AN13562

Building and Benchmarking Deep Learning Models for Smart Sensing Appliances on MCUs

Rev. 2 — 27 September 2023 Application note

Document Information

Information	Content
Keywords	Artificial Inteligence, AI, Machine learning, ML, Deep Learning, DL, Neural Networks eIQ, Glow, DeepViewRT, TensorFlow reference applications, embedded software, i.MX RT crossovers, MCU smart appliances, smart sensing, state monitor, fan state classifier, anomaly detection activity recognition, sensors, vibration, acceleration, time series
Abstract	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.

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 using 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 uses the NXP 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, with runtime inference engines, neural network compilers, and optimized libraries. This software uses 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.

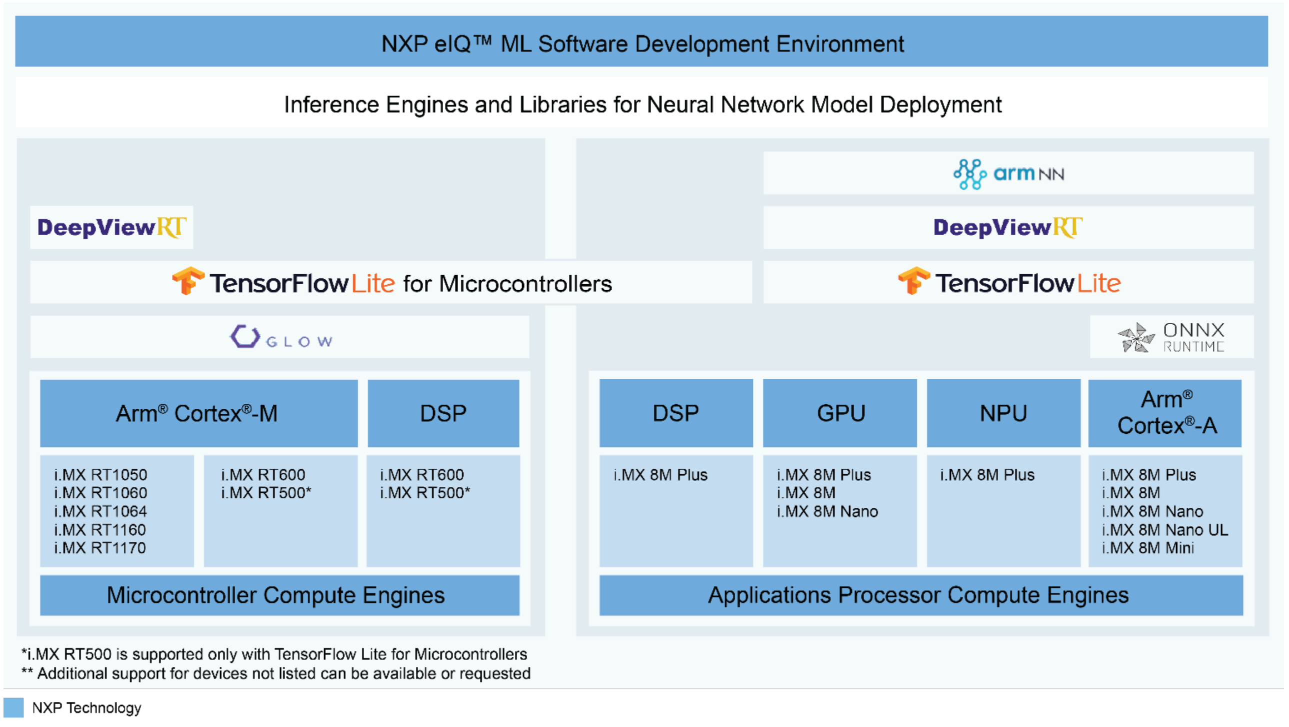

NXP eIQ™ ML Software Development Environment
Inference Engines and Libraries for Neural Network Model Deployment
arm NN
DeepViewRT
DeepViewRT
TensorFlow Lite for Microcontrollers
TensorFlow Lite
GLOW
ONNX RUNTIME
Arm® Cortex®-M
DSP
DSP
GPU
NPU
Arm® Cortex®-A
i.MX RT1050
i.MX RT1060
i.MX RT1064
i.MX RT1160
i.MX RT1170
i.MX RT600
i.MX RT500*
i.MX RT600
i.MX RT500*
i.MX 8M Plus
i.MX 8M Plus
i.MX 8M
i.MX 8M Nano
i.MX 8M Plus
i.MX 8M Plus
i.MX 8M
i.MX 8M Nano
i.MX 8M Nano UL
i.MX 8M Mini
Microcontroller Compute Engines
Applications Processor Compute Engines
*i.MX RT500 is supported only with TensorFlow Lite for Microcontrollers
** Additional support for devices not listed can be available or requested
NXP Technology

Figure 1. eIQ software ecosystem

The runtime inference engines included in the eIQ component of the MCUXpresso Software Development Kit (SDK) for microcontrollers, which is evaluated in this document, are counted 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 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

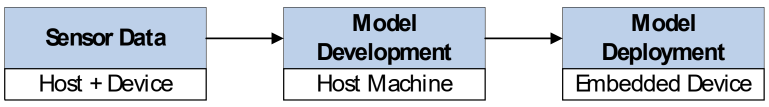


Figure 2. Application development workflow

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
   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 the deep learning frameworks available. The model is developed on the host machine by going through the phases shown in Figure 3.

:

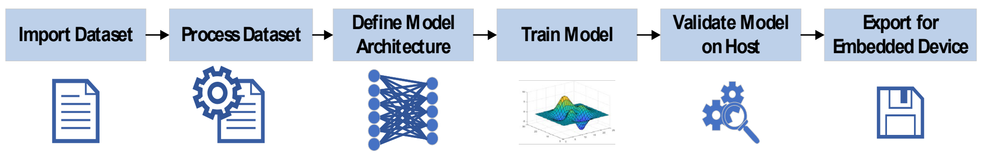


Figure 3. Model development workflow

3. Model deployment

The model created in the previous phase is deployed and evaluated on the embedded device through the runtime inference engines defined in Section 2.

4 Going through a real use case

This chapter follows the workflow diagrams in Section 3 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 Prerequisite

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 application, developed in MCUXpresso IDE.

Software requirements on the host machine:

- ML-based system state monitor source code.
- Jupyter Notebook (Python, TensorFlow, Keras, and so on).
- 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 or newer version FRDM-K22F-A8974
  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 using 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*.

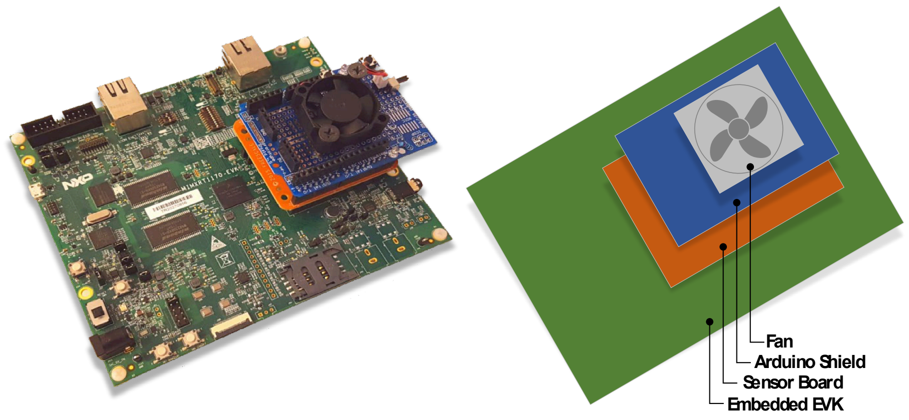


Figure 4. Stacked embedded setup

4.2 Use case

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.

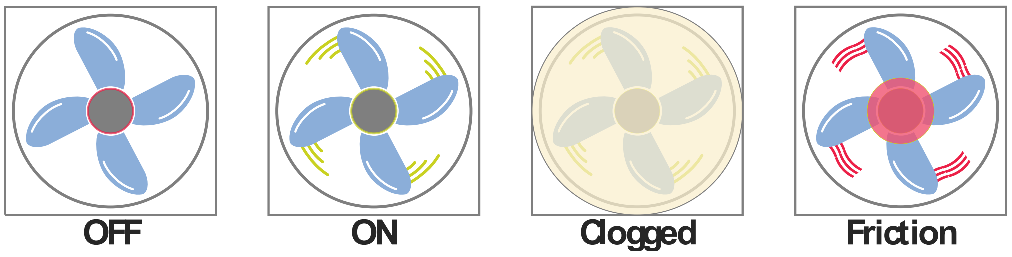


Figure 5. Fan state classifier

4.3 Applications

The development workflow for the presented use case is exemplified with two application versions, v1 and v2.

4.3.1 Application v1

The application v1 consists of two main components:

- *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.

The MCU application v1 provides a manually crafted pipeline for Time Series data with:

- The accelerometer sensor, FXOS8700
- One CNN model

The components are available at this location:

```
--ML Application:   ${repository_location}\ml_state_monitor\ml_app
\v1\ML_State_Monitor.ipynb
--MCU Applications:
  ---${repository_location}\ml_state_monitor\mcu_app\boards\
${board}\ml_state_monitor_mpp\
  --- evkmimxrt1170_ml_state_monitor_cm7, lpcxpresso55s69_ml_state_monitor_cm33,
 frdmk66f_ml_state_monitor
```

4.3.2 Application v2 with MPP (Multimedia Processing Pipeline)

The application v2 also consists of two components:

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

The MCU application v2 is an enhanced version providing the eIQ MCU Multimedia Processing Pipeline (MPP) library with exploratory support for Time Series data, supporting:

- The new accelerometer sensor, FXLS8974CF
- Four NN models with different performance metric targets

The advantage of the pipeline library comes in terms of integration ease and code reusability, providing a simple API that integrates:

- Pipeline capabilities
- Accelerometer driver
- SD card driver
- Inference engines

The components are available at this location:

```
--ML Application: ${repository_location}\ml_state_monitor\ml_app
\v2\ML_State_Monitor.ipynb
--MCU Applications:
    ---${repository_location}\ml_state_monitor\mcu_app\boards\
${board}\ml_state_monitor_mpp\
    ---evkmimxrt1170_ml_state_monitor_mpp_cm7,
 lpcxpresso55s69_ml_state_monitor_mpp_cm33
```

4.4 Case Study for Application v1

4.4.1 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

Sensor	FXOS8700 (6-axis sensor) 3-axis 14-bit accelerometer 3-axis 16-bit magnetometer
Used Channels (Nch)	Three channels (3-axis accelerometer)
Sampling Frequency (Fs)	200 Hz (5 ms)

Table 1. Sensor data specifications*...continued*

Input-Output data Type/Size	float 32/4 bytes (per each individual channel)		
Sample size (s)	Nch (each recorded sample includes all monitored channels)		
Window Size (w)	128 samples (640 ms)		
Overlap Ratio	50 % - 64 samples (320 ms)		
Input Tensor Shape	(w, 1, Nch) = (128, 1, 3)		
Input Tensor Size	1.5 KB (w * Nch * 4 bytes)		
Data Stream Format	Interleaved		
Training Data Collection **Input Tensors Count (Nt)** **Duration**	5.76 Ms 90 Kw 8 h	1.44 Ms/class 22 Kw/class 2 h/class	[**M**ega**S**ample] [**K**ilo**W**indow] [time]
Validation Data Collection **Input Tensors Count (Nt)** **Duration**	1.68 Ms 26 Kw 2 h 20 min	0.42 Ms/class 6 Kw/clas 35 min/class	[**M**ega**S**ample] [**K**ilo**W**indow] [time]

To feed the neural network, 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. For example, the first three tensors processed by the neural network have the following format:

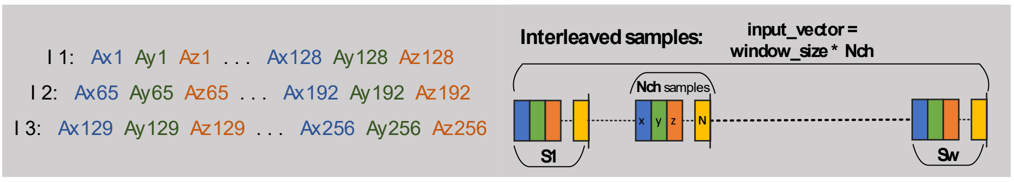

I 1: Ax1 Ay1 Az1 . . . Ax128 Ay128 Az128
I 2: Ax65 Ay65 Az65 . . . Ax192 Ay192 Az192
I 3: Ax129 Ay129 Az129 . . . Ax256 Ay256 Az256
Interleaved samples:
input_vector = window_size * Nch
Nch samples
x
y
z
N
S1
Sw

