Chapter 4: ModusToolbox™ Machine Learning Configurator

After completing this chapter, you will understand how to import, optimize, and deploy ML models onto an Infineon PSoC™ 6 MCU with the help of the Machine Learning Technology Pack.

Table of contents

Document conventions

| Convention	Usage	Example
Courier New	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

4.1　　Overview

In this chapter, we are going to learn about the back end of the ML process. That is, we'll start with pre-trained neural network (NN) models and will learn how to optimize, validate, deploy, and integrate them into a PSoC™ MCU using the ModusToolbox™ ML solution. The pre-trained model can be obtained in various ways such as using Imagimob Studio.
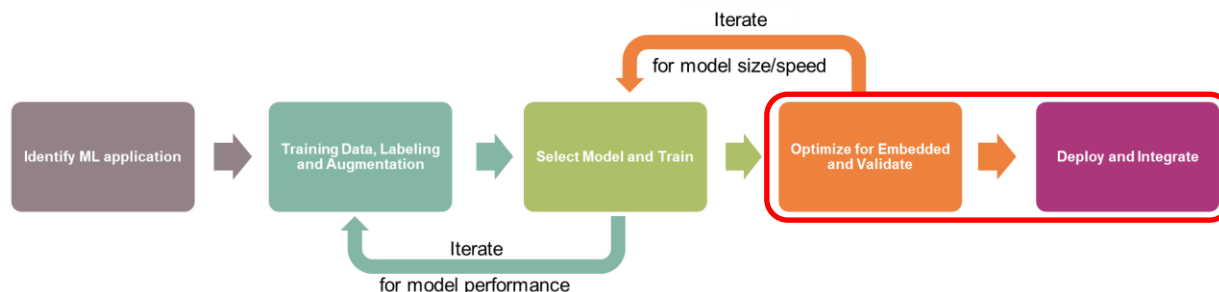
4.2　　ModusToolbox™ Machine Learning

In addition to Imagimob Studio, the ModusToolbox™ Machine Learning solution is a set of tools, middleware libraries and code examples that fit into the ModusToolbox™ ecosystem as shown here:
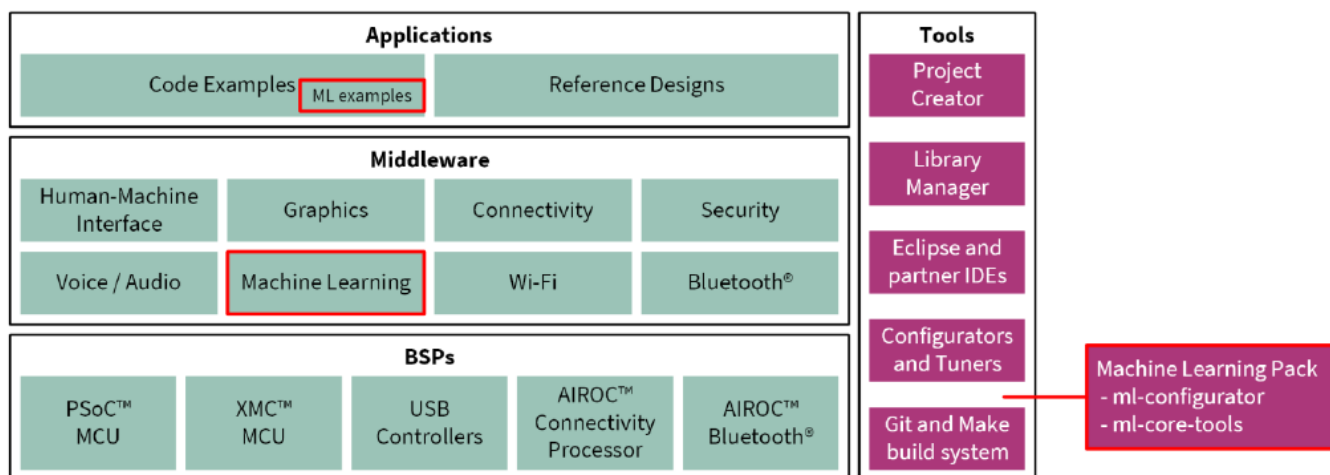
The four major components of the solution are:

- 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 tools: A collection of Python scripts (provided as executables) to perform parsing, regression and conversion of models.
- ML library: This tool includes helper functions and inference engine libraries for Infineon MCUs.
- ML code examples: A collection of applications on how to use the ML solution.

The following figure shows how these four parts interconnect with each other:
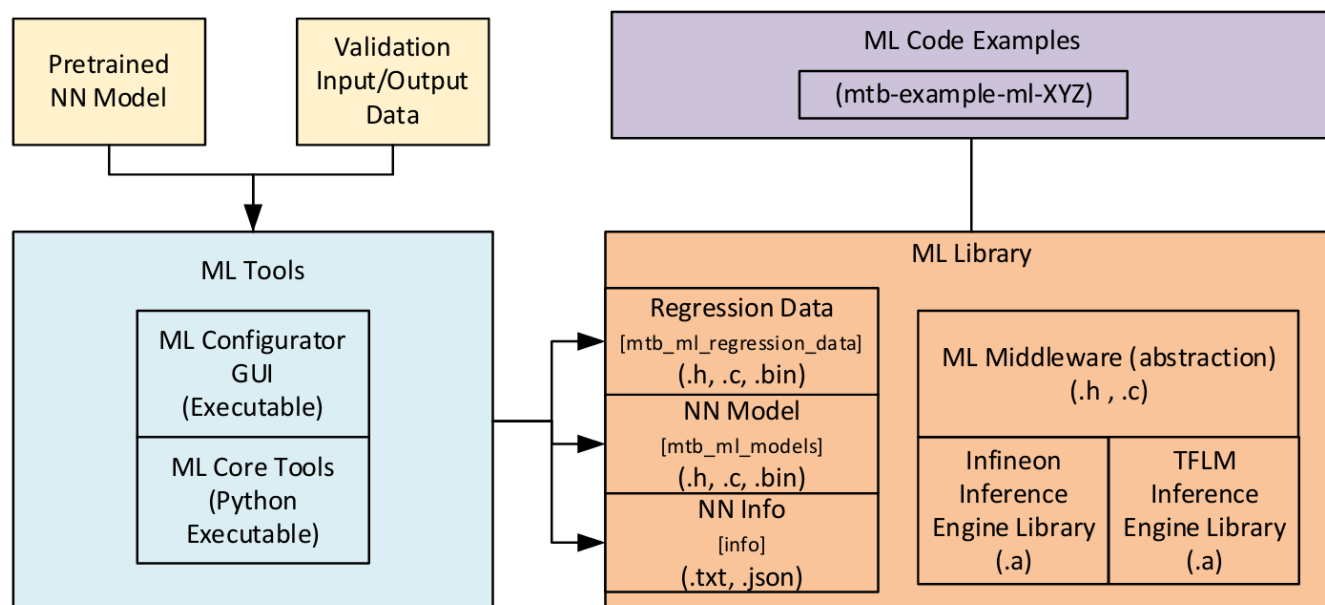
The ML core tools and configurator are provided in a Machine Learning Pack that can be installed alongside the other ModusToolbox™ tools. It can be found at:

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

The ML Configurator also requires "QEMU" for model evaluation. Instructions for downloading, installing, and configuring QEMU can be found in the Software requirements section of the ModusToolbox™ Machine Learning user guide.

