1 Overview

This application note presents the process of building and deploying deep learning models for Smart Sensing Appliances. It also highlights how to validate and evaluate the performance of a model by running it through different inference engines on an Embedded Sensing Device. By Embedded Sensing Device we mean an MCU-based device capable to measure through sensors various environmental parameters, such as, acceleration, magnitude, orientation, temperature, pressure, sound, and electric current.

This document can be used as a guideline for building applications that rely on deep learning applied on data collected from sensors to perform classification and detection of various events on MCUs. There is a broad range of use cases which can be approached leveraging this document, such as, preventive maintenance, state detection, state monitoring, activity recognition, gesture recognition, and acoustic event detection.

This document describes how to solve a real use case by building a Convolutional Neural Network (CNN) on a host machine and deploying and benchmarking it on an MCU-based device.

The development and deployment are mainly leveraging the NXP’s SDK and the eIQ technology.

For more details and resources, see the ML-Based System State Monitor Application Software Pack page.

2 Introduction of eIQ and Runtime Inference Engines

The NXP eIQ Machine Learning (ML) software development environment enables the use of ML algorithms on NXP EdgeVerse microcontrollers and microprocessors, including i.MX RT crossover MCUs and i.MX family application processors. eIQ ML software includes an ML workflow tool called eIQ Toolkit, along with runtime inference engines, neural network compilers, and optimized libraries. This software leverages open-source and proprietary technologies. It is fully integrated into our MCUXpresso SDK and Yocto development environments, making it easy to develop complete system-level applications.

Figure 1. eIQ software ecosystem
The runtime inference engines included in the eIQ component of the MCUXpresso Software Development Kit (SDK) for microcontrollers which will be evaluated in this document are enumerated on the left side of Figure 1: DeepViewRT, TensorFlow Lite / TensorFlow Lite Micro, and Glow.

2.1 DeepViewRT

eIQ Inference with DeepViewRT is a platform-optimized, runtime inference engine that scales across a wide range of NXP devices and neural network-compute engines. Provided free of charge, this inference engine enables a compact code size for resource-constrained devices, including the i.MX RT crossover MCUs (Arm Cortex-M cores) and i.MX applications processors (Cortex-A and Cortex-M cores, dedicated Neural Processing Units (NPU) and GPUs).

2.2 TensorFlow Lite for Microcontrollers

eIQ inference with TensorFlow Lite for Microcontrollers (TF Micro) is optimized for running machine-learning models on resource-constrained devices, including NXP’s i.MX RT crossover MCUs. This TF Micro implementation is faster and smaller than the traditional open source TensorFlow Lite platform for machine learning, enabling inference at the edge with lower latency and smaller binary size.

2.3 Glow

The Glow machine learning compiler enables ahead-of-time compilation. The compiler converts the neural networks into object files, then the user converts them into a binary image to increase performance and reduce memory footprint compared to a traditional runtime inference engine.

3 Application development workflow

![Application development workflow](image)

The application development follows the workflow shown in Figure 2:

1. **Sensor Data**

   This phase defines and creates the data collection for model training and validation. The application running on the embedded device collects and transfers sensor data to the host machine where the dataset is stored.

2. **Model Development**

   The eIQ Toolkit enables machine-learning development for vision-based models with development workflow tools, along with command-line host tool options as part of the eIQ ML software development environment. Model development and training can also be done using one of deep learning frameworks available. The model is developed on the host machine by going through the phases shown in Figure 3:

![Model development workflow](image)
3. **Model Deployment.**

The model created on the previous phase will be deployed and evaluated on the embedded device through the runtime inference engines defined on *Introduction of eIQ and Runtime Inference Engines.*

4 Going through a real use case

This chapter follows the workflow diagrams from *Application development workflow* and describes how to build and deploy on an embedded device a model capable to monitor the input sensor data and detect the current state of the device. The main objectives are to:

- Illustrate how to collect and record the device states dataset on a host machine
- Aggregate and analyze sensor data on the host machine
- Create, train, and validate a classification model on the host machine
- Convert the model into a format suited for an embedded device
- Evaluate the model on the embedded device