Figure 6. Interleaved data stream format

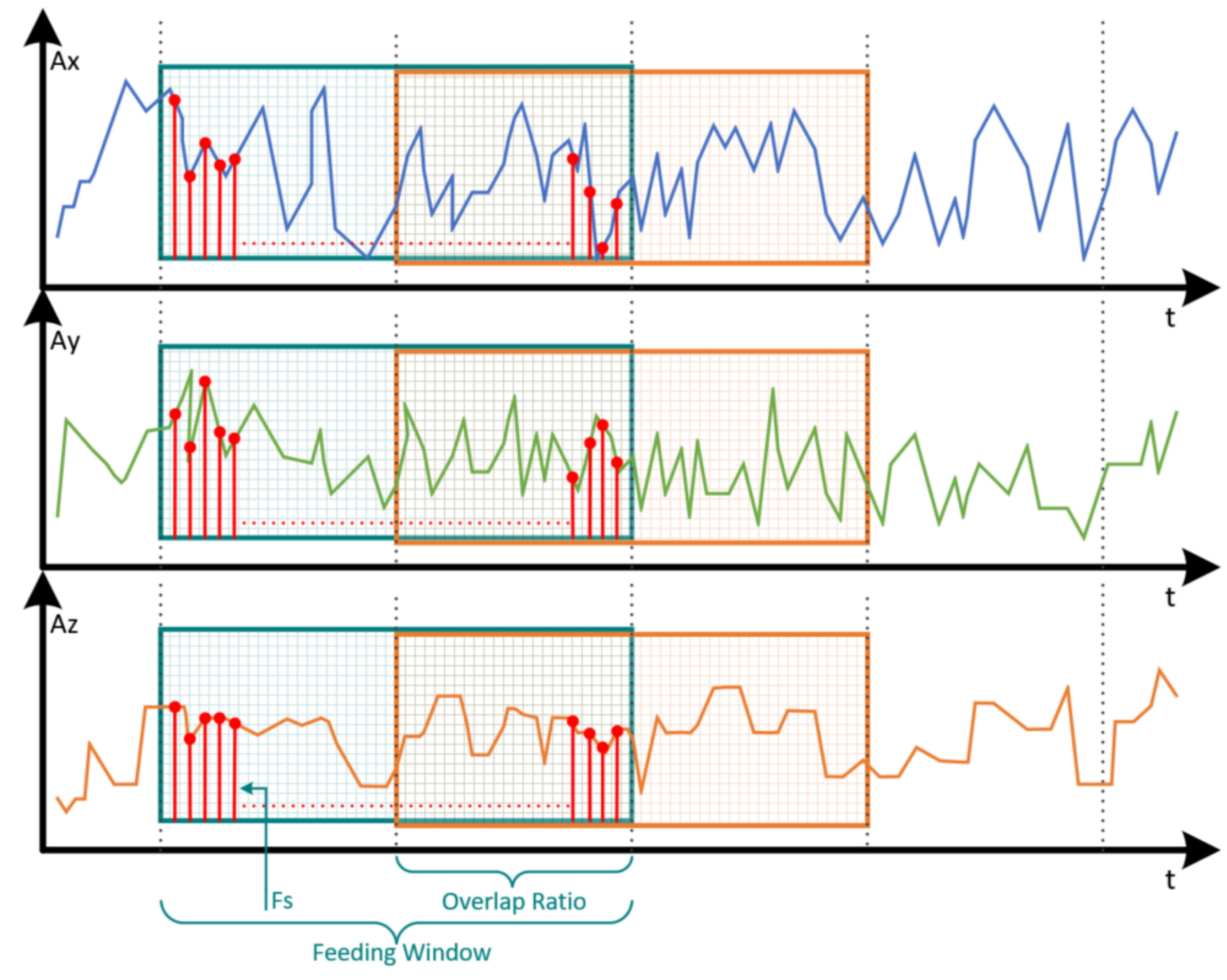


Figure 7. Model feeding using interleaved samples and moving window

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. Therefore, we cannot use datasets already available, and we must create (collect and annotate) a dataset relevant for this particular setup.

To collect the dataset, enable the embedded application to log the sensor data (on the SD card or over the serial debugging interface) and provide the required configuration on the terminal.

```
/* 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 communicates (log information, sensor data, input configuration) with the host machine through the Debug Console Interface (UART over USB, 115,200 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 to 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.

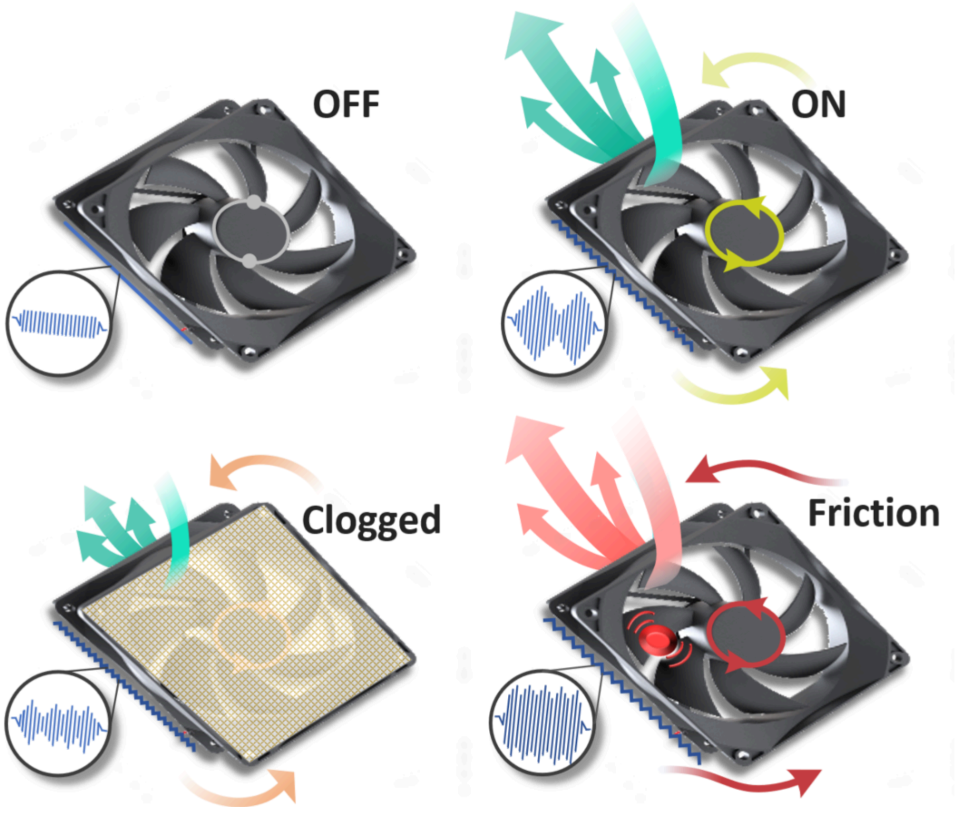


Figure 8. Setup configured to record each class

One session of recording is presented in Figure 9 where the recording was started by collecting a validation dataset for a short period (five minutes), followed by a dataset for training and validation collected for a longer period (one hour + five minutes), and ending with a validation dataset collected again for a short period (five minutes). 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.

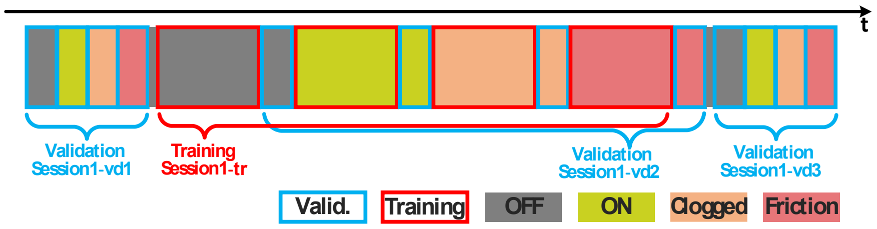
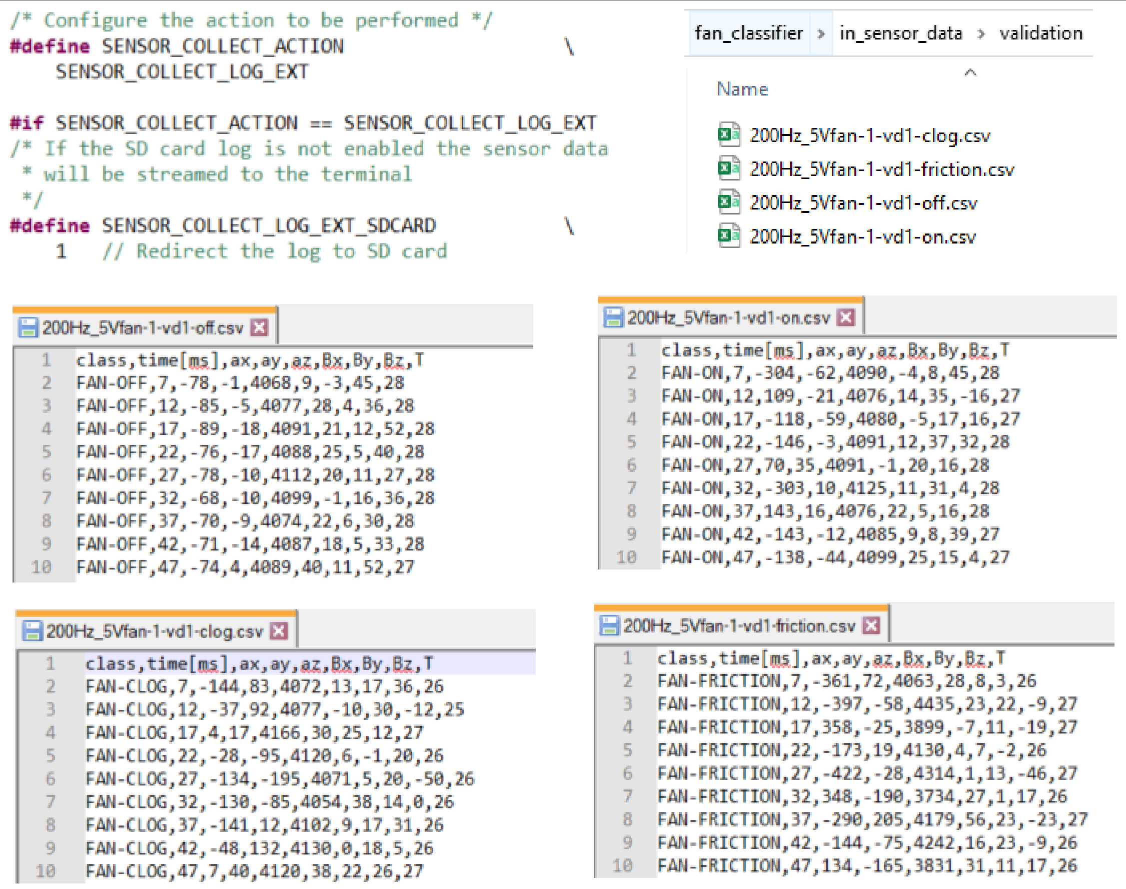


Figure 9. Timeline for one session of data recording

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.


```
/* Configure the action to be performed */
#define SENSOR_COLLECT_ACTION                      \
    SENSOR_COLLECT_LOG_EXT

#if SENSOR_COLLECT_ACTION == SENSOR_COLLECT_LOG_EXT
/* If the SD card log is not enabled the sensor data
 * will be streamed to the terminal
 */
#define SENSOR_COLLECT_LOG_EXT_SDCARD              \
    1   // Redirect the log to SD card
```

fan_classifier › in_sensor_data › validation
Name
200Hz_5Vfan-1-vd1-clog.csv
200Hz_5Vfan-1-vd1-friction.csv
200Hz_5Vfan-1-vd1-off.csv
200Hz_5Vfan-1-vd1-on.csv
200Hz_5Vfan-1-vd1-off.csv
200Hz_5Vfan-1-vd1-on.csv
200Hz_5Vfan-1-vd1-clog.csv
200Hz_5Vfan-1-vd1-friction.csv

```
class,time[ms],ax,ay,az,Bx,By,Bz,T
FAN-OFF,7,-78,-1,4068,9,-3,45,28
FAN-OFF,12,-85,-5,4077,28,4,36,28
FAN-OFF,17,-89,-18,4091,21,12,52,28
FAN-OFF,22,-76,-17,4088,25,5,40,28
FAN-OFF,27,-78,-10,4112,20,11,27,28
FAN-OFF,32,-68,-10,4099,-1,16,36,28
FAN-OFF,37,-70,-9,4074,22,6,30,28
FAN-OFF,42,-71,-14,4087,18,5,33,28
FAN-OFF,47,-74,4,4089,40,11,52,27
```

```
class,time[ms],ax,ay,az,Bx,By,Bz,T
FAN-ON,7,-304,-62,4090,-4,8,45,28
FAN-ON,12,109,-21,4076,14,35,-16,27
FAN-ON,17,-118,-59,4080,-5,17,16,27
FAN-ON,22,-146,-3,4091,12,37,32,28
FAN-ON,27,70,35,4091,-1,20,16,28
FAN-ON,32,-303,10,4125,11,31,4,28
FAN-ON,37,143,16,4076,22,5,16,28
FAN-ON,42,-143,-12,4085,9,8,39,27
FAN-ON,47,-138,-44,4099,25,15,4,27
```