The ModusToolbox™ Machine Learning solution currently supports two model formats:

- Keras-H5 (.h5)
- TensorFlow Lite (.tflite)

The ModusToolbox™ Machine Learning solution currently supports two inference engines:

- Infineon (IFX) inference engine   (Keras-H5 models)
- TensorFlow Lite for Microcontrollers (TFLM) inference engine (Keras-H5 and TensorFlow Lite models)

If you use a Keras-H5 model with the TFLM inference engine and you want to do fixed-point quantization (e.g. int8x8), you must also provide calibration data.

The TFLM inference engine can run using a runtime interpreter or it can run without an interpreter. The interpreter is used to process a model deployed as binary data. This option allows easy updates to the deployed model and allows inferencing multiple models within the application. The interpreter-less option uses pre-generated code to execute the inference. That results in a smaller binary image and less overhead when the inference is executed.

Refer to the ModusToolbox™ Machine Learning user guide for additional details about the inference engines and their differences.

4.3 Generate and optimize the model for embedded

Once you have a pre-trained model in either Keras-H5 or TensorFlow Lite format, the next step is to generate a model that can be used by the MCU. In addition to formatting, optimization allows for the best possible performance on an embedded system.

Remember that our goal is to be able to perform machine learning operations on the edge device rather than relying on the cloud. Using lots of high-powered machines in the cloud is great for training the model, but the edge device has more limited capabilities. Therefore, optimizing for embedded involves quantizing the model to use data types that will lead to reduced memory usage and faster calculations without unacceptably degrading the accuracy of the predictions. The ModusToolbox™ ML tools will generate four different versions of the model for the Infineon inference engine and two for the TFLM inference engine that you can compare for size, performance, and accuracy:

Quantization Type	Input Data	Weights	Inference Engine
int8x8	8-bit fixed point	8-bit fixed point	IFX, TFLM
int16x8	16-bit fixed point	8-bit fixed point	IFX
int16x16	16-bit fixed point	16-bit fixed point	IFX
float	floating point	floating point	IFX, TFLM

When you generate the MCU source code from the models, a report will tell you how many CPU cycles it takes for each model to predict a result for an input value and how much memory is required. The memory report provides both model memory (typically stored in Flash) and scratch memory (stored in RAM).

As you would expect, the int8x8 model uses the least memory but on a PSoC™ 6, the int16x16 actually uses the fewest CPU cycles. This is because 16-bit operations are more computationally efficient when compared to 8-bit operations on PSoC™ 6.

4.4 Validate the model

Once we have the different models, we need to validate them on test data to see how they perform. The model you choose to use on your final design will depend on this validation. As such, you want to use the fastest and smallest model that still provides sufficient performance.

The validation for all models can be done on your computer and compared to the results of the starting model to see how each one performs against the test data. In addition, a special profiling code example can be used to test the model on the target MCU. This is a final validation that the model operates as expected on the target device.

As you will see, there are different ways to provide test data for validation ranging from randomly generated values to specific labeled data sets that you provide.

4.5   Machine learning configurator

The optimization and validation steps of the ML flow described above are enabled in ModusToolbox™ using the Machine Learning Configurator. Once the ML Pack has been installed, the tool can be launched in the following ways:

- From the Eclipse Quick Panel
- Using the command line from an application (`make ml-configurator`)
- From the tool's installation directory *(~/ModusToolbox/packs/ModusToolbox-Machine-Learning-Pack/tools/ml-configurator)*.
- By entering `ml-configurator` in the Windows search bar

The settings you make in the configurator are saved to your application in a file called *design.mtbml*. If you open the tool without supplying a file, it will ask you to open or create a new file. As an example, if you open the ML Configurator for the Gesture Classification code example, you will see a screen like this:
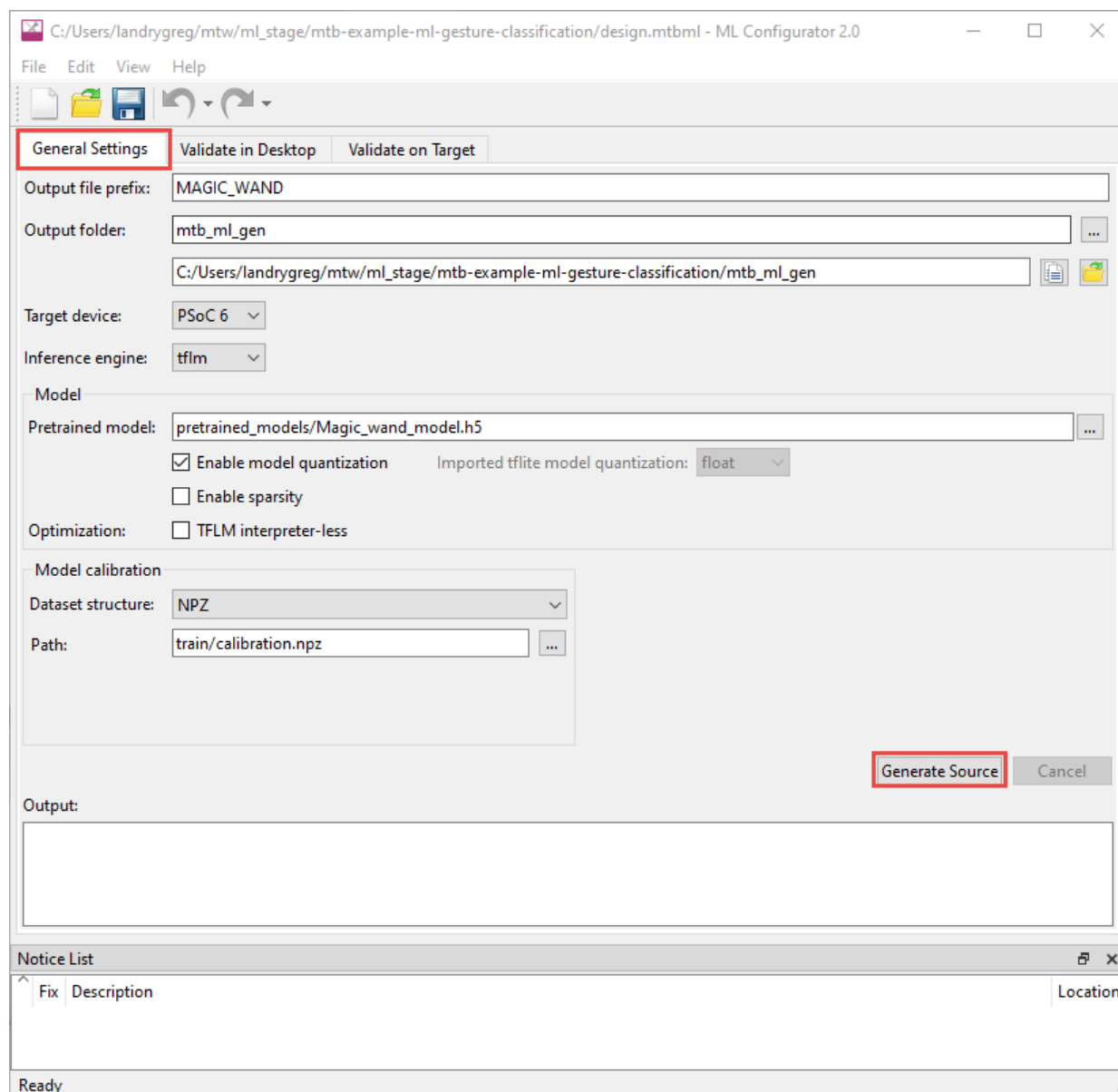
4.5.1   General Settings tab

At the top of the General Settings tab, you can specify the name and location for the generated code (**Output file prefix** and **Output folder**), the target device and inference engine to use. In the Model section, you supply the path to your pre-trained model and its format. After you have done that, you can generate the model by clicking the **Generate Source** button. That button generates the models and places them in the output directory specified under a subdirectory called *mtb_ml_models*. The default location will be *mtb_ml_gen/mtb_ml_modes*. One of these models will ultimately be included in the application.