4.1 Prerequisites

Alongside with this document, the ML-Based System State Monitor delivers the applications for the host machine and embedded devices:

- **ML_app** – the host machine application used to generate the ML-model, developed in Jupyter Notebook with Python kernel
- **MCU_app** – the embedded device applications, developed in MCUXpresso IDE

Software requirements on the host machine:

- ML-Based System State Monitor source code
- Jupyter Notebook (Python, TensorFlow, Keras, etc.)
- MCUXpresso IDE
- eIQ Toolkit

Hardware requirements:

- Windows host machine
- Target embedded device
- SD card – the SD card usage is optional but can facilitate the recording of sensor data
- Sensor board – FRMD-STBC-AGM01
  
  The usage of the external sensor shield is optional and only required if the needed sensors are not already populated on the evaluation kit.

  - Fan setup – this configuration depends on the system setup and available parts

  For the presented case study, a **5V DC Fan** was attached to the evaluation kit and powered through an Arduino Proto-Shield. **Figure 4** presents the evaluated embedded setup built by leveraging the MIMXRT117-EVK and the parts described above stacked over the Arduino Interface. For more details on Software and Hardware installation, see the Lab Guide.
4.2 Case Study

The case study that is exemplified in this Application Note is a **Fan State Classifier using Accelerometer Sensing**, but it can be extended to any other use case.

The purpose of this application is to determine the current state of a fan by running a DL model on the data read from the accelerometer. The predefined classes are the following:

- **FAN-OFF** – the fan is turned off
- **FAN-ON** – the fan is turned on and running in normal conditions
- **FAN-Clogged** – the fan is turned on and the airflow is obstructed
- **FAN-Friction** – the fan is turned on and excessive friction or a blade fault is detected.

4.2.1 Sensor Data Collection

The input signals, which are monitored, are the 3-axis accelerations read from a FOXS8700 sensor with the specifications described in Table 1.
Table 1. Sensor data specifications

<table>
<thead>
<tr>
<th>Sensor</th>
<th>FXOS8700 (6-axis sensor)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3-axis 14-bit accelerometer</td>
</tr>
<tr>
<td></td>
<td>3-axis 16-bit magnetometer</td>
</tr>
<tr>
<td>Used Channels (Nch)</td>
<td>3 channels (3-axis accelerometer)</td>
</tr>
<tr>
<td>Sampling Frequency (Fs)</td>
<td>200 Hz (5 ms)</td>
</tr>
<tr>
<td>Input-Output data Type/Size</td>
<td>float32/4 bytes (per each individual channel)</td>
</tr>
<tr>
<td>Sample size (s)</td>
<td>Nch (each recorded sample includes all monitored channels)</td>
</tr>
<tr>
<td>Window Size (w)</td>
<td>128 samples (640 ms)</td>
</tr>
<tr>
<td>Overlap Ratio</td>
<td>50 % - 64 samples (320 ms)</td>
</tr>
<tr>
<td>Input Tensor Shape</td>
<td>(w, 1, Nch) = (128, 1, 3)</td>
</tr>
<tr>
<td>Input Tensor Size</td>
<td>1.5 KB (w * Nch * 4 bytes)</td>
</tr>
<tr>
<td>Data Stream Format</td>
<td>Interleaved</td>
</tr>
<tr>
<td>Training Data Collection</td>
<td></td>
</tr>
<tr>
<td>Input Tensors Count (Nt)</td>
<td>5.76 Ms                                      1.44 Ms/class</td>
</tr>
<tr>
<td>Duration</td>
<td>90 Kw                                        22 Kw/class</td>
</tr>
<tr>
<td></td>
<td>8 h                                          2 h/class</td>
</tr>
<tr>
<td>Validation Data Collection</td>
<td></td>
</tr>
<tr>
<td>Input Tensors Count (Nt)</td>
<td>1.68 Ms                                      0.42 Ms/class</td>
</tr>
<tr>
<td>Duration</td>
<td>26 Kw                                        6 Kw/class</td>
</tr>
<tr>
<td></td>
<td>2 h 20 min                                   35 min/class</td>
</tr>
</tbody>
</table>

The interleaved data stream format and the moving window method shown in Figure 6 and Figure 7 are used to collect and reshape the dataset in order to feed the neural network. For example, the first three tensors that are processed by the neural network have the following format:

![Interleaved samples](image)

Figure 6. Interleaved data stream format
Unlike image or audio, time series sensor data are often unique for the product setup, depending on sensor type, sensor placement, location, surface, and so on. Thus, we cannot use datasets already available, and we must create (collect and annotate) a dataset relevant for this particular setup.