```
class,time[ms],ax,ay,az,Bx,By,Bz,T
FAN-CLOG,7,-144,83,4072,13,17,36,26
FAN-CLOG,12,-37,92,4077,-10,30,-12,25
FAN-CLOG,17,4,17,4166,30,25,12,27
FAN-CLOG,22,-28,-95,4120,6,-1,20,26
FAN-CLOG,27,-134,-195,4071,5,20,-50,26
FAN-CLOG,32,-130,-85,4054,38,14,0,26
FAN-CLOG,37,-141,12,4102,9,17,31,26
FAN-CLOG,42,-48,132,4130,0,18,5,26
FAN-CLOG,47,7,40,4120,38,22,26,27
```

```
class,time[ms],ax,ay,az,Bx,By,Bz,T
FAN-FRICTION,7,-361,72,4063,28,8,3,26
FAN-FRICTION,12,-397,-58,4435,23,22,-9,27
FAN-FRICTION,17,358,-25,3899,-7,11,-19,27
FAN-FRICTION,22,-173,19,4130,4,7,-2,26
FAN-FRICTION,27,-422,-28,4314,1,13,-46,27
FAN-FRICTION,32,348,-190,3734,27,1,17,26
FAN-FRICTION,37,-290,205,4179,56,23,-23,27
FAN-FRICTION,42,-144,-75,4242,16,23,-9,26
FAN-FRICTION,47,134,-165,3831,31,11,17,26
```

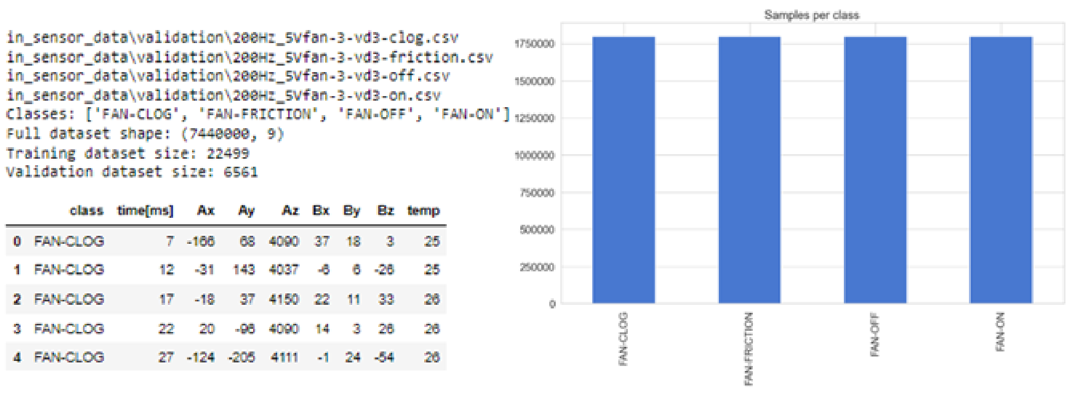
Figure 10. Data collected in one session

4.4.2 Model development

The model development flow described in Application development workflow and Figure 3 is applied to the case study by running on the host machine with Jupyter notebook provided for Application v1.

4.4.2.1 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.



```
in_sensor_data\validation\200Hz_5Vfan-3-vd3-clog.csv
in_sensor_data\validation\200Hz_5Vfan-3-vd3-friction.csv
in_sensor_data\validation\200Hz_5Vfan-3-vd3-off.csv
in_sensor_data\validation\200Hz_5Vfan-3-vd3-on.csv
Classes: ['FAN-CLOG', 'FAN-FRICTION', 'FAN-OFF', 'FAN-ON']
Full dataset shape: (7440000, 9)
Training dataset size: 22499
Validation dataset size: 6561
```

	class	time[ms]	Ax	Ay	Az	Bx	By	Bz	temp
0	FAN-CLOG	7	-166	68	4090	37	18	3	25
1	FAN-CLOG	12	-31	143	4037	-6	6	-26	25
2	FAN-CLOG	17	-18	37	4150	22	11	33	26
3	FAN-CLOG	22	20	-96	4090	14	3	26	26
4	FAN-CLOG	27	-124	-205	4111	-1	24	-54	26

Figure 11. Dataset summary

4.4.2.2 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). To anylize the real-time data, plot the data collected at real time 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 a 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. However, 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.

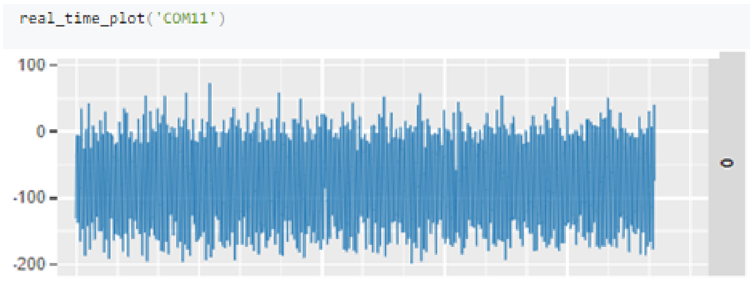

```
real_time_plot('COM11')
```

Figure 12. Real-time data plot

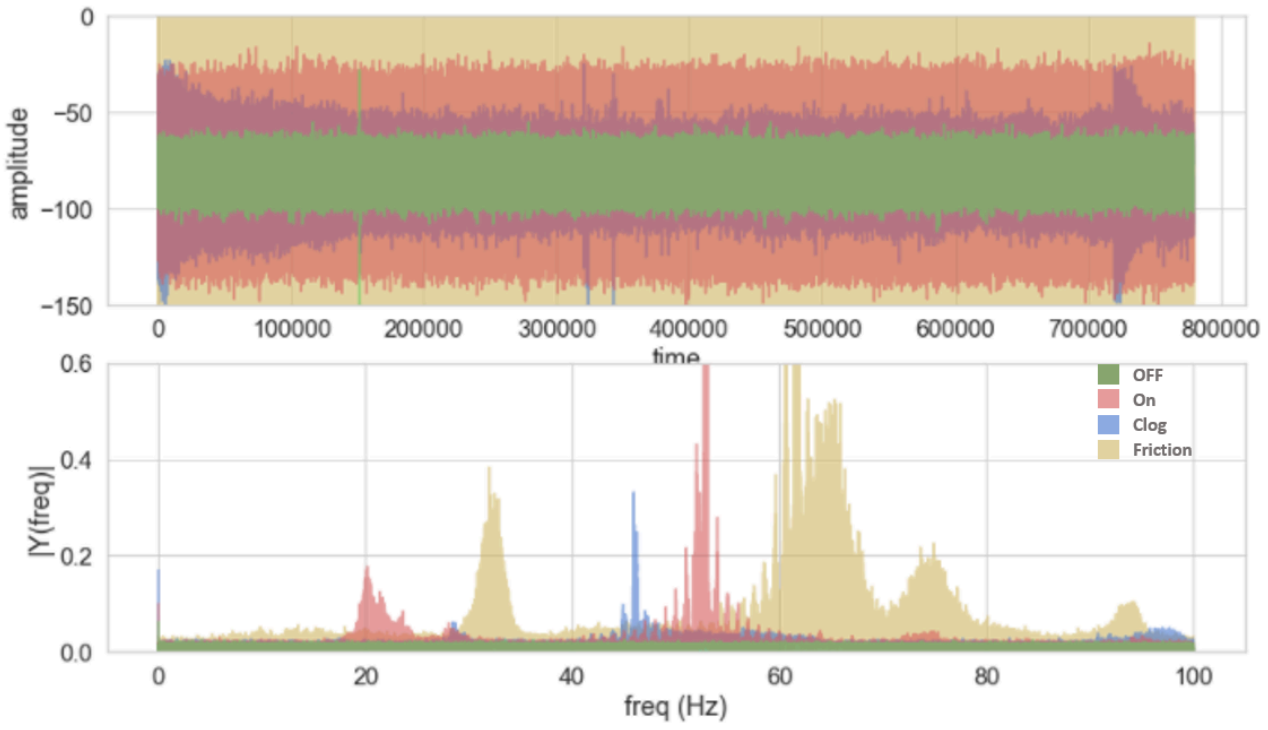


Figure 13. Data analysis in time and frequency domains

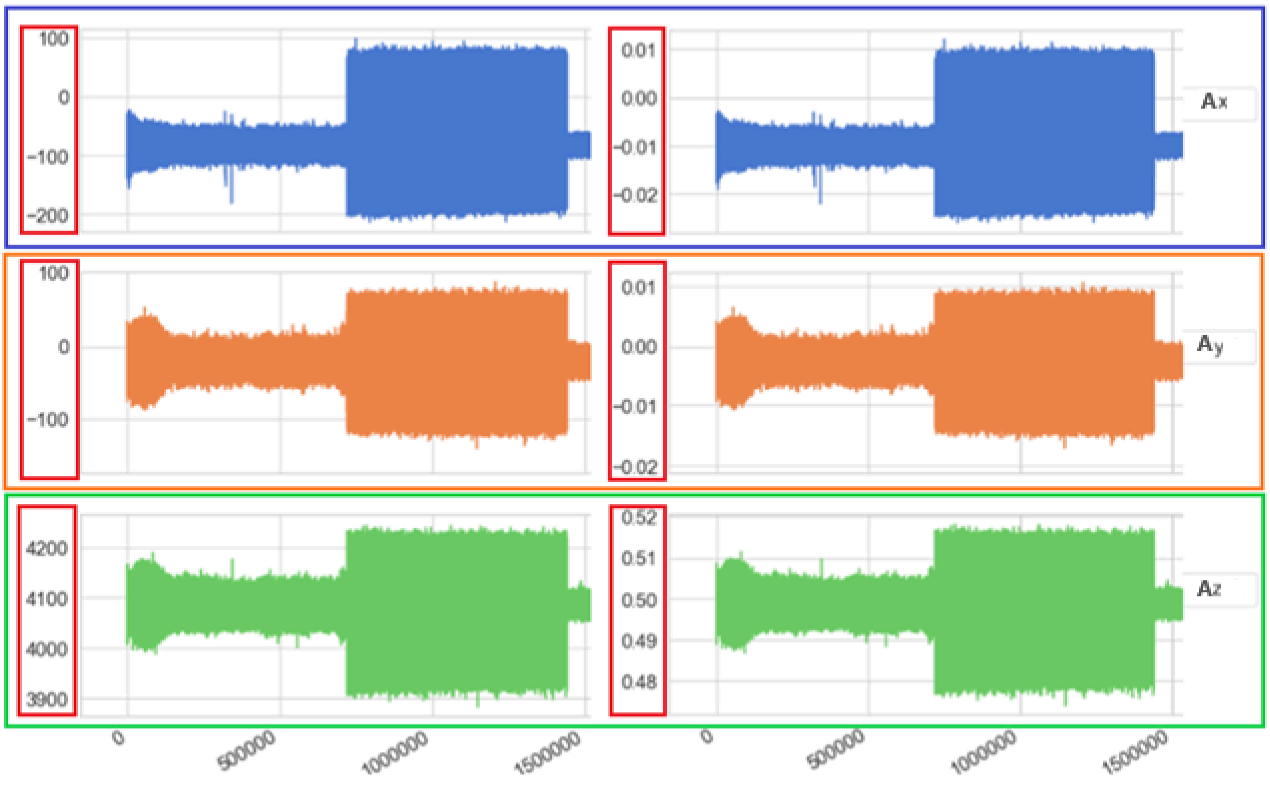


Figure 14. Dataset normalization

The step of data reshaping is mandatory 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:

```
dataset = (Nt, height, width, depth) = (Nt, window, 1, Nch) = (Nt, 128, 1, 3)
Nt - the total number of input tensors
```

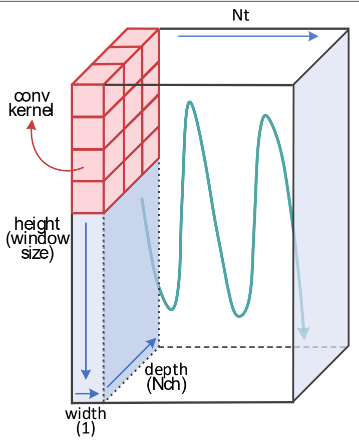


Figure 15. Shape of dataset and input tensor

4.4.2.3 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.

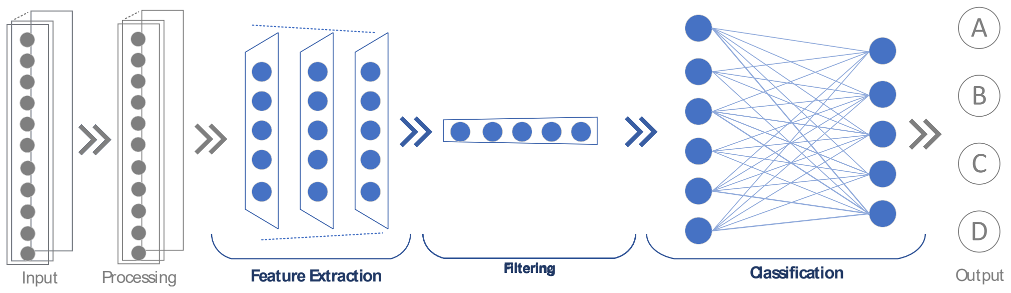
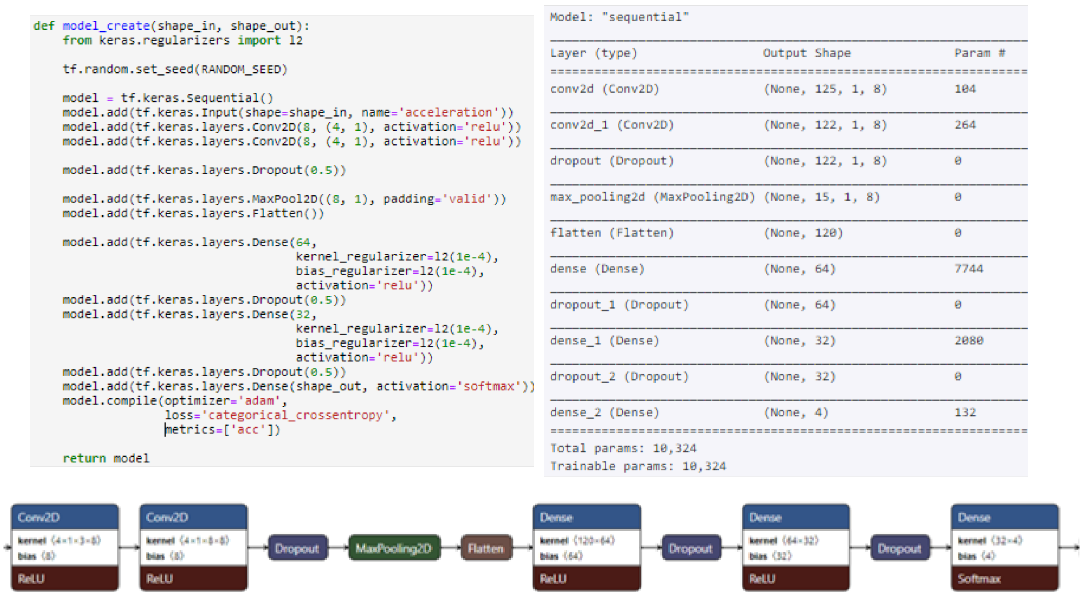


Figure 16. Neural network architecture

The network is implemented using 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. They 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 iteratively 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.

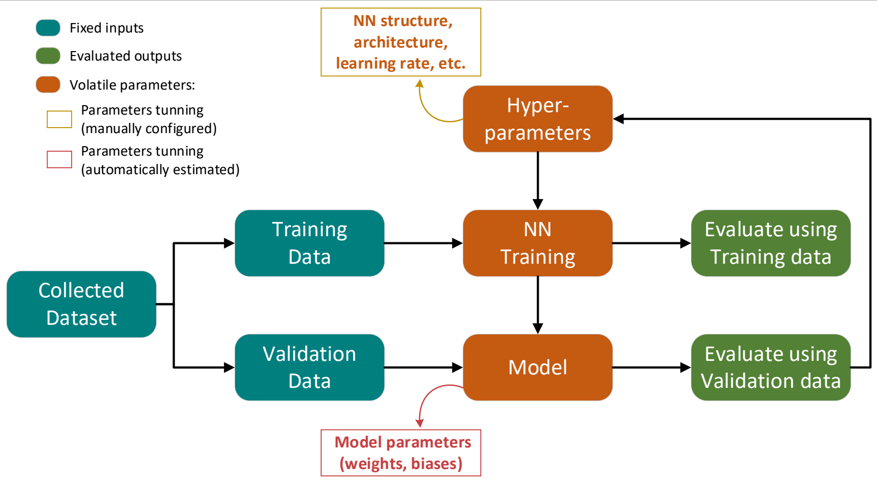


```python
def model_create(shape_in, shape_out):
    from keras.regularizers import l2

    tf.random.set_seed(RANDOM_SEED)

    model = tf.keras.Sequential()
    model.add(tf.keras.Input(shape=shape_in, name='acceleration'))
    model.add(tf.keras.layers.Conv2D(8, (4, 1), activation='relu'))
    model.add(tf.keras.layers.Conv2D(8, (4, 1), activation='relu'))

    model.add(tf.keras.layers.Dropout(0.5))

    model.add(tf.keras.layers.MaxPool2D((8, 1), padding='valid'))
    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(64,
                                    kernel_regularizer=l2(1e-4),
                                    bias_regularizer=l2(1e-4),
                                    activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(32,
                                    kernel_regularizer=l2(1e-4),
                                    bias_regularizer=l2(1e-4),
                                    activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(shape_out, activation='softmax'))
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['acc'])

    return model
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 125, 1, 8)         104
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 122, 1, 8)         264
_________________________________________________________________
dropout (Dropout)            (None, 122, 1, 8)         0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 1, 8)          0
_________________________________________________________________
flatten (Flatten)            (None, 120)               0
_________________________________________________________________
dense (Dense)                (None, 64)                7744
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 132
=================================================================
Total params: 10,324
Trainable params: 10,324
```

Figure 17. Model structure and implementation

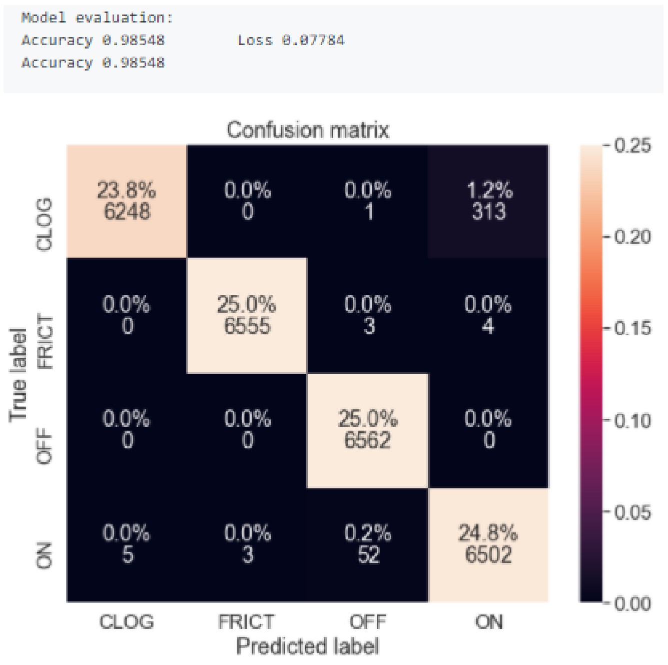
Once the network architecture is defined, the next step is to start the training process. Through training, the network learns from the provided training dataset and automatically configures the model parameters. The model parameters specify how to transform the input into the desired output. In addition, 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.



Fixed inputs
Evaluated outputs
Volatile parameters:
Parameters tunning (manually configured)
Parameters tunning (automatically estimated)
NN structure, architecture, learning rate, etc.
Hyper-parameters
Training Data
NN Training
Evaluate using Training data
Collected Dataset
Validation Data
Model
Evaluate using Validation data
Model parameters (weights, biases)

Figure 18. Model training and parameters tuning

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. To evaluate the training session, plot and analyze the history of the progress over iterations, as shown in Figure 20.



```
Model evaluation:
Accuracy 0.98548        Loss 0.07784
Accuracy 0.98548
```

Figure 19. Model evaluation using confusion matrix

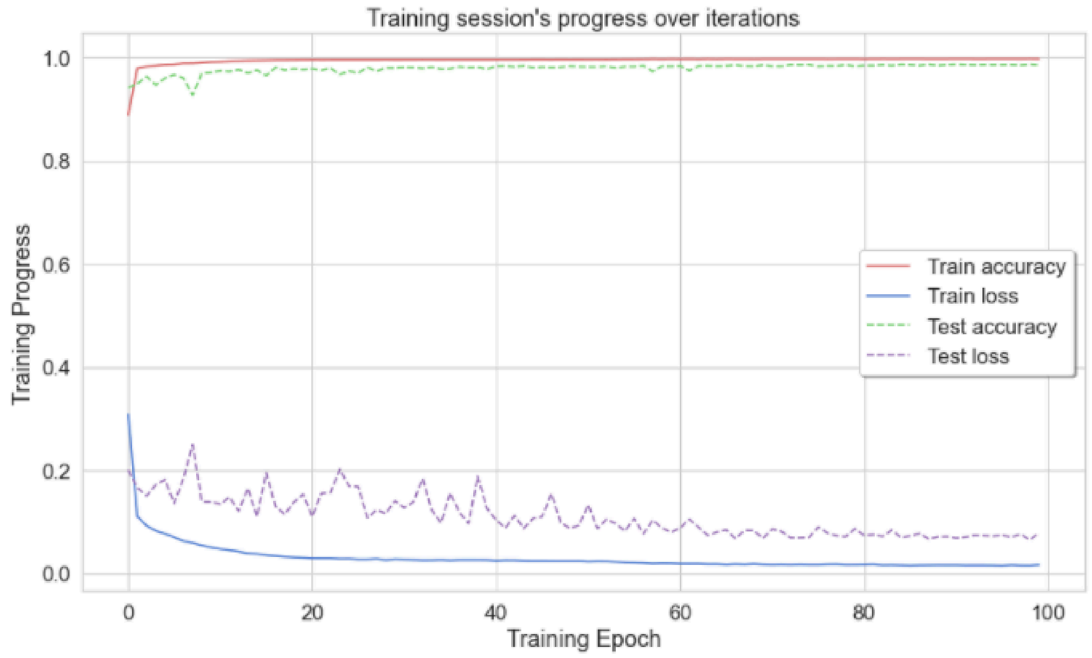
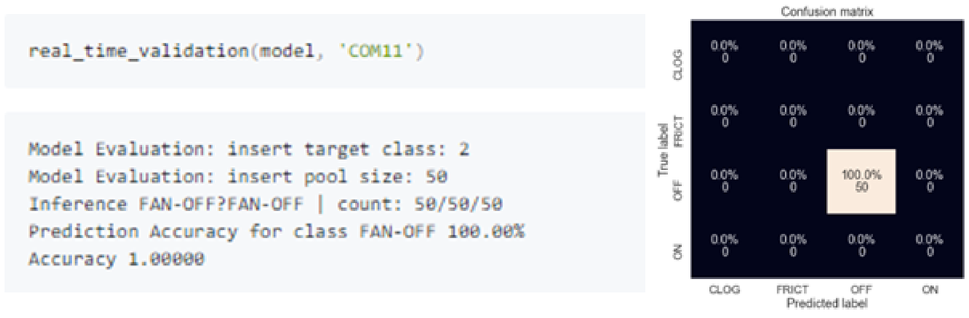


Figure 20. Training session evaluation

4.4.2.4 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). Besides running it on data previously recorded, you can evaluate the model by 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, connect the board to a host machine and configure to log externally sensor data via the console.



