verification
- Stream regression data to the target device

This tool is written using Python scripts and released as an executable, which is available with the Machine Learning pack. It is meant to be used with the ML Configurator only.

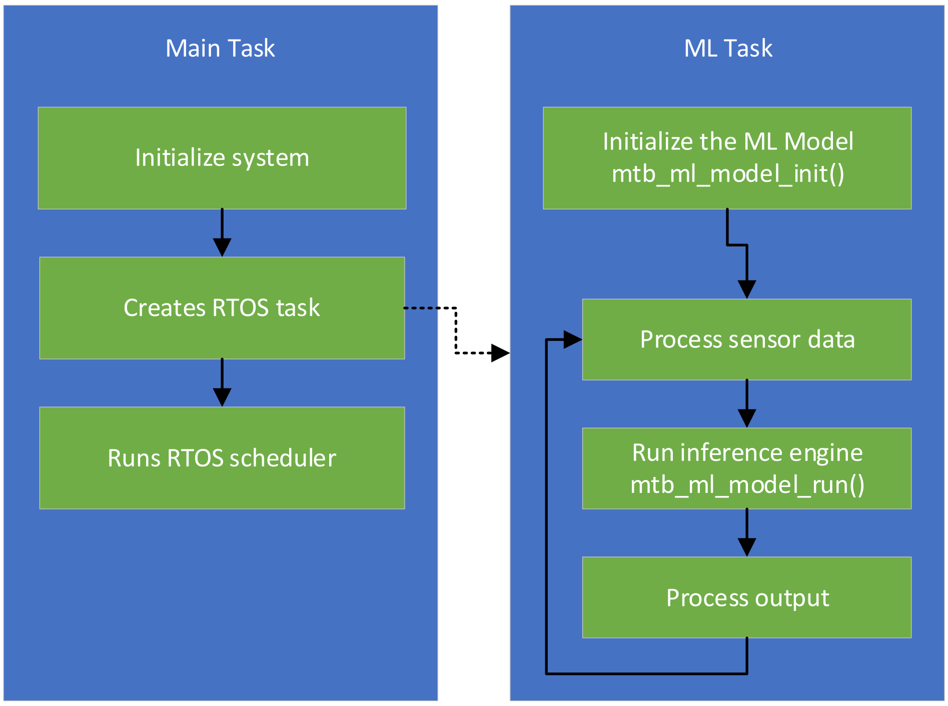
[3] This reads data from files structure in one of the ModusToolbox™ ML formats.

7 ModusToolbox™ ML integration

The easiest way to run the ML solution with PSoC™ 6 MCUs is by leveraging the code available in the ML code examples.

7.1 Using an RTOS

If using an RTOS, the most effective way to integrate the inference engine in your project is to use the ML middleware library to wrap all interactions with the ML Inference Engine library. You can have a single task or multiple tasks to handle different functions of the application. The following example shows a task handling all related machine learning functions.



As the RTOS typically handles memory management, you need to ensure that the RTOS heap has enough memory for the scratch and persistent memory required by the inference engine, which can be quite large depending on the model being executed.

7.2 Handling inputs and outputs

The ML inference engine can use both floating-point and fixed-point inputs. One good practice before feeding a NN is to normalize the input. There are different types of normalization in statistics. In this section, we refer to the min-max feature scaling type. If the input data is not normalized in the embedded firmware due to limited hardware resources, the training framework tool also cannot normalize the input data. The input data fed to the inference engine and the training algorithm need to match.

If using floating-point as input, the normalization is straight forward. It depends on the maximum and minimum value of the sensor, and the normalization range. Usually the normalization is between [0,1] or [-1,1]. Use the following formula to normalize:

$$NN\ input\ data = N_{MAX} - (S_{MAX} - Sensor\ Data) \times \frac{N_{MAX} - N_{MIN}}{S_{MAX} - S_{MIN}}$$

Where S_{MAX} and S_{MIN} are the maximum and minimum value the sensor can produce, and N_{MAX} and N_{MIN} are the normalization range. If the sensor data needs to be filtered, or subjected to some sort of processing (besides normalization), it must mimic the steps of the training.