The dataset is collected by enabling the embedded application to log the sensor data (on the SD card or over the serial debugging interface) and by providing the required configuration on the terminal:

```c
/* Configure the action to be performed */
#define SENSOR_COLLECT_ACTION        SENSOR_COLLECT_LOG_EXTERNALLY

Provide the required configuration on the terminal to start the recording:
Class to record (provide only the numeric index): >>> 0
Duration in minutes: >>> 5
SD card filename: >>> Vd1-clog.csv
```

The embedded device will communicate (log information, sensor data, input configuration) with the host machine through the Debug Console Interface (UART over USB, 115200 bps, 8 data bits, 1 stop bit, no parity, no flow control) so a serial terminal application (for example, PuTTY) must be used on the host machine to facilitate the communication (inspect the output, save sensor data, provide configuration).

The goal for any machine-learning model is to learn from examples (training data) in such a way that the model is capable to generalize the learning to new unseen instances (validation/testing data). Therefore, we must collect data covering as much variation as possible following a training and validation split for the dataset.

The setup must be configured for each class like in Figure 8 to capture all the states defined and the data need be collected and stored for every particular one. The main rule for collecting the data is to cover as much as possible every corner case and not to use the same data for training and validation. Any pattern for duration and how to switch between classes can be followed, depending on the use case and the analysis behind that. For the presented case study every recording session collected a balanced dataset (data collected equally improve the model performance by providing equal distribution and priority for each class) trying to cover as much variation as possible.
One session of recording is presented in Figure 9 where the recording was started by collecting a validation dataset for a short period (5 mins), followed by a dataset for training and validation collected for a longer period (1 h + 5 mins), and ending with a validation dataset collected again for a short period (5 mins). For the demonstrated case study, it was easy to collect data for the classes **OFF** and **ON** while for the **Clogged** class a thin piece of cardboard was used to block the airflow and for the **Friction** class a corner of the cardboard was pressed on the blades to produce friction.

**Figure 8. The setup configured to record each class**

**Figure 9. The timeline for one session of data recording**

**Figure 10** shows a snapshot from a validation dataset collected in one recording session. The files capture the state of the fan, the sample time, the X, Y, and Z axis of the accelerometer, the X, Y, and Z axis of the magnetometer, and the temperature (T). Our model only uses the time and accelerometer data, but other applications may find those other fields useful for their deep learning models.
4.2.2 Model development

The model development flow described in chapter Application development workflow and Figure 3 is applied to the case study by running on the host machine with Jupyter Notebook provided.

4.2.2.1 Importing the dataset

The files storing the sensor data in the /in_sensor_data/ directory are imported by executing the first sections of the notebook. A summary of the imported dataset is returned by these sections.
4.2.2.2 Data analysis, processing, and shaping

Data analysis and preprocessing are optional, though these steps can increase the clarity of the analyzed data (Figure 12 and Figure 13). Real-time data analysis can also be achieved by plotting real time collected data using the provided real_time_plot function.

In terms of preprocessing, an increase in model performance has been observed after applying a fixed normalization and reducing the dataset to normalized range [-1, 1] as in Figure 14. In addition, the frequency domain highlighted in Figure 13 produces quite differentiating spectrums which could be extracted and fed into a neural network. Though, for this specific use case only time domain has been considered as it reduces the work of developers and embedded processing overhead (that is, the raw data are directly fed into the network without too much processing) and increases the model adaptability to various patterns.
The step of data reshaping is mandatory in order to be able to feed the model to process the inputs. The dataset is split and reshaped as a 2D-tensor for every channel, resulting in a reshaped final dataset with the following format:

\[
\text{dataset} = (N_t, \text{height, width, depth}) = (N_t, \text{window, 1, Nch}) = (N_t, 128, 1, 3)
\]

\(N_t\) – the total number of input tensors
4.2.2.3 Model definition, training, and evaluation

The architecture of the neural network relies on two stages: the first one is used for feature extraction and the last computes the result. Between them, an auxiliary phase for filtering and adaptation is used.