```
real_time_validation(model, 'COM11')
```

```
Model Evaluation: insert target class: 2
Model Evaluation: insert pool size: 50
Inference FAN-OFF?FAN-OFF | count: 50/50/50
Prediction Accuracy for class FAN-OFF 100.00%
Accuracy 1.00000
```

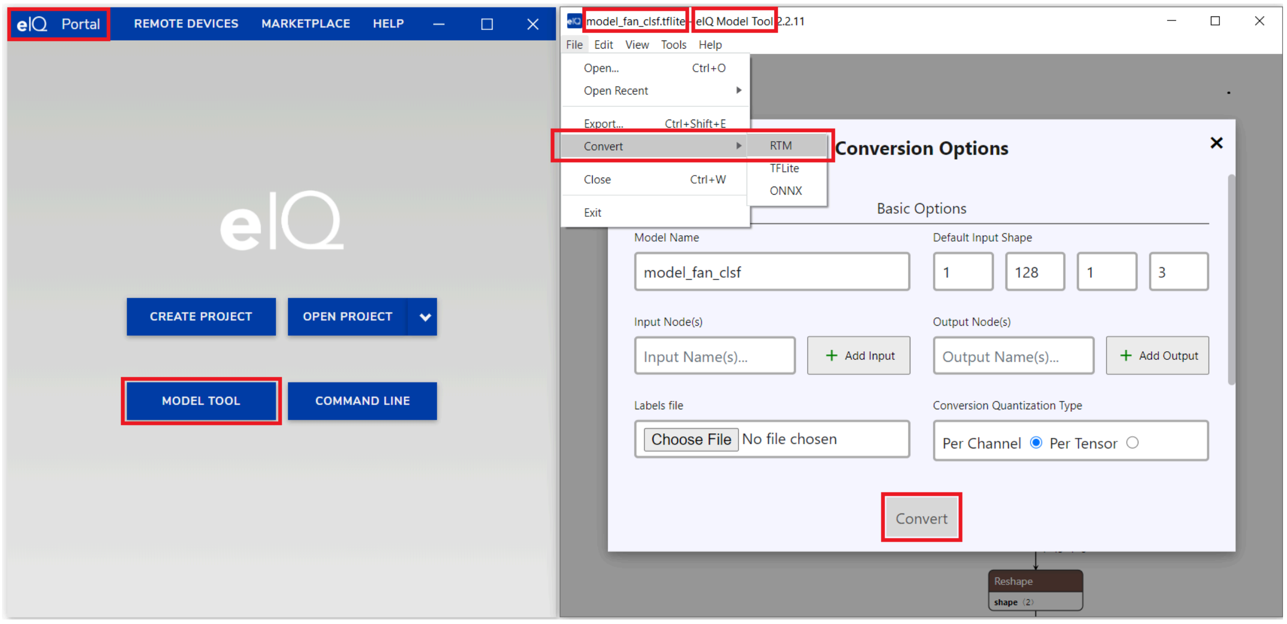
Figure 21. Model validation on host

4.4.2.5 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`), the converted TFLite model (`model.tflite`), and the TFLite-quantized model (`model_quant.tflite`). For quantization a hybrid posttraining 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 convert properly the model in the specific format. These steps rely on the eIQ toolkit (eIQ Portal, eIQ Model Tool, eIQ Glow). For more information, see Section 2.

The model tool from the eIQ portal software suite must be used for conversion in the RTM format (DeepViewRT), as shown in Figure 22.

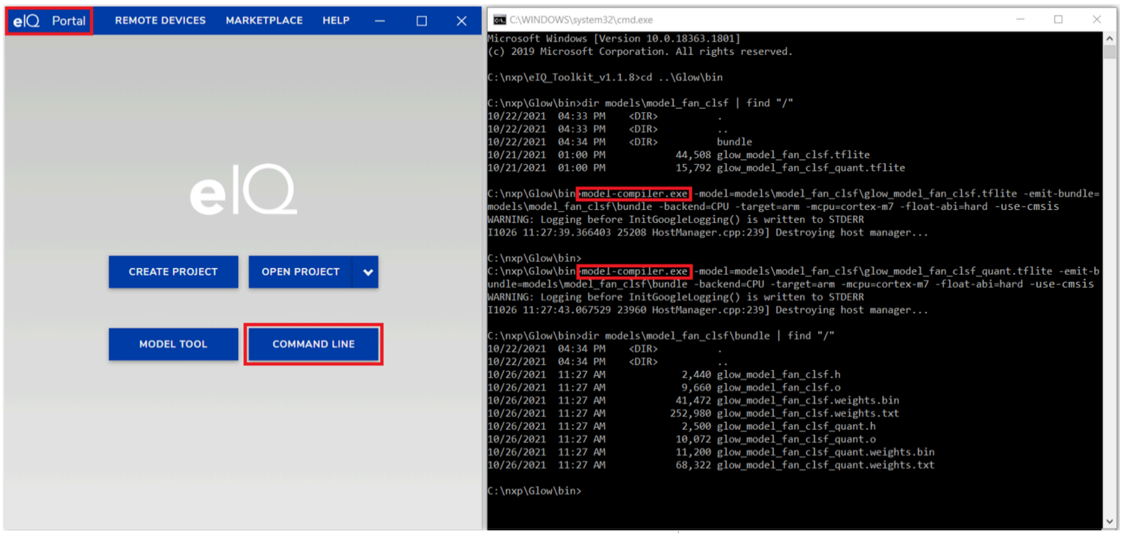


Model Name: model_fan_clsf
Default Input Shape: 1 | 128 | 1 | 3
Input Node(s): Input Name(s)...
Output Node(s): Output Name(s)...
Labels file: No file chosen
Conversion Quantization Type: [✓ Per Channel] Per Tensor

Figure 22. Model tool conversion for DeepViewRT

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, modify the `-target` and `-mcpu` parameters as required.

```
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
```



```
Microsoft Windows [Version 10.0.18363.1801]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\nxp\eIQ_Toolkit_v1.1.8>cd ..\Glow\bin

C:\nxp\Glow\bin>dir models\model_fan_clsf | find "/"
10/22/2021  04:33 PM    <DIR>          .
10/22/2021  04:33 PM    <DIR>          ..
10/22/2021  04:34 PM    <DIR>          bundle
10/21/2021  01:00 PM            44,508 glow_model_fan_clsf.tflite
10/21/2021  01:00 PM            15,792 glow_model_fan_clsf_quant.tflite

C:\nxp\Glow\bin>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
WARNING: Logging before InitGoogleLogging() is written to STDERR
I1026 11:27:39.366403 25208 HostManager.cpp:239] Destroying host manager...

C:\nxp\Glow\bin>
C:\nxp\Glow\bin>model-compiler.exe -model=models\model_fan_clsf\glow_model_fan_clsf_quant.tflite -emit-b
undle=models\model_fan_clsf\bundle -backend=CPU -target=arm -mcpu=cortex-m7 -float-abi=hard -use-cmsis
WARNING: Logging before InitGoogleLogging() is written to STDERR
I1026 11:27:43.067529 23960 HostManager.cpp:239] Destroying host manager...

C:\nxp\Glow\bin>dir models\model_fan_clsf\bundle | find "/"
10/22/2021  04:34 PM    <DIR>          .
10/22/2021  04:34 PM    <DIR>          ..
10/26/2021  11:27 AM             2,440 glow_model_fan_clsf.h
10/26/2021  11:27 AM             9,660 glow_model_fan_clsf.o
10/26/2021  11:27 AM            41,472 glow_model_fan_clsf.weights.bin
10/26/2021  11:27 AM           252,980 glow_model_fan_clsf.weights.txt
10/26/2021  11:27 AM             2,500 glow_model_fan_clsf_quant.h
10/26/2021  11:27 AM            10,072 glow_model_fan_clsf_quant.o
10/26/2021  11:27 AM            11,200 glow_model_fan_clsf_quant.weights.bin
10/26/2021  11:27 AM            68,322 glow_model_fan_clsf_quant.weights.txt

C:\nxp\Glow\bin>
```

Figure 23. Model conversion for Glow

4.4.3 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 application provided.
  Note: *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.
  Note: *Enable the sensor shield if the optional sensor toolbox (FRDM-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.
  Note: *The models ported by following the guidelines from Section 4.4.2.5 must be stored in this directory.*
- `mcu_app/source/models/model_selection.h`: must be configured and selects between quantized and not quantized versions and how to load the model (from Flash or from RAM).
  Note: *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 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 and selects the application runtime behavior.
  – Log externally the sensor data.

```
/* 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).

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

#if 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 is used for evaluation. Depending on which class is evaluated, the pointer which references the validation data must be configured in the `sensor_collect.c` file.

```
/* 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];
```

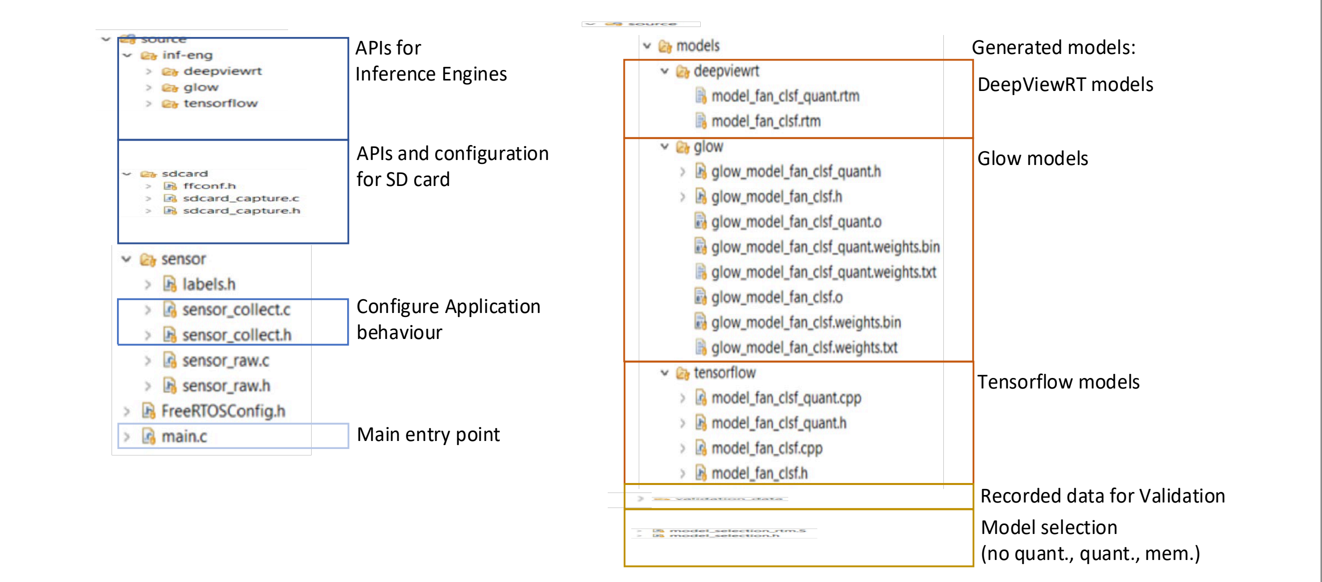


Figure 24. Source code structure

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%
```

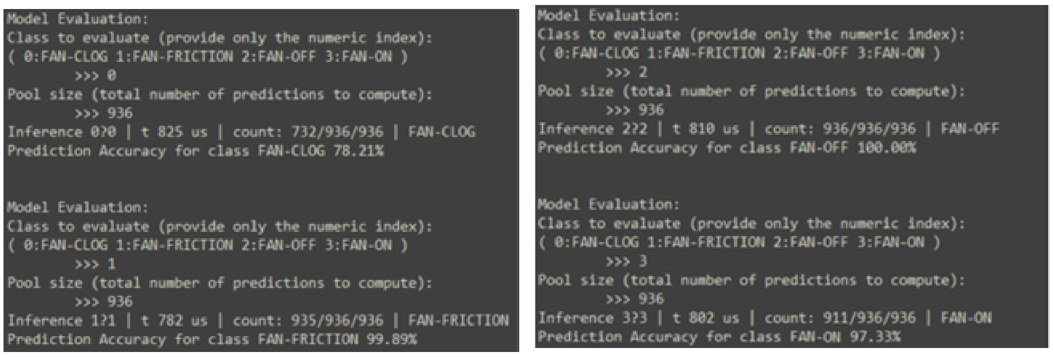
To compute the accuracy, divide the *number of correct predictions* to the *total number of predictions*. When the model is evaluated on the board, two parameters are provided.

1. `Target class`: specifies which class is evaluated and for which class the accuracy is 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 to 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, as shown 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 expected as the quantization is a lossy process that converts the model to smaller precision data.*



```
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%

Model Evaluation:
Class to evaluate (provide only the numeric index):
( 0:FAN-CLOG 1:FAN-FRICTION 2:FAN-OFF 3:FAN-ON )
        >>> 1
Pool size (total number of predictions to compute):
        >>> 936
Inference 1?1 | t 782 us | count: 935/936/936 | FAN-FRICTION
Prediction Accuracy for class FAN-FRICTION 99.89%
```

```
Model Evaluation:
Class to evaluate (provide only the numeric index):
( 0:FAN-CLOG 1:FAN-FRICTION 2:FAN-OFF 3:FAN-ON )
        >>> 2
Pool size (total number of predictions to compute):
        >>> 936
Inference 2?2 | t 810 us | count: 936/936/936 | FAN-OFF
Prediction Accuracy for class FAN-OFF 100.00%