Note: *If you change the **Output file prefix**, **Output folder**, or **Inference engine**, you also need to change the* `NN_MODEL_NAME`, `NN_MODEL_FOLDER`, *or* `NN_INFERENCE_ENGINE` *variable, respectively, in the Makefile. The use of these variables will be explained later in [the Firmware section.](#)*

Note: *The **Generate Source** button also generates the regression data used for validation. Therefore, the settings on the **Validate in Desktop** tab must be correct for your model, otherwise model generation will fail.*

The options shown in the model and optimization sections and their usage will depend on the inference engine that you select. See the ML Configurator user guide for details on each option.

Once the files have been generated, the Output panel will show information about cycles required and memory used for each model. For example:

```
Output:

[INFO] Model conversion complete.

[INFO]        Model   |  Cycles  |  Model Memory  | Scratch Memory  |
[INFO]      Floating  |  91000   |   52044 bytes  |    128 bytes    |
[INFO]      Int16x16  |  50584   |   26034 bytes  |    148 bytes    |
[INFO]      Int16x8   |  70792   |   13029 bytes  |    148 bytes    |
[INFO]       Int8x8   |  58172   |   13029 bytes  |    116 bytes    |
[INFO] Model conversion is completed.
[INFO] Converting validation data for the reference evaluation...
```

Note: *You will have to scroll back in the output window to see this information.*

4.5.2 Validate in Desktop tab

Before deciding which model to use in the application, you will want to run validation to see how the accuracy compares between models. The **Validate in desktop** tab is used to compare the various models running on your computer against random test data or other test data that you provide. The window looks like this:
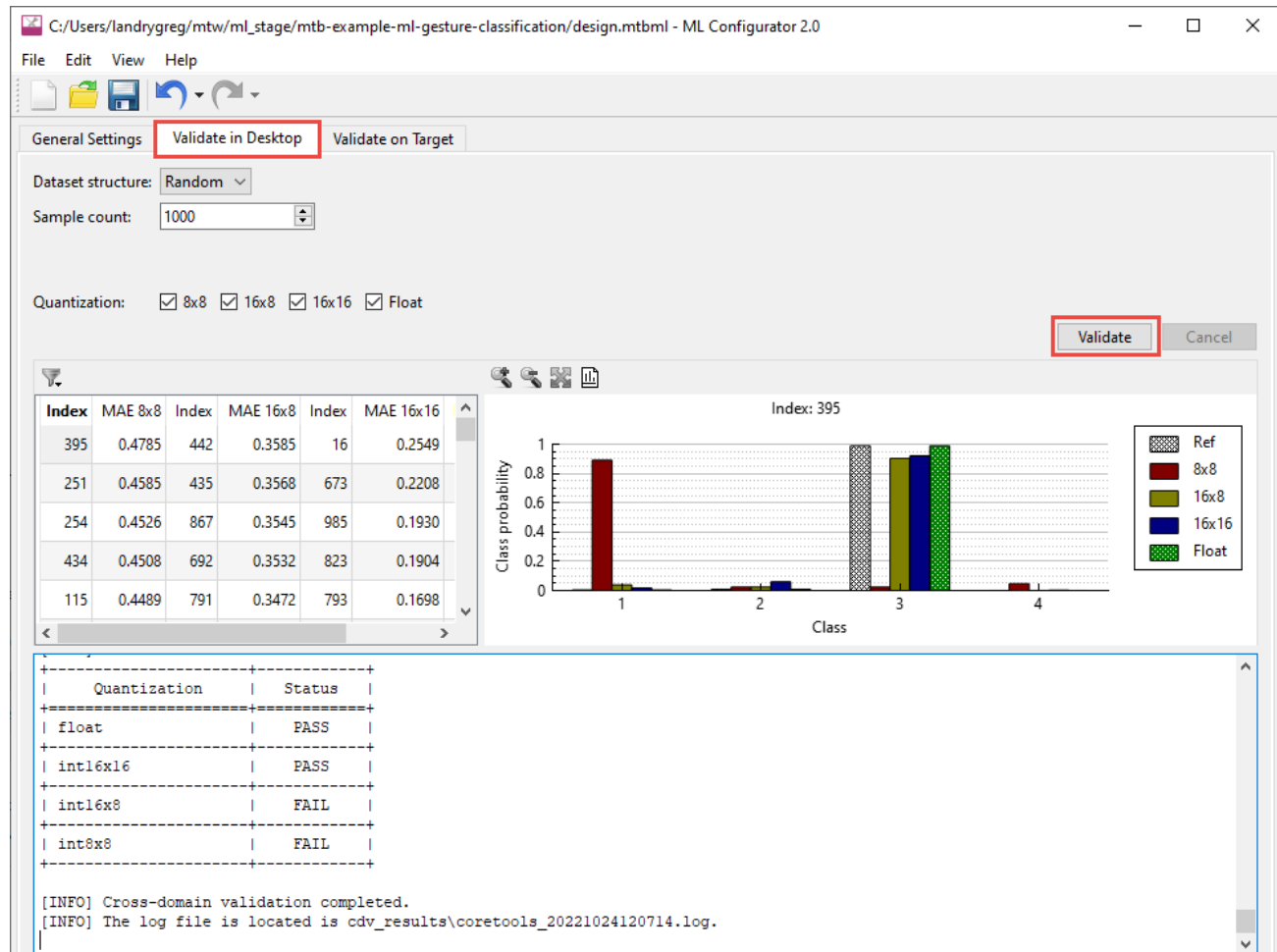
For the data set, you can choose one of four options:

- **Random:** A set of samples is automatically generated. You can specify how many samples to use.
- **NPZ:** Samples are contained in NumPy arrays in zipped format.
- **Folder:** A folder is specified which contains a sub-folder for each class.
  Each subfolder contains CSV files or JPEG images that are a part of that class.
- **ML:** A CSV file containing only numeric data with no header and no sample ID columns.
  You must specify the columns used for features and targets in the file.

The additional information (such as number of samples to run) will depend on the type of data set you select. See the ModusToolbox™ Machine Learning user guide or the Machine Learning Configurator user guide for additional information.

You can select which of the models you want to validate against – you can select as many or as few models as you want.

Once you have selected the data set and which models to use, click the **Validate** button. Once you click that button, the models are re-generated based on your selections from the **General Settings** tab. This is done to make sure the models on disk being validated match what is in the configurator. Then the test data (either randomly generated or from another source that you provided) is formatted as .c/.h files and stored under the output directory (default *mtb_ml_gen*) in the subdirectory *mtb_ml_regression_data*. Finally, the data is sent through the inference engine and the results are displayed.