The network is implemented leveraging the Keras framework which is the Python deep learning API for TensorFlow. The network structure and implementation are shown in Figure 16 and Figure 17 and rely on the following components and layers:

- **Conv** – consists of filters and weights applied to the input that results in output activation.
- **MaxPool** – pooling layers are used to reduce the dimension of the input and the MaxPool filter selects the sharpest features and decreases the model sensitivity to noise.
- **Flatten** – converts the input into a single long feature vector for the final classification phase.
- **Dense** – constructs the fully connected stage which computes the classification results.
- **Dropout and Regularization** – reduce the overfitting effect and increase the model capability to perform well on new unseen data by randomly dropping out units/diluting weights and by adding a penalty in the error function.
- **Adam optimizer** – is an improved version of the stochastic gradient descent used to train the network and update the weights iterative based on training data.
• Categorical cross-entropy – is the loss function used in training to compute the quantity that the network seeks to minimize.

• Accuracy – is the metric used to evaluate the model performance.

Once the network architecture is defined, the next step is to start the training process. Through training, the network learns from the provided training dataset and automatically configures the model parameters. The model parameters specify how to transform the input into the desired output. In addition, hyperparameters can be tuned at this stage. Hyperparameters refer to the network structure, architecture, used optimizer, learning rate, and so on, and directly influence the performance. There is no specific way to calculate the hyperparameters and they must be manually tuned to find the optimal configuration for a specific use case. This process is typically based on experimentation and reiterations when evaluating performance with a validation dataset.
The model can be evaluated on the validation dataset either by visually analyzing the accuracy and loss or by using a confusion matrix like shown in Figure 19. The training session can be evaluated by plotting and analyzing the history of the progress over iterations as in Figure 20.
4.2.2.4 Model validation on host machine

The obtained accuracy for the built model goes up to 99% as shown in Figure 19 (99% for training data and 98% for validation data). In addition to evaluating the model by running it on data previously recorded, there is another method to validate the model running on a PC using new unseen data by reading and processing sensor data provided in real time by the embedded device. To run this validation phase, the board must be connected to a host machine and configured to to externally log sensor data via the console.

4.2.2.5 Model porting for embedded device

The trained model can be converted to run on the embedded board by using the provided `save_model` function. This function saves three flavors of the model: full Keras model (`model.h5`), converted TFLite model (`model.tflite`), and the TFLite-quantized...
model (model_quant.tflite). For quantization a hybrid post-training approach is to maintain compatibility with applications by keeping the full size of inputs/outputs tensors (float32) while quantizing and decreasing to int8 only the parameters between layers.

TensorFlow Lite Micro inference engine is used to run on the embedded board the model exported in tflite format while for DeepViewRT and Glow some additional steps must be executed to properly convert the model in the specific format. These steps rely on the eIQ Toolkit (eIQ Portal, eIQ Model Tool, eIQ Glow). For more information, see chapter Introduction of eIQ and Runtime Inference Engines.

The Model Tool from eIQ Portal software suite must be used for conversion in the RTM format (DeepViewRT) as shown in Figure 22.

For glow format, the model-compiler.exe with the following parameters must be executed in the command line to generate the bundle as shown in Figure 23 (the glow_model_fan_clsf.tflite is the model_fan_clsf.tflite model copied and renamed for file distinction). Depending on the embedded device target, the -target and -mcpu parameters might need to be changed.

```bash
model-compiler.exe -model=models\model_fan_clsf\glow_model_fan_clsf.tflite -emit-bundle=models\model_fan_clsf\bundle -backend=CPU -target=arm -mcpu=cortex-m7 -float-abi=hard -use-cmsis
```
4.2.3 Model deployment and evaluation on the embedded device

Once the models were generated, you can use the provided `mcu_app` application and the MCUXpressoIDE to deploy, run, and evaluate them on the embedded board.