Model Evaluation:
Class to evaluate (provide only the numeric index):
( 0:FAN-CLOG 1:FAN-FRICTION 2:FAN-OFF 3:FAN-ON )
        >>> 3
Pool size (total number of predictions to compute):
        >>> 936
Inference 3?3 | t 802 us | count: 911/936/936 | FAN-ON
Prediction Accuracy for class FAN-ON 97.33%
```

Figure 25. Model evaluation on the board using prerecorded validation data

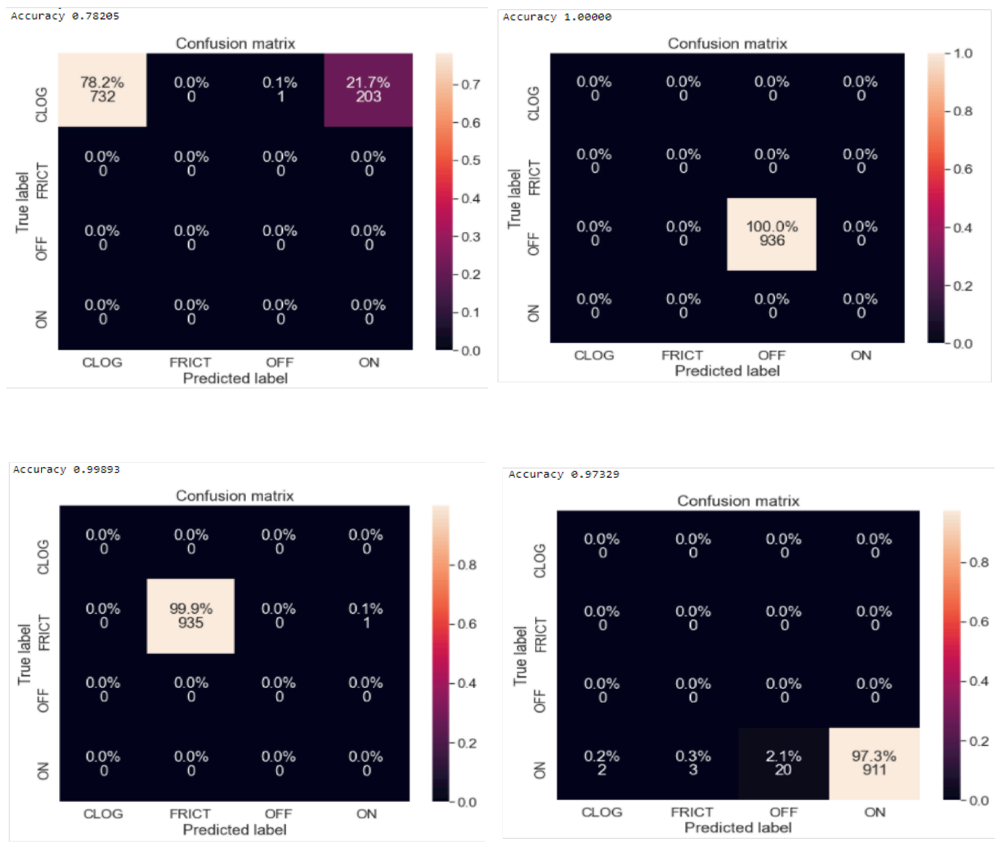


Figure 26. Model evaluation on the host machine using prerecorded validation data

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. To evaluate this duration, load 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 in the inference engines memory footprint, because the model is small, but this ratio would be less relevant for larger models.

Table 2. Model evaluation on the embedded board

		TFLite (no quant)	TFLite (quant)	DeepViewRT[1] (no quant)	DeepViewRT[1] (quant)	Glow (no quant)	Glow (quant)
InferenceTime (ms)							
@996 MHz	RAM	0.74	0.48	1.16	1.14	0.35	0.17
	Flash	1.46	0.55	1.77	1.31	0.98	0.23
@156 MHz	RAM	4.89	1.92	3.50	4.32	2.19	1.08
	Flash	5.31	1.93	3.80	4.44	2.42	1.09
Model size (KB)							
Flash/RAM (const)		43.5	15.4	44.3	15.4	40.5	10.6
RAM (var)		9.4	4.5	11.3	6.3	9.6	3.7
Code size (KB):							
Flash		56.3	60.9	109.2	109.2	10.9	10.4

[1] The evaluated software version for the DeepViewRT inference engine is released by the end of Q1-2022.

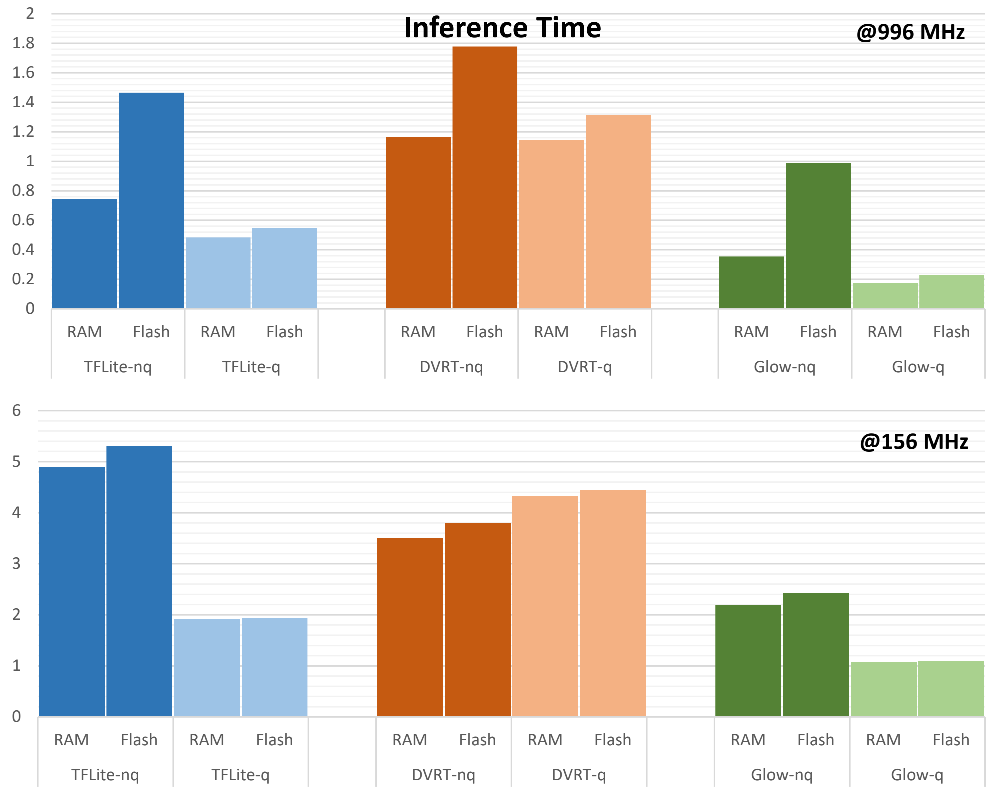


Figure 27. Model performance - inference time

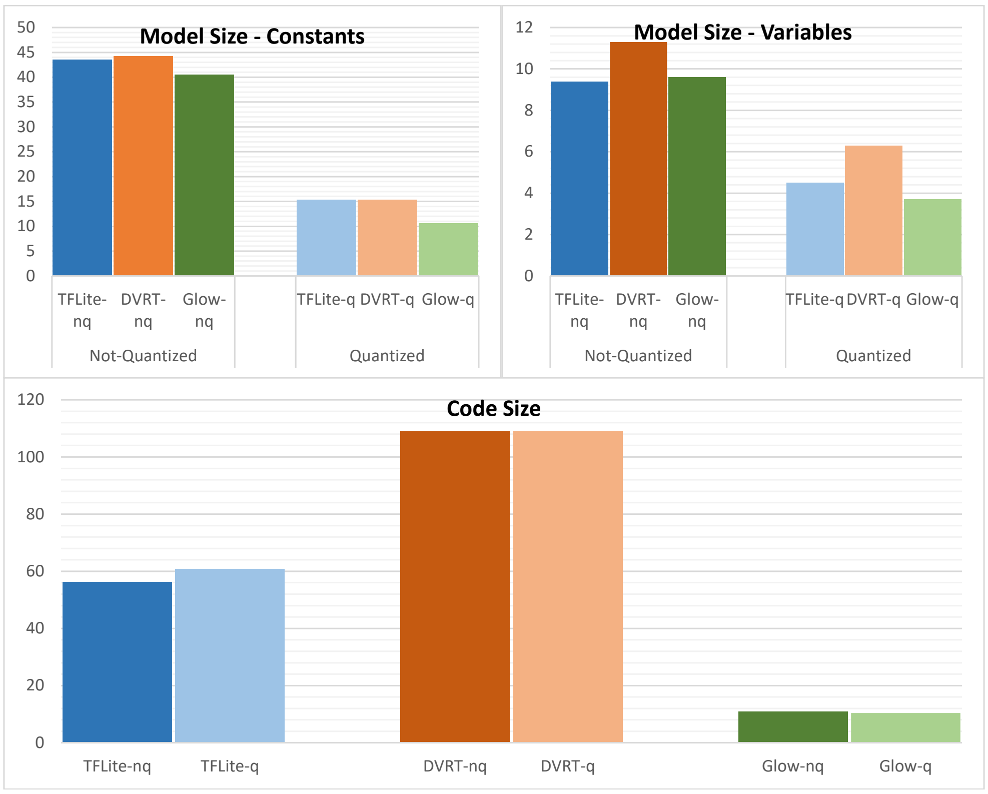


Figure 28. Memory usage

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

InferenceTime (ms) @150 MHz	TFLite (no quant)	TFLite (quant)	Glow (no quant)	Glow (quant)
MIMXRT1170-EVK (Arm Cortex-M7)	4.89	1.92	2.19	1.08
FRDM-K66F (Arm Cortex-M4)	8.87	4.18	4.24	5.47[1]
LPC55S69-EVK (Arm Cortex-M33)	9.95	5.77	4.16	3.88

[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.

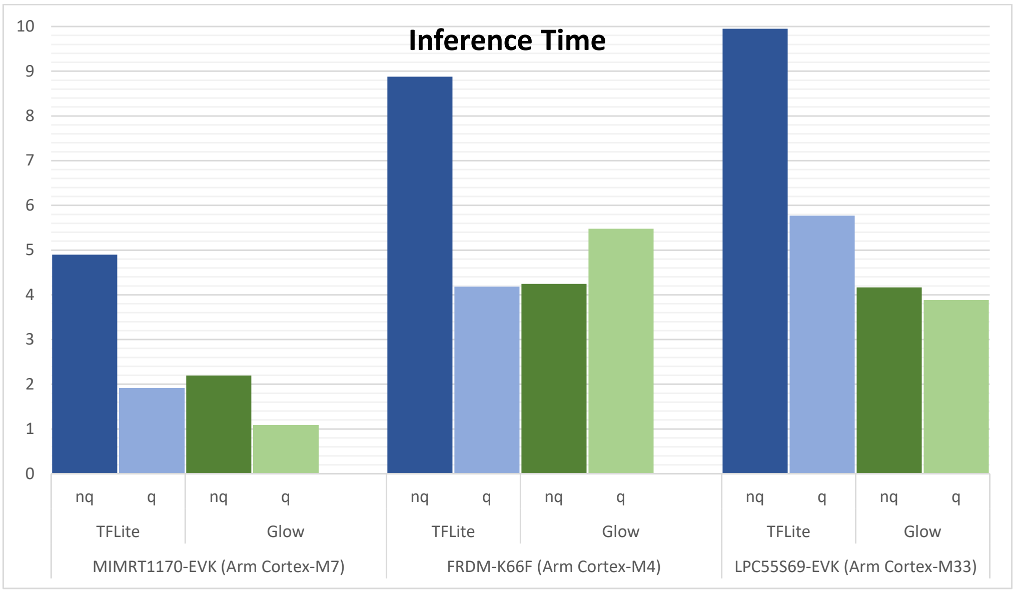


Figure 29. Inference-time evaluated on multiple embedded devices

4.5 Case Study for Application v2 with MPP

The case study for Application v2 with MPP is derived from Application v1, following the same steps with the differences listed below.

4.5.1 Sensor data collection

The sensor in Table 1 is replaced with FXLS8974CF, as listed below:

Sensor(s)	FXLS8974CF (3 axis sensor) 3-axis 12-bit accelerometer

To collect the dataset, enable the embedded application to log the sensor data on the SD card and provide the required configuration on the terminal.

```
/* SELECT data usage: 0 - run ML inference; 1 - run SD log data capturing */
#define SENSOR_COLLECT_LOG_EXT 1

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
```

4.5.2 Model development

The model development flow described in Application development workflow and Figure 3 is applied to the case study by running on the host machine with Jupyter notebook provided for Application v2 with MPP.

The following aspects are similar to Application v1 and detailed in Section 4.4.2.

- Importing the dataset
- Data analysis, processing and shaping
- Model validation on host machine

- Model porting for embedded device

4.5.2.1 Model definition, training, and evaluation

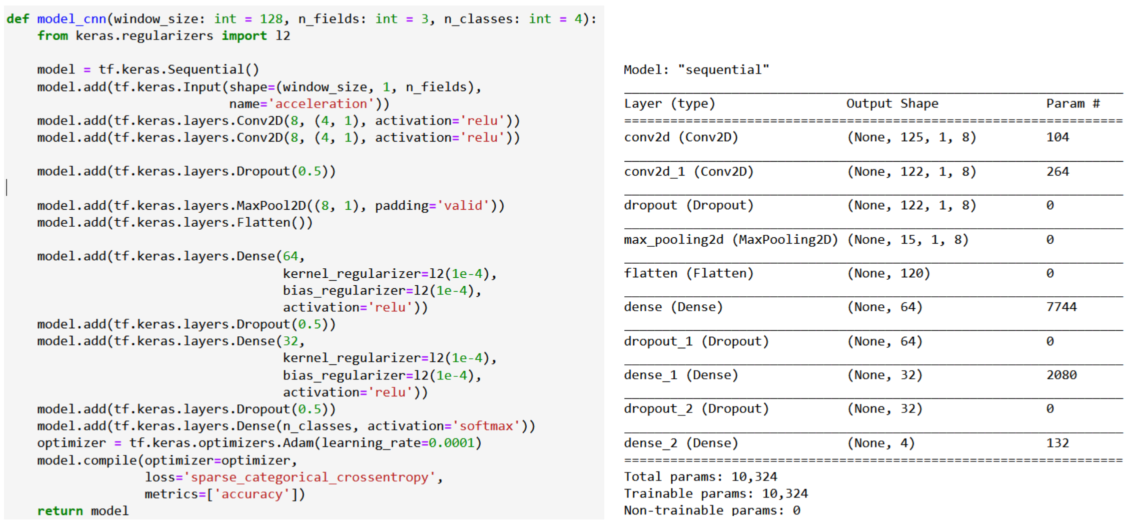
Application v2 provides four different model architectures: CNN, Light CNN, MLP, and Light ResNet with different performance for the following metrics: accuracy, inference latency, and memory usage.

The application developers trade off between higher accuracy, lower inference latency, and lower memory usage, depending on the application needs and the resources available. The most balanced model from the four is the Light CNN, which provides good results for latency and memory usage for a slight decrease in accuracy. The application developers can also use the models as seeds for future architectures.

Each model was trained on the training data and evaluated using the validation data. After training, each model was converted to TensorFlow Lite format using the TensorFlow Lite converter. Then, the TensorFlow Lite has been quantized to fit in low-memory devices.

4.5.2.2 CNN

The Convolutional Neural Network (CNN) overview and the CNN model architecture is identical with the one provided in Application v1.