The left panel shows a sample index and the MAE (Mean Absolute Error) for the result for that sample compared to the reference. If you click on an index, the right panel will show you the probability for each class for each of the models for that sample. The table is limited to the 100 samples with the largest MAE.
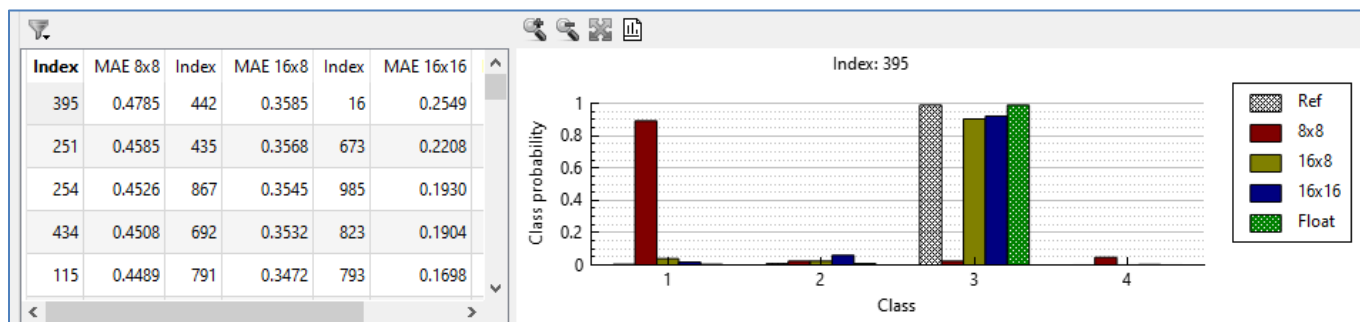
You can sort the index columns in ascending or descending order by clicking on the desired column. You can also use the filter to show results for only some of the models. The default sorting order is the largest to smallest MAE for each model. That's why each model has a different index order. However, when you click on an index, the graph is shown for all models for that index.

Note: *Since the validation is running on the desktop, you do NOT need to program the application to the PSoC™ 6 device before running the validation.*

The Output panel lists information about each model that was validated. At the very end of the log you will see a pass/fail summary. The passing condition is greater than 98% accuracy and less than 1 MB of scratch memory used by the model. A given sample counts as accurate if the class with the highest probability for the model in question is the same as the reference.

```
[INFO] Overall CDV Results:
+----------------------+------------+
|     Quantization     |   Status   |
+======================+============+
| float                |    PASS    |
+----------------------+------------+
| int16x16             |    PASS    |
+----------------------+------------+
| int16x8              |    PASS    |
+----------------------+------------+
| int8x8               |    PASS    |
+----------------------+------------+
```

For additional detail, scroll back to see information on each model. For example:

```
[INFO] Cross-domain validation passed for the int16x8:
+---------------------+-----------+-----------+-----------+
|       Metric        |  Actual   |  Required |   Status  |
+=====================+===========+===========+===========+
| Relative Accuracy   |    99     |    98     |    PASS   |
+---------------------+-----------+-----------+-----------+
| Mismatch Error      |   0.026   |   1.000   |    PASS   |
+---------------------+-----------+-----------+-----------+
| Scratch Memory      |   0.140   |    999    |    PASS   |
+---------------------+-----------+-----------+-----------+
```

4.5.3   Validate on Target tab

To validate on the target MCU, you must use the machine learning profiler code example. It comes with a default pre-trained model, but it is easy to use it with your own pre-trained model, as you will see in the exercises. The application can be found in Project Creator in the Machine Learning category:
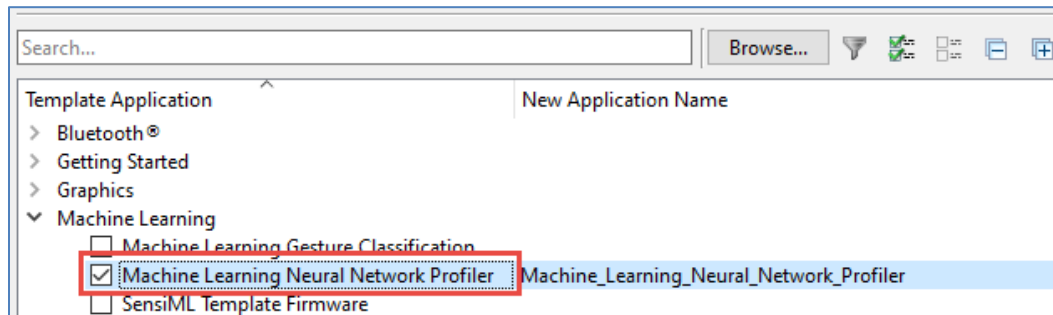
Note: The machine learning profiler code example is only available once the ModusToolbox™ Machine Learning tech pack is installed.

The application can either stream the test data to the MCU via the UART interface (default) or test data can be programmed onto the device so that local testing is done. This is set in *main.c* using the macro `REGRESSION_DATA_SOURCE`.

Note: Unlike validation on the desktop, you must program the application with one of your models to the PSoC™ 6 target device before running the validation. If you don't program the application to the device, you will not be validating the correct model.

4.5.3.1   Streaming data

When using streaming data, the ML configurator is used just like it is for desktop validation. The main difference is that you can only select one model from the device. This model will be compared against the desktop models so that you can notice any differences.

Note: The model that you select in the GUI must match the model that is programmed onto the device. That is, the variables `NN_TYPE` and `NN_INFERENCE_ENGINE` that were set in the Makefile when you built the application and programmed the device must match what is set in the GUI.

The **Available ports** must be set to the correct COM port for the board you are using for validation. Since the ML configurator is using the COM port, no other program (such as putty) can have that port open.
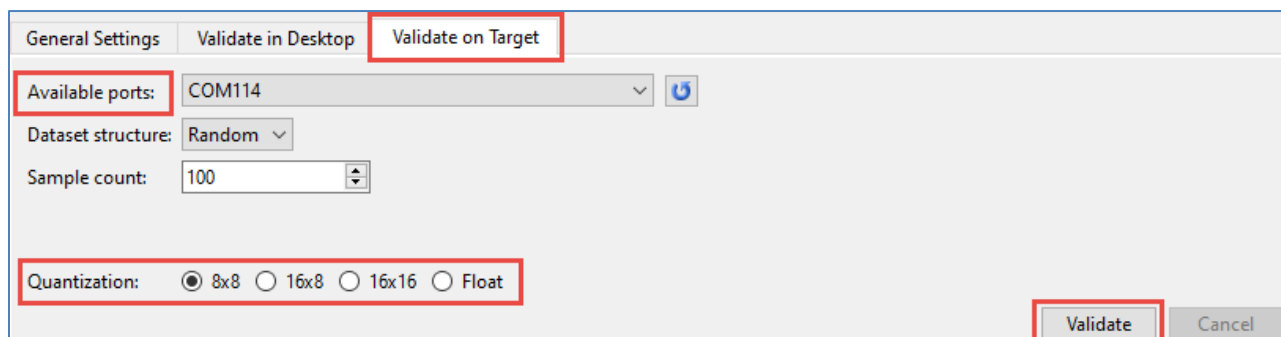
Once you click **Validate**, the test will run and you will see results that look the same as the results from desktop validation. However, in this case, the performance of the model on the MCU will be included.

4.5.3.2 Local data

When using local data, you can program the device and look at the messages in a UART terminal window. Once the device starts up, it runs through the regression data once, prints the results and then the processor goes to sleep.

When using local data, the number of samples must be limited to allow the data to fit without overflowing the flash memory. Typically, local random testing is recommended for 100 samples or less.

The data that is programmed into the device will match whatever was last generated by the tool. That is, you should set up the desired test data from the **Validate in Desktop** tab and run validation before building the application and programming the PSoC™ 6 device. Remember that running the validation puts the formatted data into the specified ML output folder under *mtb_ml_regression_data*. That folder is where the build will locate the data to include into the application.

The target MCU will only contain one version of the model, so you will only get validation data for that model. You specify which model to use in the application with the variable `NN_TYPE` in the *Makefile*.

When using local data, additional options are available for reporting. The macro `PROFILE_CONFIGURATION` in *main.c* is used to control this. The available selections are shown as comments. Note that if you are using the TFLM inference engine, the per-layer and per-frame selections will not work.