The source code structure is highlighted in Figure 24 and includes:

- `mcu_app/doc/readme.txt`: short documentation for the provided application
  Refer to this document for guidelines on hardware requirements, board settings, and running the demo.
- `mcu_app/board/board.h`: contains settings related to board configuration
- `mcu_app/board/frdm_stbc_agm01_shield.h`: contains settings related to the used sensors.
  Enable the sensor shield if the optional sensor toolbox (FRMD-STBC-AGM01) is used.
- `mcu_app/source/inf-eng`: contains the APIs used to run the inference engines
- `mcu_app/source/models`: contains the models ported for every specific format plus validation data to be used for testing.
  The models ported by following the guidelines from Model porting for embedded device must be stored in this directory.
- `mcu_app/source/models/model_selection.h`: must be configured by the user and selects between quantized and non-quantized versions and how to load the model (from Flash or from RAM).
  If the glow quantized version is selected to be used the name of the model must also be configured in the project settings by following this path: right click on the project> Properties> C/C++ Build> Settings> Tool Settings> MCU C++ Linker> Miscellaneous> Other Objects.
- `mcu_app/source/models/validation_data/`: contains the prerecorded data for offline validation
- `mcu_app/source/sensor/sensor_collect.c`: contains the main thread
- `mcu_app/source/sensor/sensor_collect.h`: must be configured by the user and selects the application runtime behavior:
  - Log externally the sensor data:

```c
/* Configure the action to be performed */
#define SENSOR_COLLECT_ACTION SENSOR_COLLECT_LOG_EXT
#if SENSOR_COLLECT_ACTION == SENSOR_COLLECT_LOG_EXT
#define SENSOR_COLLECT_LOG_EXT_SDCARD 1 // Redirect the log to SD card
```
Run the selected inference engine and compute predictions either on real-time data, or on prerecorded validation data (the `SENSOR_FEED_VALIDATION_DATA` flag toggles between real-time or offline validation):

```c
/* Configure the action to be performed */
#define SENSOR_COLLECT_ACTION                   SENSOR_COLLECT_RUN_INFERENCE
#define SENSOR_COLLECT_RUN_INFENG               SENSOR_COLLECT_INFENG_TENSORFLOW
#define SENSOR_FEED_VALIDATION_DATA             1   // Feed the model with data recorded previously for validation
#define SENSOR_RAW_DATA_NORMALIZE               1   // Normalize the raw data
#define SENSOR_EVALUATE_MODEL                   1   // Evaluate the model performance by computing the accuracy
#define SENSOR_COLLECT_INFENG_VERBOSE_EN        0   // Enable verbosity
```

When offline validation is configured, the prerecorded data from the `validation_data/` directory will be used for evaluation. Depending on which class is evaluated, the pointer which references the validation data needs to be configured in `sensor_collect.c`

```c
/* Replace with the buffer that contains the data recorded * for the specific class that will be evaluated * (i.e., vdset_3_vd3_clog, vdset_3_vd3_friction, * vdset_3_vd3_off, vdset_3_vd3_on) */
static const float *vdset_ptr = &vdset_3_vd3_clog[0][0];
```

Figure 24. Source code structure
After the settings have been configured the application can be programmed on the target board and then launched by pressing the reset button or by executing the debugger in the IDE. After the board is flashed, the terminal prints "Starting Application..." and the application runs. Example output:

```
Starting Application...
MainTask started
SENSOR_Collect_Task started
Model loaded to SDRAM...

Model Evaluation:
Class to evaluate (provide only the numeric index):
{ 0:FAN-CLOG 1:FAN-FRICTION 2:FAN-OFF 3:FAN-ON }
    >>> 0
Pool size (total number of predictions to compute):
    >>> 936

Inference 0?0 | t 825 us | count: 732/936/936 | FAN-CLOG
Prediction Accuracy for class FAN-CLOG 78.21%
```

The accuracy is computed by dividing the number of correct predictions to the total number of predictions. When the model is evaluated on the board, two parameters are required to be provided:

1. **Target class** – specifies which class will be evaluated and for which class the accuracy will be computed (That is, 0 for Fan-Clog, 1 for Fan-Friction, 2 for Fan-OFF, 3 for Fan-ON).
2. **Pool size** – the total number of predictions that will be computed. In the above snapshot, the Clog state was evaluated from a pool of 936 input tensors. The provided pool size value is actually the size of the last validation dataset recorded in session 3 (vdset_3_vd3_clog.h) and exported for offline validation on the board:

```
Pool size = 936 (60000/64 - 1)
Number of samples in the dataset = 60000 (5minutes of recording at 200Hz)
Moving Window size = 64 samples
Remainder = 1
```

Also, the results obtained when evaluating offline the prerecorded data on the board must match the results obtained by running the script on the host machine evaluating the same batch of data, like in Figure 25 and Figure 26.

**NOTE**

A slight deviation in the computed accuracy (up to 0.3 %) can be observed when running predictions with the float models versus quantized models. This behavior is totally expected as the quantization is a lossy process that converts the model to smaller precision data.
Figure 26. Model evaluation on the host machine using prerecorded validation data.

Table 2 presents the results obtained by running and evaluating the models on the i.MXRT1170-EVK (Cortex-M7 core). Figure 27 and Figure 28 present through charts the benchmark results.

The **Inference time** is the average duration to process an input tensor and represents how fast the embedded board computes the predictions and classifies one single state of the device. This duration has been evaluated by loading the model from RAM or from Flash memory with the processing core configured to run at a higher frequency of 996 MHz or at a lower frequency of 156 MHz.

The memory usage is presented through the **Model size** and **Code size** for every inference engine.

The model size consists of constant parameters which reside in Flash memory and can also be loaded from the RAM memory for faster inference time, and mutable weights which reside in RAM.

Some noticeable differences can be observed on the inference engines memory footprint mainly because the model is very small, but this ratio would be less relevant for larger models.
Table 2. Model evaluation on the embedded board

<table>
<thead>
<tr>
<th></th>
<th>TFLite (no quant)</th>
<th>TFLite (quant)</th>
<th>DeepViewRT (no quant)</th>
<th>DeepViewRT (quant)</th>
<th>Glow (no quant)</th>
<th>Glow (quant)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InferenceTime (ms)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>@996 MHz</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>0.74</td>
<td>0.48</td>
<td>1.16</td>
<td>1.14</td>
<td>0.35</td>
<td>0.17</td>
</tr>
<tr>
<td>Flash</td>
<td>1.46</td>
<td>0.55</td>
<td>1.77</td>
<td>1.31</td>
<td>0.98</td>
<td>0.23</td>
</tr>
<tr>
<td><strong>@156 MHz</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>4.89</td>
<td>1.92</td>
<td>3.50</td>
<td>4.32</td>
<td>2.19</td>
<td>1.08</td>
</tr>
<tr>
<td>Flash</td>
<td>5.31</td>
<td>1.93</td>
<td>3.80</td>
<td>4.44</td>
<td>2.42</td>
<td>1.09</td>
</tr>
<tr>
<td><strong>Model size (KB)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash/RAM (const)</td>
<td>43.5</td>
<td>15.4</td>
<td>44.3</td>
<td>15.4</td>
<td>40.5</td>
<td>10.6</td>
</tr>
<tr>
<td>RAM (var)</td>
<td>9.4</td>
<td>4.5</td>
<td>11.3</td>
<td>6.3</td>
<td>9.6</td>
<td>3.7</td>
</tr>
<tr>
<td><strong>Code size (KB):</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash</td>
<td>56.3</td>
<td>60.9</td>
<td>109.2</td>
<td>109.2</td>
<td>10.9</td>
<td>10.4</td>
</tr>
</tbody>
</table>

1. The evaluated software version for DeepViewRT inference engine will be released by the end of Q1-2022

Figure 27. Model performance - inference time
Table 3 and Figure 29 present the inference time evaluated by running the model on multiple embedded devices with the processing cores configured to run at 150 MHz.