```python
def model_cnn(window_size: int = 128, n_fields: int = 3, n_classes: int = 4):
    from keras.regularizers import l2

    model = tf.keras.Sequential()
    model.add(tf.keras.Input(shape=(window_size, 1, n_fields),
                             name='acceleration'))
    model.add(tf.keras.layers.Conv2D(8, (4, 1), activation='relu'))
    model.add(tf.keras.layers.Conv2D(8, (4, 1), activation='relu'))

    model.add(tf.keras.layers.Dropout(0.5))

    model.add(tf.keras.layers.MaxPool2D((8, 1), padding='valid'))
    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(64,
                                    kernel_regularizer=l2(1e-4),
                                    bias_regularizer=l2(1e-4),
                                    activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(32,
                                    kernel_regularizer=l2(1e-4),
                                    bias_regularizer=l2(1e-4),
                                    activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(n_classes, activation='softmax'))
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 125, 1, 8)         104
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 122, 1, 8)         264
_________________________________________________________________
dropout (Dropout)            (None, 122, 1, 8)         0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 1, 8)          0
_________________________________________________________________
flatten (Flatten)            (None, 120)               0
_________________________________________________________________
dense (Dense)                (None, 64)                7744
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 132
=================================================================
Total params: 10,324
Trainable params: 10,324
Non-trainable params: 0
```

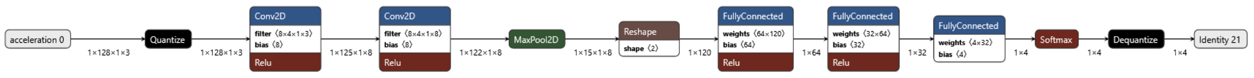


Figure 30. CNN model structure and implementation

4.5.2.3 LightCNN

Lighter CNN architecture is a simplified version of the CNN architecture that uses one Conv2D and two Dense layers, occupying less memory and having better latency with a slight decrease in accuracy.

```python
def model_lcnn(window_size: int = 128, n_fields: int = 3, n_classes: int = 4):
    model = tf.keras.Sequential()
    model.add(tf.keras.Input(shape=(window_size, 1, n_fields),
                             name='acceleration'))
    model.add(tf.keras.layers.Conv2D(8, (4, 1), activation='relu'))

    model.add(tf.keras.layers.Dropout(0.2))

    model.add(tf.keras.layers.MaxPool2D((8, 1), padding='valid'))
    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(16,
                                    kernel_regularizer=l2(1e-4),
                                    bias_regularizer=l2(1e-4),
                                    activation='relu'))

    model.add(tf.keras.layers.Dense(n_classes, activation='softmax'))
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 125, 1, 8)         104
_________________________________________________________________
dropout_7 (Dropout)          (None, 125, 1, 8)         0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 15, 1, 8)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 120)               0
_________________________________________________________________
dense_6 (Dense)              (None, 16)                1936
_________________________________________________________________
dense_7 (Dense)              (None, 4)                 68
=================================================================
Total params: 2,108
Trainable params: 2,108
Non-trainable params: 0
```

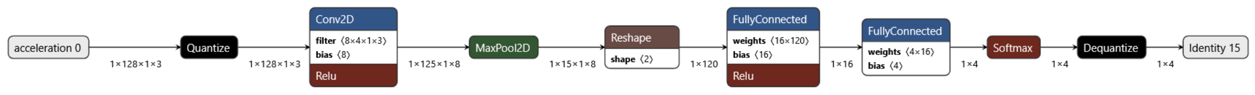


Figure 31. Light CNN model structure and implementation

4.5.2.4 MLP

Multi Layer Perceptron (MLP) is a fully connected type of Artificial Neural Network (ANN) that has input and output layers, and one or more hidden layers with many neurons connected. In this type of architecture, each layer feeds the next one with the result of their computation. This goes all the way through the hidden layers to the output layer. In this project, we used an MLP model of one hidden layer with 16 units. Figure 32 shows the MLP architecture.

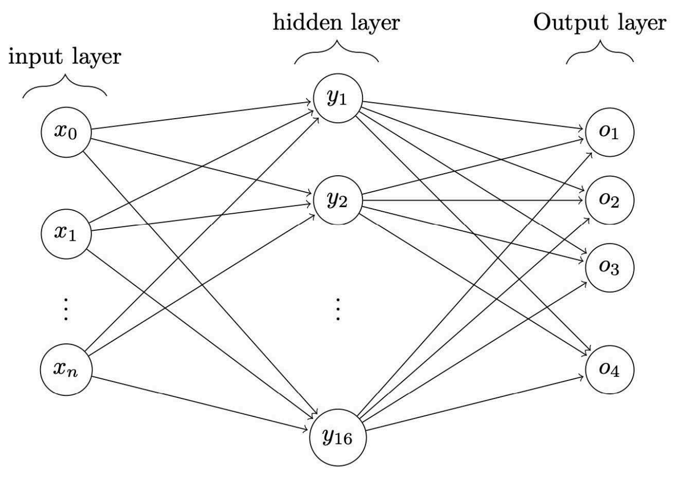
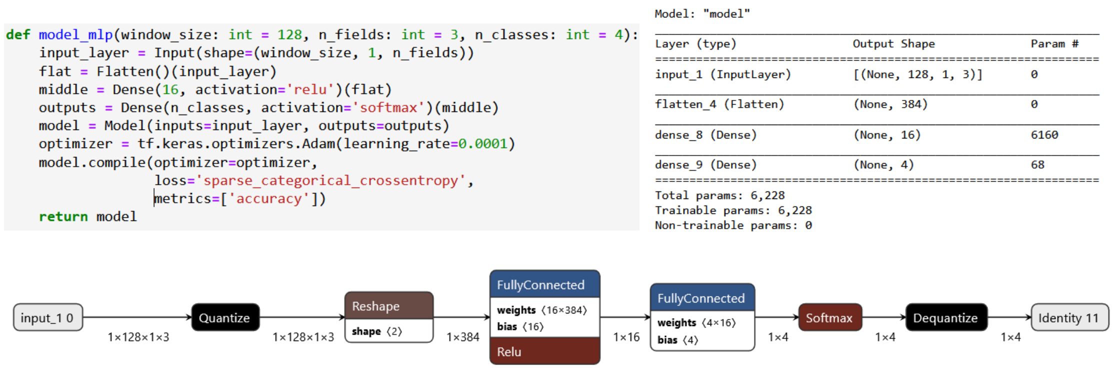


Figure 32. MLP architecture

The MLP architecture provided with the Application v2 is the smallest and fastest version of the four options with the downside of a lower accuracy.



```python
def model_mlp(window_size: int = 128, n_fields: int = 3, n_classes: int = 4):
    input_layer = Input(shape=(window_size, 1, n_fields))
    flat = Flatten()(input_layer)
    middle = Dense(16, activation='relu')(flat)
    outputs = Dense(n_classes, activation='softmax')(middle)
    model = Model(inputs=input_layer, outputs=outputs)
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
Model: "model"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_1 (InputLayer)        [(None, 128, 1, 3)]       0
_________________________________________________________________
flatten_4 (Flatten)         (None, 384)               0
_________________________________________________________________
dense_8 (Dense)             (None, 16)                6160
_________________________________________________________________
dense_9 (Dense)             (None, 4)                 68
=================================================================
Total params: 6,228
Trainable params: 6,228
Non-trainable params: 0
```

Figure 33. MLP model structure and implementation

4.5.2.5 Light ResNet

Residual Network (ResNet) is a CNN architecture used mostly for computer vision applications. It has been shown that the classical CNN model can result in the vanishing/exploding gradient issue when the number of layers grow. The vanishing gradient phenomenon happens when the gradient is so small that the weight values of the model remain unchanged during their update. ResNet architecture is designed to support hundreds or thousands of convolutional layers without vanishing gradient issue. This new architecture introduces a concept called Residual Blocks. In this architecture, the activations of a layer are connected to further layers by skipping some layers in between. This forms a residual block (see Figure 34). To make the resnets model, stack these residual blocks.

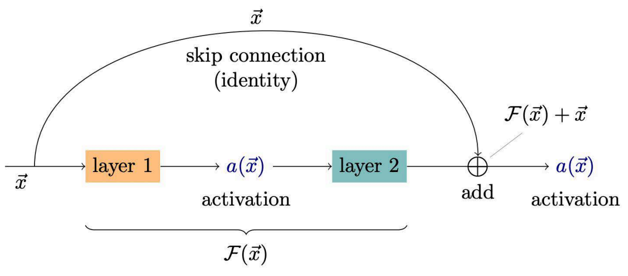
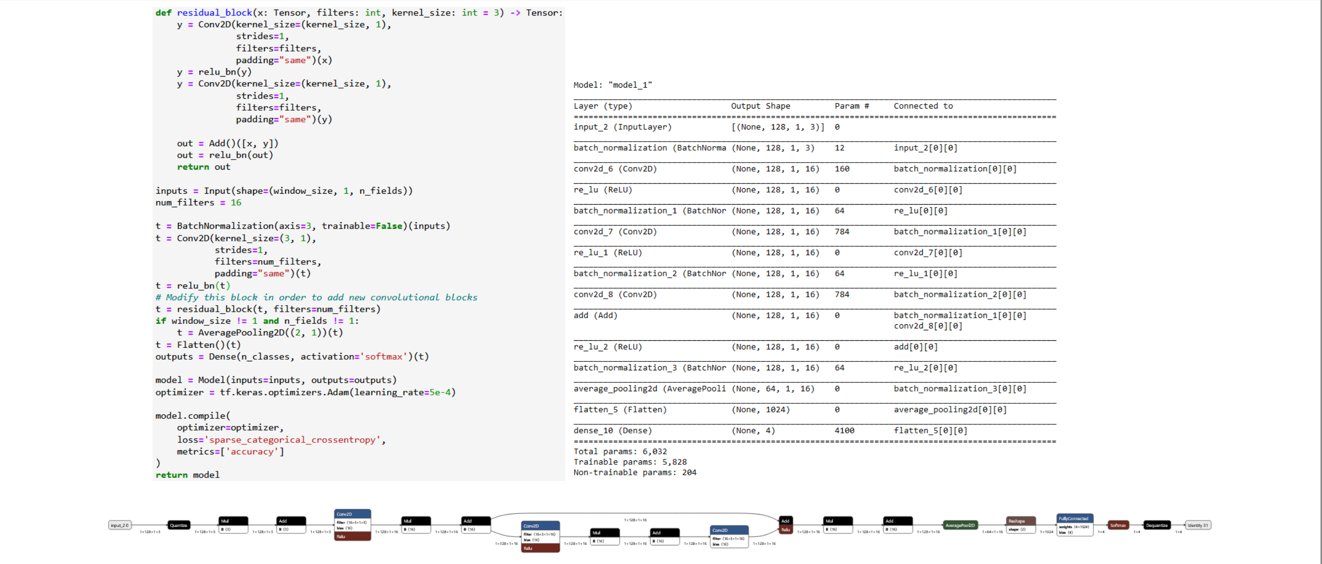


Figure 34. Residual block of Restnet architecture

The Light ResNet architecture provided with the Application v2 is a small version with one residual block that provides high accuracy with a downside on memory usage and inference latency:



```
def residual_block(x: Tensor, filters: int, kernel_size: int = 3) -> Tensor:
    y = Conv2D(kernel_size=(kernel_size, 1),
               strides=1,
               filters=filters,
               padding="same")(x)
    y = relu_bn(y)
    y = Conv2D(kernel_size=(kernel_size, 1),
               strides=1,
               filters=filters,
               padding="same")(y)

    out = Add()([x, y])
    out = relu_bn(out)
    return out

inputs = Input(shape=(window_size, 1, n_fields))
num_filters = 16

t = BatchNormalization(axis=3, trainable=False)(inputs)
t = Conv2D(kernel_size=(3, 1),
           strides=1,
           filters=num_filters,
           padding="same")(t)
t = relu_bn(t)
# Modify this block in order to add new convolutional blocks
t = residual_block(t, filters=num_filters)
if window_size != 1 and n_fields != 1:
    t = AveragePooling2D((2, 1))(t)
t = Flatten()(t)
outputs = Dense(n_classes, activation='softmax')(t)

model = Model(inputs=inputs, outputs=outputs)
optimizer = tf.keras.optimizers.Adam(learning_rate=5e-4)

model.compile(
    optimizer=optimizer,
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
return model
```

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_2 (InputLayer)            [(None, 128, 1, 3)]  0

__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 128, 1, 3)    12          input_2[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 128, 1, 16)   160         batch_normalization[0][0]
__________________________________________________________________________________________________
re_lu (ReLU)                    (None, 128, 1, 16)   0           conv2d_6[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 128, 1, 16)   64          re_lu[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 128, 1, 16)   784         batch_normalization_1[0][0]
__________________________________________________________________________________________________
re_lu_1 (ReLU)                  (None, 128, 1, 16)   0           conv2d_7[0][0]
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 128, 1, 16)   64          re_lu_1[0][0]
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 128, 1, 16)   784         batch_normalization_2[0][0]
__________________________________________________________________________________________________
add (Add)                       (None, 128, 1, 16)   0           batch_normalization_1[0][0]
                                                                 conv2d_8[0][0]
__________________________________________________________________________________________________
re_lu_2 (ReLU)                  (None, 128, 1, 16)   0           add[0][0]
__________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 128, 1, 16)   64          re_lu_2[0][0]
__________________________________________________________________________________________________
average_pooling2d (AveragePooli (None, 64, 1, 16)    0           batch_normalization_3[0][0]
__________________________________________________________________________________________________
flatten_5 (Flatten)             (None, 1024)         0           average_pooling2d[0][0]
__________________________________________________________________________________________________
dense_10 (Dense)                (None, 4)            4100        flatten_5[0][0]
==================================================================================================
Total params: 6,032
Trainable params: 5,828
Non-trainable params: 204
```