4.6 Machine Learning code examples

ModusToolbox™ contains several code examples that can be used to collect data, profile models, and deploy them. For example:

Name	Description
Machine Learning Imagimob Data Collection	This code example is used to collect data for creating a machine learning model in Imagimob Studio. It supports motion data from an IMU or digital microphone data from a PDM to PCM interface.
Machine Learning Imagimob Deploy	This code example can be used to test and deploy a Machine Learning model that was generated in Imagimob Studio.
Machine Learning Imagimob MTBML Deploy	This code example can be used to test and deploy a Machine Learning model that was generated in Imagimob Studio and then further optimized using the ModusToolbox™ Machine Learning Configurator.
Machine Learning Neural Network Profiler	This code example can be used with any Machine Learning model to obtain profile information such as the number of CPU cycles used for each layer in the model.
Machine Learning Gesture Classification	This code example demonstrates a very simple gesture classification model that is created using TensorFlow and Keras.

4.7 Deploying and integrating

Once we have a trained model and have validated it, the next step is to include it into your end application.

If you are using a model that was generated with Imagimob Studio, it is best to use the Machine Learning MTBML Deploy (mtb-example-ml-imagimob-mtbml-deploy) code example in ModusToolbox™. That code example implements all the necessary functionality to run the Imagimob model and its preprocessing.

The Keras .h5 model and the preprocessor .c and .h files are generated directly from Imagimob Studio as described in the previous chapter.

The rest of this section provides additional detail on the methods and firmware used for integration and deployment. This applies to any models that are generated from the ModusToolbox™ ML configurator.

4.7.1 RTOS

Your final application may be running an RTOS such as FreeRTOS. You might have multiple tasks running, one of which is used for running the ML operations. For example, an ML system may use an interrupt and a task to collect data from a sensor, a task to run the ML engine, and a task to act on those results. RTOS queues can be used to send data between the tasks.
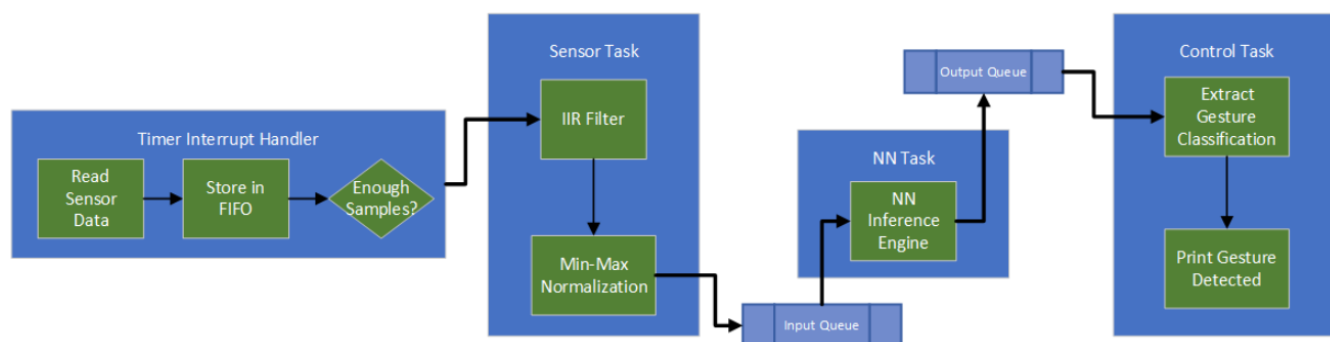
4.7.2 Sample rate

It is important to understand how long (i.e. how many CPU cycles) the inference engine will take to produce a result so that the data is collected at an appropriate rate. Namely, you probably don't want to record data from a sensor and send samples to a machine learning task faster than it can process those samples. If you were do that continuously, the input queue would eventually fill up, unless you turned off sample collection periodically to allow the machine learning task to catch up.

The typical system will slice up the time available for machine learning like this:
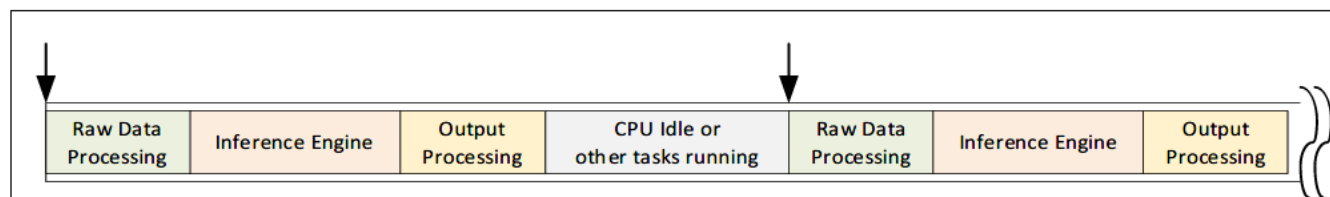
The number of cycles reported by the ML Configurator will help you determine how long the inference engine will take for each sample (time = cycles / CPU clock frequency) so that you can adjust sampling rates appropriately.

4.7.3 Handling inputs and outputs

The data fed into a neural network machine learning model is often normalized first. It is important that any normalization done to the data during training of the model is also done to the data sent into the trained model once it is deployed. One type of normalization is <u>min-max feature scaling</u> which scales the range to [0, 1] or [-1, 1]. The formula is:

$$NN\ input\ data = Nmax - (Smax - Sensor\ Data)\frac{Nmax - Nmin}{Smax - Smin}$$

The values for Smin and Smax are the sensor min and max values while Nmin and Nmax are the normalization range. If we want to normalize to [-1, 1] we get:

$$NN\ input\ data = 1 - \frac{2(Smax - Sensor\ Data)}{Smax - Smin}$$

When using fixed-point values instead of floating point, additional scaling needs to be done. This scaling is handled differently by the Infineon and TFLM inference engines.

4.7.3.1 Infineon inference engine fixed-point normalization

When using the Infineon inference engine, fixed-point format data needs to use the Q format to specify fractional bits. The value of Q indicates how many of the bits are used to represent the fractional portion of the value. For example, if a 16-bit value is normalized in the range of [-1, 1], you can use q15 format – in that case the most significant bit is the sign and the other 15 bits specify the fraction of 1.

Common Q formats for 16- and 8-bit numbers are q15 and q7, respectively. The ranges of numbers they can represent are:

Number	Q format	Floating point range	Integer range
16-bit	q15	$[-1 , 1-2^{-15}]$ $[-1, 0.9999695\ldots]$	[-32768, 32767]
8-bit	q7	$[-1, 1-2^{-7}]$ $[-1, 0.9921875]$	[-128, 127]

The equations for normalizing input sensor data in both floating-point and fixed-point Q format can be found in the ModusToolbox™ Machine Learning user guide.

As you will see, the ML middleware library contains functions that assist with the fixed-point normalization task.

4.7.3.2   TFLM inference engine fixed-point scaling

The TFLM inference engine does not require Q format when using the integer quantization. Internally, it uses asymmetric quantization, which uses zero-point and scaler variables. The zero-point and scaler values are determined during the model calibration, which requires you to provide some calibration data when generating the model files for the int8x8 model.

For TFLM, the inference engine 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 get good estimates of min/max value range, it can result in poor network performance or even in the inference engine failing to produce a result.

The results from the inference engine must be "dequantized" to get meaningful results. Again, the ML middleware library will help you with that task.

4.7.4   Libraries

There are three libraries that are part of the ModusToolbox™ Machine Learning solution. You will need one of the inference engine libraries (or both if you want the option to use either inference engine like the code examples), plus the ML middleware library.

- ml-inference: Infineon inference engine
- ml-tflite-micro: TFLM inference engine
- ml-middleware: Abstraction layer for the Infineon and TFLM inference engines and other utility functions

Each of the libraries can be added to your application in the standard fashion using the Library Manager.