Table 3. Inference-time evaluated on multiple embedded devices

<table>
<thead>
<tr>
<th>Device</th>
<th>Inference Time (ms) @150 MHz</th>
<th>TFLite (no quant)</th>
<th>TFLite (quant)</th>
<th>Glow (no quant)</th>
<th>Glow (quant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIMXRT1170-EVK</td>
<td>4.89</td>
<td>1.92</td>
<td>2.19</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>(Arm Cortex-M7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRDM-K66F</td>
<td>8.87</td>
<td>4.18</td>
<td>4.24</td>
<td>5.47</td>
<td></td>
</tr>
<tr>
<td>(Arm Cortex-M4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPC55S69-EVK</td>
<td>9.95</td>
<td>5.77</td>
<td>4.16</td>
<td>3.88</td>
<td></td>
</tr>
<tr>
<td>(Arm Cortex-M33)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Expected higher inference time for Glow quantized model versus float model because the Glow model-compiler does not support the CMSIS-NN acceleration for Arm Cortex-M4 at this point.
5 Conclusion

This Application Note provides the guidelines that can be followed to build and run a Smart Sensing Appliance on an MCU, relying on Deep Learning to solve a specific problem. For that purpose, this document exemplifies through a real use case the steps required to produce and assemble a dataset, define the architecture of a neural network, train, and deploy a model on an embedded board by leveraging the NXP’s SDK and the eIQ technology.

This document also shows which metrics to use and how to evaluate the behavior of a neural network model at runtime on an embedded board, as well as benchmarks and performance results.

6 Revision history

The table below summarizes the changes done to the document since the initial release.

<table>
<thead>
<tr>
<th>Revision number</th>
<th>Date</th>
<th>Substantive changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25 April 2022</td>
<td>Added inference-time benchmarks for MIMXRT1170-EVK (CM7), FRDM-K66F (CM4), and LPC55S69-EVK (CM33)</td>
</tr>
<tr>
<td>0</td>
<td>03 February 2022</td>
<td>Initial release</td>
</tr>
</tbody>
</table>
Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors’ aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer’s own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer’s sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer’s applications and products planned, as well as for the planned application and use of customer’s third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer’s applications or products, or the application or use by customer’s third party customer(s). Customer is responsible for doing all necessary testing for the customer’s applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer’s third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer’s general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors’ warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors’ specifications such use shall be solely at customer’s own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors’ standard warranty and NXP Semiconductors’ product specifications.
Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKPro, μVision, Versatile — are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

Airstar — is a trademark of NXP B.V.

Bluetooth — the Bluetooth wordmark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by NXP Semiconductors is under license.

Cadence — the Cadence logo, and the other Cadence marks found at www.cadence.com/go/trademarks are trademarks or registered trademarks of Cadence Design Systems, Inc. All rights reserved worldwide.

CodeWarrior — is a trademark of NXP B.V.

ColdFire — is a trademark of NXP B.V.

ColdFire+ — is a trademark of NXP B.V.

EdgeLock — is a trademark of NXP B.V.

EdgeScale — is a trademark of NXP B.V.

EdgeVerse — is a trademark of NXP B.V.

eIQ — is a trademark of NXP B.V.

FeiCa — is a trademark of Sony Corporation.

Freescale — is a trademark of NXP B.V.

HITAG — is a trademark of NXP B.V.

ICODE and I-CODE — are trademarks of NXP B.V.

Immersiv3D — is a trademark of NXP B.V.

I2C-bus — logo is a trademark of NXP B.V.

Kineto — is a trademark of NXP B.V.

Layerscape — is a trademark of NXP B.V.

Mantis — is a trademark of NXP B.V.

MIFARE — is a trademark of NXP B.V.

MOBILEGT — is a trademark of NXP B.V.

NTAG — is a trademark of NXP B.V.

Processor Expert — is a trademark of NXP B.V.

QorIQ — is a trademark of NXP B.V.

SafeAssure — is a trademark of NXP B.V.

SafeAssure — logo is a trademark of NXP B.V.

StarCore — is a trademark of NXP B.V.

Synopsys — Portions Copyright © 2021 Synopsys, Inc. Used with permission. All rights reserved.

Tower — is a trademark of NXP B.V.

UCODE — is a trademark of NXP B.V.

VortiQa — is a trademark of NXP B.V.