When dealing with fixed-point inputs, the data needs to be in the fixed-point (integer) format where fractional bits are specified using the Q format. As an example, a signed 16-bit number can be represented in Q15 format for maximum precision if the number is in [-1, 1] range, the most significant bit is the sign bit and the other bits are fractional bits. You can also specify the number of integer bits such as Q14 to use 1 sign bit and 1 integer bit. Here are some examples.

Q format	Max value (Integer)	Min Value (Integer)	Max value (Float)	Min value (Float)
q15 (16-bit)	32767 (0x7FFF)	-32768 (0x8000)	$+1.0 - 2^{-15}$	-1.0
q7 (8-bit)	127 (0x7F)	-128 (0x80)	$+1.0 - 2^{-7}$	-1.0
q14 (16-bit)	32767 (0x7FFF)	-32768 (0x8000)	$+2.0 - 2^{-14}$	-2.0

To normalize the input sensor data to the desire Q format, use the following formula.

$$NN\ input\ data = \frac{(2^{Q+1} - 1) \times (Sensor\ Data - S_{MAX}) + (2^{Q} - 1) \times (S_{MAX} - S_{MIN})}{S_{MAX} - S_{MIN}}$$

If you want to convert the NN input data to unsigned integer, sum the result of the above formula with 2^{Q}.

7.2.1 Using Infineon Inference Engine

When feeding the inference engine with the function `Cy_ML_Model_Inference()`, you need to provide which Q format you are using as argument. The function then returns the Q format of the output buffer. If you need to convert the result to floating point, use the following code:

```
q_format = 15; // Use q15 format for the input data
result = Cy_ML_Model_Inference(&model_xx_obj, // NN model data container pointer
                               in_buffer,     // Input buffer
                               out_buffer,    // Output buffer
                               &q_format);    // Pointer to Q format
float q_norm = 1.0f / (float) (1 << q_format);
for (int i = 0; i < output_size; i++)
{
    float_buffer[i] = out_buffer[i] * q_form;
}
```

Alternatively, you can use the helper function `mtb_ml_utils_int_to_flt()` that comes with the ML Middleware library.

When feeding the inference engine with the function `mtb_ml_model_run()`, you can provide which Q format you are using through the function `mtb_ml_model_set_input_q_fraction_bits()`. And you can use the function `mtb_ml_model_get_output()` to acquire the Q format output.

Note: *This function is only used in the beginning during the initialization of the model.*

You only need to use the Q format functions if using fixed-point interference engine.

7.2.2 Using TFLM Inference Engine

The TFLM inference engine does not require Q format when using the integer quantization. Internally, it uses asymmetric quantization, which uses a zero-point and scaler variables. The following equation applies:

$$FloatValue = (IntegerValue - ZeroPoint) * Scaler$$

The zero-point and scaler values are defined during the model calibration, which requires the user to provide some calibration data when generating the model files. The input and output of a TFLM model might have different values for the zero-point and scaler.

To quantize the results from float to integer, you can use the `mtb_ml_utils_model_quantize()` function, which converts a given input float array to integer using the input zero-point and scaler from the model. This information is stored internally in the `mtb_ml_model_t` structure.

To de-quantize the results to some meaningful value, you can use the `mtb_ml_utils_model_dequantize()` function, which provides the float representation of the model's output by using the output zero-point and scaler from the model.