4.7.5 Makefile variables

The *Makefile* for most ML code examples contain four variables used for machine learning configuration. These are:

Variable	Purpose	Allowed Values
NN_INFERENCE_ENGINE	Inference engine to be used	ifx, tflm, tflm_less
NN_TYPE	Quantization type	float, int8x8, int16x8, int16x16
NN_MODEL_NAME	Prefix for generated model files	any name
NN_MODEL_FOLDER	Directory for generated model files	any name

Each of these variables are used only in the *Makefile* to set `COMPONENTS`, set `DEFINES`, and include the correct model and regression data in the build operations. It is possible to hard-code each of the appropriate `COMPONENTS`, `DEFINES`, `CY_IGNORE`, `SOURCES` and `INCLUDES` variables, but using the provided framework makes the code more re-usable.

The variable `NN_MODEL_NAME` also creates a define for `MODEL_NAME` which is used in the firmware to include the model and populate the model's data structure with the correct information.

Depending on the requirements of the application, these variables may be used differently. For example, the profiling code example needs both the model and the regression data in the firmware so it includes those as source files:

```
# Select only the regression and model files that belong to the desired
# settings.
MODEL_PREFIX=$(subst $\",,$(NN_MODEL_NAME))
CY_IGNORE+=$(NN_MODEL_FOLDER)
# Add the model file based on the inference and data types
SOURCES+=$(wildcard $(NN_MODEL_FOLDER)/mtb_ml_models/$(MODEL_PREFIX)_$(NN_INFERENCE_ENGINE)_model_$(NN_TYPE).c*)
# Add the regression files
ifeq (ifx, $(NN_INFERENCE_ENGINE))
SOURCES+=$(wildcard $(NN_MODEL_FOLDER)/mtb_ml_regression_data/$(MODEL_PREFIX)_$(NN_INFERENCE_ENGINE)_*_data_$(NN_TYPE).c)
else
SOURCES+=$(wildcard $(NN_MODEL_FOLDER)/mtb_ml_regression_data/$(MODEL_PREFIX)_tflm_*_data_$(NN_TYPE).c)
endif
```

On the other hand, the gesture code example does not need the regression data so it only includes the model in order to save space:

```
# Select only the regression and model files that belong to the desired
# settings.
MODEL_PREFIX=$(subst $\",,$(NN_MODEL_NAME))
CY_IGNORE+=$(NN_MODEL_FOLDER)
# Add the model file based on the inference and data types
SOURCES+=$(wildcard $(NN_MODEL_FOLDER)/mtb_ml_models/$(MODEL_PREFIX)_$(NN_INFERENCE_ENGINE)_model_$(NN_TYPE).c*)
```

4.7.6 Firmware

The underlying steps to get the ML system up and running depends on whether you are using the Infineon or TFLM inference engine. However, the ml-middleware library provides an abstraction layer that makes the process very similar between the two even though what happens "under the hood" is considerably different. This manual focuses on using the middleware library. If you want to see the lower-level inference library functions being used, refer to the ModusToolbox™ Machine Learning user guide or the documentation of the inference library itself.

There are two main functions: one to initialize the model (including memory allocation) and one to run the inference engine. Additional helper functions and macros provide ways to get information about the model, help with quantization, and allow model profiling. The basic flow used is:

4.7.6.1 Initialize and allocate memory

The main initialization function is `mtb_ml_model_init`. You pass it a pointer to a structure with the model binary data, an optional pointer to a buffer for persistent and scratch memory, and a pointer to a model object. If the memory buffer is not provided, the API allocates memory automatically. The middleware library has a macro for getting the model's binary data based on the model name used in the configurator. If you use the *Makefile* variables as described above, `MODEL_NAME` will automatically be set to the name of your model based on the value of `NN_MODEL_NAME` set in the *Makefile*.

For example:

```
mtb_ml_model_t *model_obj;
mtb_ml_model_bin_t model_bin = {MTB_ML_MODEL_BIN_DATA(MODEL_NAME)};
result = mtb_ml_model_init(&model_bin, NULL, &model_obj);
```

If you are using the Infineon inference engine in integer mode (i.e. not float), you also need to set the Q factor:

```
#if (COMPONENT_ML_INT16x16 || COMPONENT_ML_INT16x8)
    #define QFORMAT_VALUE    15
#endif
#if (COMPONENT_ML_INT8x8)
    #define QFORMAT_VALUE    7
#endif
mtb_ml_model_set_input_q_fraction_bits(model_obj, QFORMAT_VALUE);
```

Finally, you call one last initialization function to get the location and size of the model's output buffer:

```
MTB_ML_DATA_T *result_buffer;
int model_output_size;
mtb_ml_model_get_output(model_obj, &result_buffer, &model_output_size);
```

4.7.6.2 Run the inference engine

The function to run the inference engine is `mtb_ml_model_run`. You pass it a pointer to the model object, and a pointer to the input data that will be processed by the model. For example:

```
MTB_ML_DATA_T  *input_buffer;
mtb_ml_model_run(model_obj, input_buffer);
```

4.7.6.3 Data quantization

There are three functions that can be used for conversion of the data to be used by the models. They are:

`mtb_ml_utils_convert_flt_to_int`

> Converts floating-point to fixed-point based on the provided Q value. This is for the IFX engine. If called for TFLM, this does nothing as the conversion is taken care of in the inference engine itself.

`mtb_ml_utils_convert_int_to_flt`

> Converts fixed-point to floating-point based on the provided Q value. This is for the IFX engine. It should not be called when using TFLM, as it will produce data that is not useful.

`mtb_ml_utils_model_dequantize`

> Does the dequantize for the TFLM engine. For the IFX engine it calls `mtb_ml_utils_convert_int_to_flt` so it can be used for both engines.

Therefore, the same set of functions can be used for both inference engines for quantizing, running the inference engine, and dequantizing. For example, you could do something like this:

```
mtb_ml_utils_convert_flt_to_int(input_buffer_flt, input_buffer,
                                size, q_value);
mtb_ml_model_run(model_obj, input_buffer);
float *result_buffer_flt = (float *) malloc(model_output_size*sizeof(float));
mtb_ml_utils_model_dequantize(model_obj, result_buffer_flt);
```

Quantization and de-quantization are only necessary if the data and the model aren't matched. That is, if the data is floating-point and you have a floating-point model, conversion is not required.

4.8　　　　　Exercises

Exercise 1:　Install ModusToolbox™ Machine Learning Technology Pack

Note:　　　　　*If you already did the installation in Chapter 1 you can skip this exercise.*

☐　　1.　You should already have ModusToolbox™ installed from either the Level 1 Getting Started or Level 2 PSoC™ MCUs class. If not, install it first.

　　　　　You must have ModusToolbox™ version 3.0 or later.

☐　　2.　If you are using ModusToolbox™ version 3.2 or later, run ModusToolbox™ Setup and install the Machine Learning pack.

　　Note:　　　*If you are using ModusToolbox™ version 3.0 or 3.1, go to* https://softwaretools.infineon.com/tools/com.ifx.tb.tool.modustoolboxpackmachinelearning *to download and install the ModusToolbox™ Machine Learning pack.*

　　Note:　　　*Install the pack in the same location that you installed ModusToolbox™. If you just installed ModusToolbox™ for the current user, repeat the same for the pack.*

Exercise 2:　Install QEMU

Note:　　　　　*If you already did the installation in Chapter 1 you can skip this exercise.*

☐　　1.　The ModusToolbox™ Machine Learning Configurator requires a 3rd party tool called QEMU. Download and install it now.