Figure 35. Light ResNet model structure and implementation

4.5.3 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 36 and includes:

- `doc/readme.md`: short documentation for the application provided.
  Note: *Refer to this document or the laboratory guide for guidelines on hardware requirements, board settings, and running the demo.*
- `source/main.c`: contains the main source of the application that uses the MPP library with exploratory Time Series support.

```
MPP API
=======

The MPP (Multimedia Processing Pipleine) library with exploratory time series
 support is built on top of MPP eIQ release https://github.com/nxp-mcuxpresso/
mcux-sdk-middleware-eiq/tree/MCUX_2.13.0/mpp

The exploratory library located in eiq/mpp folder extends the pipeline API with
 the following functions documented in mpp_api.h:
    int mpp_api_init(uint32_t input_dequeue_frequency)
    int mpp_accelerometer_add(...)
    int mpp_static_accelerometer_add(...)
    int mpp_sd_add(...)

The library provides source code for drivers in eiq/mpp/hal folder.
```

- `sensors/`: contains the low-level driver for fxls8974 sensor.
- `ource/sdcard/`: contains a low-level driver for data capturing on SD card.
- `source/models/glow` and `source/models/tensorflow`: contain the C source code of the models.

- `source/inf-eng/glow` and `source/models/tensorflow`: contain the integration of the models.
- `mcu_app/source/app_config.h`: must be configured to select the application runtime behavior.
  - The application supports two configurations of the MPP pipeline:

```
Fxls8974 accelerometer (with real time or static data) and
inference components to detect fan system state using Neural Networks

+--------------------------+      +----------------+
|                          |      |                |
|                          |      |                |
|      Accelerometer       | -->  |   Inference    |
| Fxls8974 or static data  |      |                |
|                          |      |                |
+--------------------------+      +----------------+
```

```
Fxls8974 accelerometer and SD card component to collect data for training
 Neural Networks models

+--------------------------+      +----------------+
|                          |      |                |
|                          |      |                |
|      Accelerometer       | -->  |    SD card     |
| Fxls8974 or static data  |      |                |
|                          |      |                |
+--------------------------+      +----------------+
```

  - Run the selected inference engine and compute predictions either on real time data, or on prerecorded validation data (the *SOURCE_STATIC_ACCEL_DATA* flag toggles between real time or offline validation).

```
/* SELECT data source: 0 - live accelerometer sensor data; 1 - static
 validation data */
#define SOURCE_STATIC_ACCEL_DATA 0

/* SELECT data usage: 0 - run ML inference; 1 - run SD log data capturing */
#define SENSOR_COLLECT_LOG_EXT 0

/* Available ML inference engines */
#define INFERENCE_ENGINE_TFLM 1
#define INFERENCE_ENGINE_GLOW 0

/* SELECT the ML inference engine */
#define INFERENCE_ENGINE INFERENCE_ENGINE_GLOW
```

  - Select the desired NN model and quantization for inference.

```
/* Available ML NN models for inference */
#define MODEL_CNN       1
#define MODEL_LCNN      2
#define MODEL_MLP       3
#define MODEL_LRESNET   4

/* SELECT ML NN model for inference */
#define SELECTED_MODEL MODEL_LCNN

/* SELECT quantization: 0 - no quantization; 1 - use quantized models */
#define MODEL_QUANTIZED   1
```

– When offline static validation is configured the prerecorded data from the *source/models/validation_data/* directory is used for evaluation. Configure the desired dataset class for validation.

```
/* SELECT data source: 0 - live accelerometer sensor data; 1 - static
 validation data */
#define SOURCE_STATIC_ACCEL_DATA 1

/* Available pre-recorded validation datasets */
#define VDSET_CLOG      0
#define VDSET_FRICTION  1
#define VDSET_ON        2
#define VDSET_OFF       3

#if (SOURCE_STATIC_ACCEL_DATA == 1)
/* SELECT the static dataset for validation */
#define STATIC_ACCEL_VDSET VDSET_ON
#endif
```

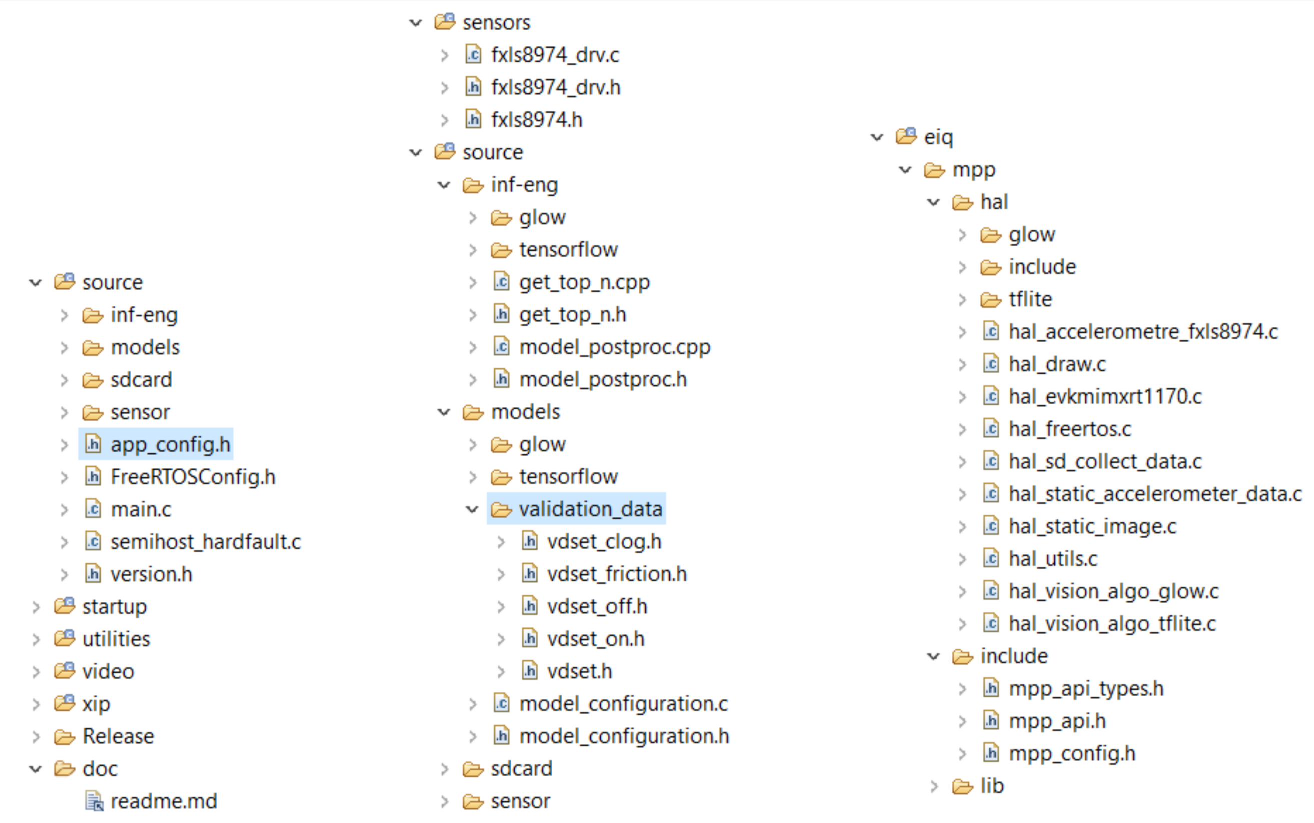
– Collect sensor data on SD card.

```
/* SELECT data usage: 0 - run ML inference; 1 - run SD log data capturing */
#define SENSOR_COLLECT_LOG_EXT 1
```

– To update the dataset after data collection on SD card, run the Jupiter Notebook to generate the corresponding source files and update them in the *source/models/validation_data/* directory.

– When updating the dataset, update the mean and standard deviation of the dataset provided by the Jupiter notebook in *source/models/model_configuration.c*.

```
/* Mean and standard deviation of the dataset used for NN models training */
float model_mean[] = {-10.423518161780331, 0.3063457550753402,
 1044.4877729336488};
float model_std[] = {262.3036980857252, 239.52283182578253,
 268.3708749240864};
```



```
source
  inf-eng
  models
  sdcard
  sensor
  app_config.h
  FreeRTOSConfig.h
  main.c
  semihost_hardfault.c
  version.h
startup
utilities
video
xip
Release
doc
  readme.md
```

```
sensors
  fxls8974_drv.c
  fxls8974_drv.h
  fxls8974.h
source
  inf-eng
    glow
    tensorflow
    get_top_n.cpp
    get_top_n.h
    model_postproc.cpp
    model_postproc.h
  models
    glow
    tensorflow
    validation_data
      vdset_clog.h
      vdset_friction.h
      vdset_off.h
      vdset_on.h
      vdset.h
    model_configuration.c
    model_configuration.h
  sdcard
  sensor
```

```
eiq
  mpp
    hal
      glow
      include
      tflite
      hal_accelerometre_fxls8974.c
      hal_draw.c
      hal_evkmimxrt1170.c
      hal_freertos.c
      hal_sd_collect_data.c
      hal_static_accelerometer_data.c
      hal_static_image.c
      hal_utils.c
      hal_vision_algo_glow.c
      hal_vision_algo_tflite.c
    include
      mpp_api_types.h
      mpp_api.h
      mpp_config.h
    lib
```

Figure 36. Source code structure

Below is the comparative evaluation of the four quantized models running on LPC55S69 and i.MX RT 1170 using Glow and TensorFlow inference engines.

- CNN provides the best accuracy with higher latency and memory usage.
- Light CNN is the most balanced model providing good results for latency and memory usage for a slight decrease in accuracy. This is the recommended option if there are no special needs for the application.
- MLP is the smallest and fastest version of the four options with the downside of a lower accuracy.
- Light ResNet offers higher accuracy with a downside in memory usage and latency.

Table 4. Models evaluation on the embedded boards

LPC55S69 - Glow						
Model - Metric	**Accuracy (%)**	**Duration (ms)**	**Total Memory (KB)**	**Model RAM (KB)**	**Model Flush (KB)**	**Code Flash (KB)**
CNN Quantized	98.98 %	4	25.52	3.71	10.62	11.19
Light CNN Quantized	98.07 %	2	17.05	3.71	2.31	11.03
MLP Quantized	94.69 %	0	16.64	2.03	10.75	3.86
Light ResNet Quantized	98.93 %	10	31.20	7.78	6.25	17.17
LPC55S69 - TF						
Model - Metric	**Accuracy (%)**	**Duration (ms)**	**Total Memory (KB)**	**Model RAM (KB)**	**Model Flush (KB)**	**Code Flash (KB)**
CNN Quantized	98.98 %	6	77.17	4.15	14.92	58.1
Light CNN Quantized	98.07 %	3	67.14	3.62	5.42	58.1
MLP Quantized	94.69 %	0	52.69	3.17	8.31	41.21
Light ResNet Quantized	98.93 %	39	106.35	9.51	13.12	83.72
RT1170 - TF						
Model - Metric	**Accuracy (%)**	**Duration (ms)**	**Total Memory (KB)**	**Model RAM (KB)**	**Model Flush (KB)**	**Code Flash (KB)**
CNN Quantized	98.98 %	0.35	76.60	4.19	14.92	57.49
Light CNN Quantized	98.07 %	0.33	66.53	3.66	5.38	57.49
MLP Quantized	94.69 %	0.15	56.62	3.21	8.28	45.13
Light ResNet Quantized	98.93 %	2.01	100.27	9.55	13.03	77.69

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 using the NXP 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 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

7 Revision history

Table 5 summarizes the revisions to this document.

Table 5. Revision history

Revision number	Release date	Description
2	27 September 2023	• Updated Section 4.1 • Updated Section 4.2 • Updated Section 4.3 • Updated Section 4.4 • Updated Section 4.5 • Documented the enhanced version providing the eIQ MCU Multimedia Processing Pipeline (MPP) library with exploratory support for Time Series data, new FXLS8974CF accelerometer sensor and four new NN models with different performance metrics
1	25 April 2022	Added inference-time benchmarks for MIMXRT1170-EVK (CM7), FRDM-K66F (CM4), and LPC55S69-EVK (CM33)
0	03 February 2022	Initial release

8 Legal information

8.1 Definitions

8.2 Disclaimers

8.3 Trademarks

Contents

Date of release: 27 September 2023
Document identifier: AN13562