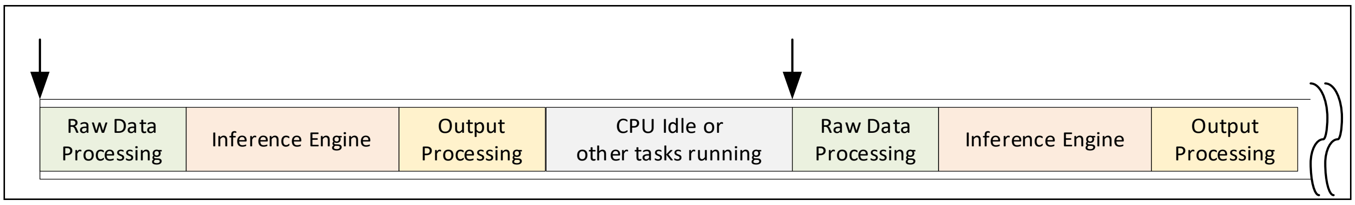
For TFLM inference using int8x8 quantization, the model calibration data plays an important role on the accuracy of the model. If the data provided is not representative enough to stimulate the network to get good estimates of activation value statistics (min/max value range), it can result in poor network performance. In extreme cases, assertions in the reference TFLITE interpreter built in to TensorFlow may fail, causing TensorFlow to abort without providing a meaningful error-status return.

7.3 Memory and CPU requirements

When using the ML inference engines (Infineon and TFLM), different types of memory blocks are required and special attention is required to allocate them. The following table provides a summary of what is required at a minimum:

Memory block	**Inference Engine**	**Location**	**Description**
Model weights	Infineon / TFLM	Flash	Contains the model weights. It can be very large, requiring to be stored in the external memory.
Model parameters	Infineon	Flash	Contains the model parameters. Usually only a few bytes.
Scratch Memory	Infineon	SRAM	Temporary memory that can be discarded after each frame is processed.
Persistent Memory	Infineon	SRAM	Temporary memory that persists on each frame and used by the next frame.
Tensor arena	TFLM	SRAM	A combination of scratch and persistent memory, handle automatically by the TLFM inference engine
Input Buffer	Infineon / TFLM	SRAM	Buffer storing the input data to the NN. It depends on how many nodes in the input layer. Some types of NN have a huge number of inputs, for example, when processing images.
Output Buffer	Infineon / TFLM	SRAM	Buffer storing the output data to the NN. It depends on how many nodes in the output layer. There is a hard limit of 64 nodes in the output layer.
Regression Input Data	Infineon / TFLM	Flash	Only used for regression. Usually it is very large and require to be stored in external flash. With the ML 2.0 solution, we recommend to stream regression data.
Regression Output Data	Infineon / TFLM	Flash	Only used for regression. Usually it is very large and require to be stored in external flash. With the ML 2.0 solution, we recommend to stream regression data.

In terms of CPU requirements, it is important to understand what is the sample rate, the sensor processing time, the inference engine time and output processing time. The following graphic shows an example how the CPU handles the bandwidth on every task.



The total amount of time it takes to do all the processing around the data needs to be smaller than the period of the sampling rate. Choosing a different quantization for the inference engine can drastically reduce the cycles time, at the cost of using more memory. See the Evaluating a Model section.

Revision history

Revision	Date	Description
**	2021-03-12	New tool.
*A	2021-04-29	Added information for the dataset structures. Updated table of files generated by the ML Configurator.
*B	2021-08-18	Updated instructions based on ModusToolbox™ ML 1.2 solution.
*C	2021-09-14	Add information about the "Advanced scratch memory optimization" check box in the ML Configurator. Clarified the note about the array with the model weights, and how to add to internal flash and external memory.
*D	2022-07-15	Updated instructions based on ModusToolbox™ ML 2.0 solution. Added support for TensorFlow Lite for Microcontrollers. Added more information to the ml-coretools section. Added list of layers/operators coverage supported by TFLite and Keras formats Added JSON field parameters table for MTBML format Updated list of files generated by the ML Configurator
*E	2022-08-23	Updated getting started instructions. Added note about TFLM inference using int8x8 quantization. Added information about the sparsity feature.
*F	2022-10-10	Added instructions to install QEMU Added note about warning messages when generating/validating the model
*G	2022-11-10	Updated links to Infineon website to download/install the pack. Updated QEMU installation instructions.
*H	2023-05-23	Explained the usage of the mtb_ml_utils_quantize() function. Removed support for 2D and 3D feature data for CSV format. Added instructions to enable CMSIS_DSP component.

Edition 2023-05-23

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-32590 Rev. *H