　　Note:　　　*Instructions for downloading, installing, and configuring QEMU can be found in the Software requirements section of the ModusToolbox™ Machine Learning user guide. The default location of the user guide is ~/ModusToolbox/packs/ModusToolbox-Machine-Learning-Pack/help/ml-user-guide.pdf.*

　　Note:　　　*You must install a specific version of QEMU. Be sure to follow the instructions in the section for your operating system rather than following the link to the top level QEMU site as that will point you to a different version.*

　　Note:　　　*When creating the* `IFX_MTBML_QEMU_PATH` *variable, be sure there are no trailing spaces at the end of the variable name or path. You must use backslashes for the path (e.g. C:\Program Files\qemu).*

Exercise 3: Optimize a pre-trained Imagimob model

In this exercise, we will take the Human Activity Recognition model you created in the Imagimob chapter and further optimize it using the ModusToolbox™ ML configurator.

☐ 1. Create the Machine Learning Imagimob MTBML Deploy (mtb-example-ml-imagimob-mtbml-deploy) example in ModusToolbox™. Name the application **ch04_ex03_model_optimization**.

☐ 2. Copy the Kara's h5 model file and the .c/.h preprocessor files generated from the Human Activity Recognition exercise in the Imagimob chapter to the newly created application.

Note: *Refer to the previous chapter for details on how to export the model files for use in the ModusToolbox™ ML configurator.*

Note: *The files should replace the existing files in the Edge/IMU directory of the ModusToolbox™ application.*

☐ 3. Open the Machine Learning Configurator and make sure it is pointing to the model you copied above.

☐ 4. Change the Model optimization settings to **Enable sparsity**.

☐ 5. Generate Source for the model to work with Tensor Flow Light Micro (TFLM).

☐ 6. Optional: Validate the model on the desktop or on the target to confirm that it works as expected.

☐ 7. Program the design onto the board and try it out to ensure it works as expected.

☐ 8. Compare the performance of the optimized model vs what was obtained in the exercise in the prior chapter. The size of the model is easy to compare just by looking at the binary size. The performance can be measured by starting a timer when the inference operations begin and stopping it when they finish.

Note: *The model created by the ModusToolbox™ ML configurator makes use of specialized kernels for ARM devices.*

Exercise 4: Use the profiler code example

In this exercise, you will utilize the profiler code example with the default model to learn how to use the different features of the profiler.
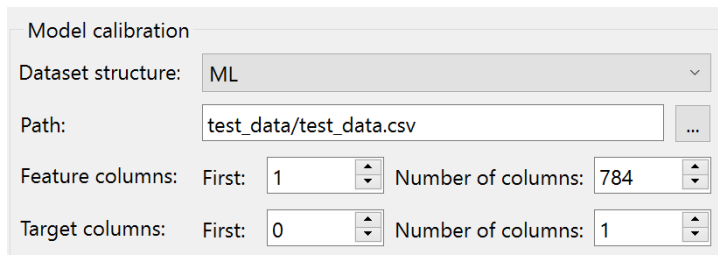
Note: *The default model in the profiler code example is a handwritten digit recognition model that uses the [MNIST](#) (Modified National Institute of Standards and Technology) database for training and testing of the model. The model has 10 classes – one for each of the digits ranging from 0 to 9.*

Setup the model and generate source

1. Create an application using the Machine Learning Neural Network Profiler code example. Name the application **ch04_ex04_profiler**.

2. Open the ML configurator from the application.

   Note: *Either use the Quick Panel link in Eclipse or use make ml-configurator from the command line.*

3. On the **General Settings** tab, set the inference engine is set to **ifx** and check the **Advanced scratch memory optimization** box.

4. Save changes and then click the **Generate Source** button to create the model and regression data files.

5. Review messages in the output window and record how many cycles and how much memory each model requires.

6. Change the inference engine to **tflm** and check the box for **Enable model quantization**.

7. Under Model calibration, verify that the settings are as shown below.

8. Save changes and then click the **Generate Source** button to create the model and regression data files.

9. Review messages in the output window and record how many cycles and how much memory each model requires. Compare these with the Infineon inference engine results.

10. Change back to the **ifx** inference engine. The **Advanced scratch memory optimization** box should remain checked.

Validate in Desktop

☐ 11. Switch to the **Validate in Desktop** tab. Set the **Dataset structure** to ML use the same settings that were used in the Model calibration section of the General settings tab (see step 7 above). Verify that all **Quantization** selections are checked. Save changes and then click **Validate**.

☐ 12. Examine the MAE (Mean Absolute Error) values for each model in the table and view the probabilities of each class being selected by each model for a few of the index values. Look in the output window to see how each model performed for accuracy and memory use.

Note: The chart only shows the 100 indices with the largest MAE for each model.

☐ 13. If desired, you can rerun the validation with the dataset structure set to ML. The settings for ML are the same as those used above during model calibration.

Validate on Target (Stream)

☐ 14. Switch the **Validate on Target** tab.

☐ 15. Select the appropriate COM port for your kit.

☐ 16. Select either Random or ML for the **Dataset structure** and select the options as needed.

☐ 17. Select the **Quantization** value that matches the value that will be programmed to the device.

Note: The value you choose must be the same quantization for the model that is programmed to the device. Otherwise, validation will fail. You can try it to see what happens if you want.

Note: Remember that the model quantization selection is in the Makefile.

☐ 18. Go to the **General Settings** tab and click **Generate Source**. This is necessary so that the regression dataset that you program to the device will match what you selected in the previous steps.

☐ 19. Open the application's Makefile and set the `NN_INFERENCE_ENGINE` to `ifx` since that's the model we want to validate.

☐ 20. Build the application and program it to the kit.

☐ 21. Go to the **Validate on Target** tab. Click **Validate** and review the results once the validation has completed.

Validate on Target (Local)

22. If you are using Random data, set the **Sample count** to 100 to make sure it will not overflow the flash memory on the device.

23. Go to the **General Settings** tab and click **Generate Source**. This is necessary so that the regression dataset that you program to the device will match what you selected in the previous step.

24. Open *main.c* and change the `REGRESSION_DATA_SOURCE` to `USE_LOCAL_DATA`.

25. Open a UART terminal window (115200, N, 8, 1) to the appropriate COM port.

26. Build the application and program it to the kit. Once programming completes, validation will run and the results will be visible in the terminal window.

27. Try changing the `PROFILE_CONFIGURATION` in *main.c* to different options. Reprogram the device and view the output printed in the UART terminal window.

Note: *The different options allow you to look at statistics per layer in the model, per frame (i.e. per sample), or both.*

Note: *The TFLM inference engine only supports the MTB_ML_PROFILE_ENABLE_MODEL and MTB_ML_LOG_ENABLE_MODEL_LOG selections.*

Exercise 5: Run the gesture classification code example

In this exercise, you will create the gesture classification code example and will run it with different options.

1. Create an application using the Machine Learning Gesture Classification code example. Name the application **ch04_ex05_gesture**.

2. If you are using the CY8CKIT-028-SENSE, you must change the value of `SHIELD_DATA_COLLECTION` in the *Makefile*.

Note: *If the silk screen on the back of the shield says "9-axis IMU" and "600-60608-01 REV03" it is v1. If the silk screen says "6-axis IMU" and "600-60608-01 REV04" it is v2.*

3. Open the ML configurator to look at the model settings. Click **Generate Source** to make sure the models on disk match the settings.

4. If you are using a Mac, you may have to change the following line in the *Makefile* to work with csh instead of bash. The necessary addition is a pair of single quotes.

Old: `PREBUILD+=sed –i 's/UINT8_C(0xD1)/UINT8_C(0xD8)/' bmi160/bmi160_defs.h;`
New: `PREBUILD+=sed –i   ' '   's/UINT8_C(0xD1)/UINT8_C(0xD8)/' bmi160/bmi160_defs.h;`

5. You can validate on the desktop, but validating on the target will not work because the firmware is not set up for it.

6. Read the *README.md* file to understand how the code example works.

7. Program the application to the kit.

8. Open a UART terminal to the kit and verify that the gesture detection works correctly.

Questions

Here are some questions to test your understanding of how the application works.

1. Which model is running on the device (inference engine and quantization level)?

2. Which task runs the inference engine? What function call is used to run the inference engine?

3. How is the sensor data collected and how is it fed to the inference engine?

4. How often is a sample collected from the sensor? How many samples are collected before sending them to the inference engine to process? How often are a set of samples sent to the inference engine?

5. What pre-processing is done to the sensor data before sending it to the inference engine?

6. How long does the inference engine take to come up with a result? How does it compare to the amount of time between sets of samples being ready for processing?

7. If the inference engine comes up with: Circle 40%, Square 30%, Side-to-Side 30%, what gesture is reported and why? For a hint, look in *control.c*.

Exercise 6: Use the profiler code example to evaluate the gesture model

In this exercise, you will replace the default model in the profiler code example with the gesture model so that you can use it to evaluate the model's performance on the MCU.

1. Create an application using the Machine Learning Neural Network Profiler code example. Name the application **ch04_ex06_gesture_profiler**.

2. Copy the Keras (.h5) model from the *pretrained_models* directory in the gesture application from the previous exercise into the *pretrained_models* directory of the new application.

3. Copy the file *train/calibration.npz* from the gesture application into the *test_data* directory of the new application.

   Note: *This is necessary for calibration if we are going to use the TFLM inference engine.*

4. Open the ML configurator in the new application and change the **Pretrained model** to use the gesture model that you copied over.

5. Verify that the **Inference engine** matches what was used in the gesture code example.

6. Under **Model calibration**, select NPZ for the **Dataset structure** and set the path to the *calibration.npz* file that you copied over.

7. Go to the **Validate in Desktop** tab and set the **Dataset structure** and **Path** to the same values as the previous step.

Note: This is necessary because the regression data is generated when the model is generated. If the dataset were to be left as the test data for the default model contained in the profiler code example, it would not match the new model's requirements and the generation would fail.

Note: You could also use random data to test the model, but targeted data gives a better idea of how the model will perform in real conditions.

8. Save changes, go back to the **General Settings** tab, and click **Generate Source**.

9. Go to the **Validate in Desktop** tab and click **Validate.** View the results of the desktop validation once it has completed.

10. Open the application's *Makefile* and set the inference engine and quantization level to match what was used in the gesture code example.

11. Program the application to the kit.

12. On the **Validate on Target** tab, select the correct COM port. Make sure the **Quantization** selection matches what is set in the *Makefile*.

13. Click **Validate** and view the results of the validation on the target MCU.

Exercise 7: Re-train the gesture model with an up-down gesture

In this exercise, you will collect data and re-train the gesture model to replace the square gesture with an up-down gesture.

1. Create an application using the Machine Learning Gesture Classification code example. Name the application **ch04_ex07_gesture_updown**.

2. If you are using the CY8CKIT-028-SENSE shield, you must change the value of `SHIELD_DATA_COLLECTION` in the *Makefile* to `CY_028_SENSE_SHIELD_v1` (shield rev **) or `CY_028_SENSE_SHIELD_v2` (shield rev *A or later).

3. Open the *README.md* file from the application and go to the **Model generation** section.

4. Follow the instructions under "Collect data," "Generate a model," and "Model deployment" with the modifications listed below:

   For "Collect data", step 2, only remove the "Square" sub-directory and its contents so that we can replace it with our new Up-Down gesture. Do NOT remove the other three directories so that we can re-use the training data for those gestures.

   When collecting data:
   - Make sure you don't have a terminal window open since the script needs access to the COM port.
   - Use the name "Up-Down" for the new gesture.
   - Move the kit up and down for at least two minutes while capturing data.
   - Move the kit at varying speeds and distances while capturing data.
   - If you prefer, you can run multiple captures. Just use a different "person name" for each capture so that the files are unique (e.g. greg1, greg2).
   - You can even capture data from other people (or just get a copy of their file and place it int the *train/gesture_data/Up-Down* directory) to get more variation in the training data set.

When generating the model:

- You may see error messages such as " Could not load dynamic library 'cudart64_110.dll." This will occur if your PC does not have a GPU. You can ignore this error. It only means that training will take longer.
- When the data is read in, you will see a message listing how many samples have been found for each gesture (class) and messages indicating how the data was split between training, validation, and test data. This is useful to determine if you have collected enough data.
- Once the model is being trained, you will see messages for each Epoch showing the loss (and accuracy) of the model for the training data and the validation data. Observe how the loss goes down on every epoch for the training data but the validation loss eventually starts to go back up. This indicates when overfitting starts to occur.

5. Review the *collect.sh* script (and the underlying *collect.py* script) to see how the samples are collected.

6. Review the *generate_model.sh* script (and the underlying Python scripts) to see how the data is split and how the training is done.

4.9   Appendix

Answers to the questions asked in the exercises above are provided here.

4.9.1   Exercise 5 Answers

1. Which model is running on the device (inference engine and quantization level)?

   - The TFLM float model is running on the device. This can be seen in the Makefile:
     ```
     NN_INFERENCE_ENGINE=tflm
     NN_TYPE=float
     ```

2. Which task runs the inference engine? What function call is used to run the inference engine?

   - The task `gesture_task` runs the inference engine.
   - The function call to run the inference engine is:
     ```
     mtb_ml_model_run(magic_wand_obj, input_reference);
     ```

3. How is the sensor data collected and how is it fed to the inference engine?

   - The sensor ISR reads the sensor and collects the data into a FIFO called `sensor_fifo`.
   - The gesture task function calls `sensor_get_data` which waits for the ISR to set an event (which it does every 128 interrupts or every 1 second) and then reads the data from the FIFO and passes it pack the gesture task into an int16 array called `temp_buffer`.

4. How often is a sample collected from the sensor? How many samples are collected before sending them to the inference engine to process. How often is a set of samples sent to the inference engine?

   - A new sample is collected using a timer that creates an interrupt at a rate of 128Hz (one sample every 7.8125 ms).
   - 128 samples are collected before sending them to the inference engine.
   - A set of samples is sent to the inference engine every 1 second (128 samples * 7.8125 ms/sample).

5. What pre-processing is done to the sensor data before sending it to the inference engine?

   - An IIR filter and Min-Max normalization (to get all data between -1 and 1) are used to pre-process the sensor data. This can be seen in *gesture.c*.

6. How long does the inference engine take to come up with a result? How does it compare to the time between sets of samples being ready for processing?

   - The example uses the float model by default. The inference engine for that model takes 11,712,331 cycles to come up with a result. The CPU clock is running at 100 MHz. That means it takes 117ms
   - (11,712,331 cycles / 100,000,000 cycles/second).
   - A new set of samples is sent every 1 seconds and the inference engine takes 117 ms to produce a result, so there is plenty of margin in the system.

7. If the inference engine comes up with: Circle 40%, Square 30%, Side-to-Side 30%, what gesture is reported. Why? (Hint: look in *control.c*).

   - The gesture will be "negative" because the confidence must be greater than 60% to identify a gesture. This reduces the possibility of a false gesture being detected. It is better to have a false negative than a false positive in this application.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.