AN13438

Inferencing Deep Learning on Cortex M0/M0+ with the eIQ^{TM} Glow Inference Engine

Rev. 0 — 18 November 2021

Application Note

Contents

1 Introduction

This application note focuses on enabling deep learning on NXP microcontrollers with Arm® Cortex®-M0/M0+ cores. Machine learning and deep learning models are powerful algorithms associated with high demand of processing power and often known as memory hungry applications. In this application note, we demonstrate the possibility to embed a deep learning model on different NXP microcontroller families with Arm Cortex-M0 and M0+ as main core.

For that purpose, we designed a simple convolutional neural network (CNN) as a deep learning model example, but similar porting approach could be followed for other types of DL models as well. As an example use case, we selected solving the handwritten digit classification task using the MNIST data set, but the same flow could be replicated for any other type of vision, audio or anomaly detection use cases. This document also clarifies the main conditions to achieve a model running on a small specific embedded device.

This application development is mainly leveraging the NXP's SDK and the eIQ^{TM} technology.

2 eIQ^{TM} and Glow introduction

The eIQ™ is an NXP® machine learning (ML) software development environment that enables the use of ML algorithms on NXP EdgeVerse™ microcontrollers and microprocessors, including i.MX RT crossover MCUs, and i.MX family application processors. This application note shows how to extend it to other microcontroller families, such as Kinetis and LPC.

eIQ ML software includes an ML workflow tool called eIQ Toolkit, along with inference engines, neural network compilers and optimized libraries. This software leverages open-source and proprietary technologies and is fully integrated into NXP MCUXpresso SDK and Yocto development environments, allowing to develop complete system-level applications with ease. Figure 1 depicts the different inference engines enabled on different processing cores correspondingly.

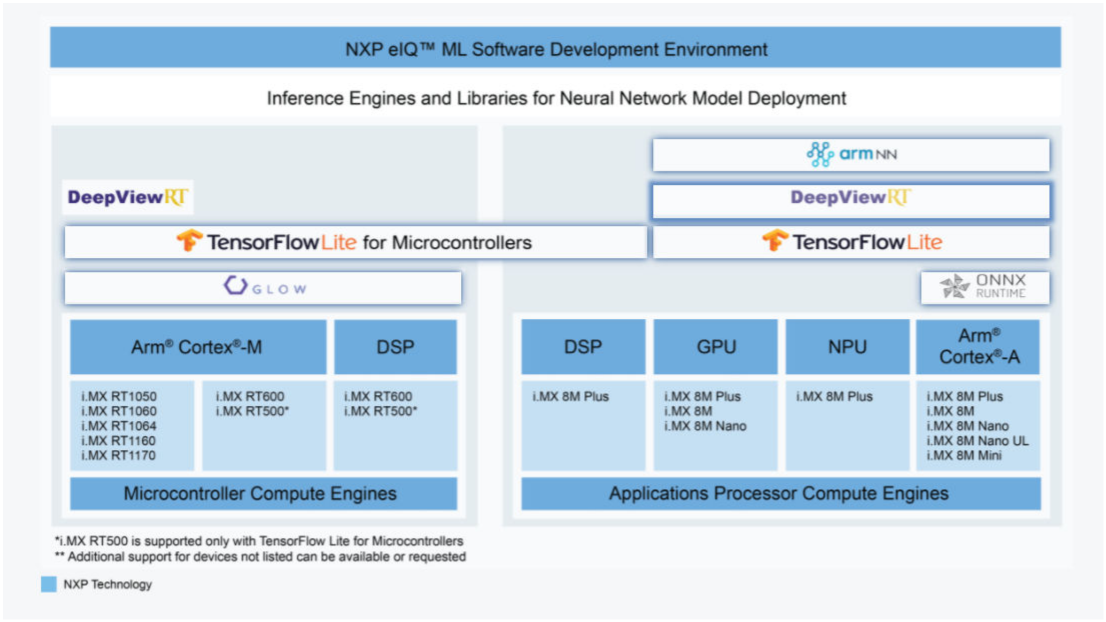

NXP eIQ™ ML Software Development Environment
Inference Engines and Libraries for Neural Network Model Deployment
arm NN
DeepViewRT
DeepViewRT
TensorFlow Lite for Microcontrollers
TensorFlow Lite
GLOW
ONNX RUNTIME
Arm® Cortex®-M
DSP
DSP
GPU
NPU
Arm® Cortex®-A
i.MX RT1050
i.MX RT1060
i.MX RT1064
i.MX RT1160
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

On the left side of the figure, we enumerate different inference engines included inside the eIQ component of the software development kit for microcontrollers; DeepViewRT™, TensorFlow™ Lite / TensorFlow™ Lite Micro, and Glow. This application note leverages glow inference engine to provide additional support for more devices not listed on the latest figure.

Glow machine learning compiler enables ahead-of-time compilation; it converts the neural networks into object files, then the user converts these into a binary image for increased performance and smaller memory footprint as compared to other runtime inference engines. Since Glow enables this conversion to target different Arm Cortex-M cores, this application note demonstrates how to leverage the eIQ glow to embed deep learning models on Arm Cortex-M0+ NXP microcontrollers.

Moreover, the two mentioned advantages of the Glow compiler (high performance and the small memory footprint) are very useful when the target is a low end microcontroller with limited core frequency and limited embedded memory, which makes glow a very suitable inference engine for such use case.

3 Requirements for running deep learning on Arm Cortex-M0/M0+

As depicted in Figure 2, the ML development steps consist of first defining the model architecture then iteratively train and optimize the model to achieve the required accuracy based on the collected data.

This is typically done on the PC or on the cloud using one of deep learning frameworks. Once the model is validated, the eIQ glow documentation guides the user through required steps to inference the model on the targeted device.

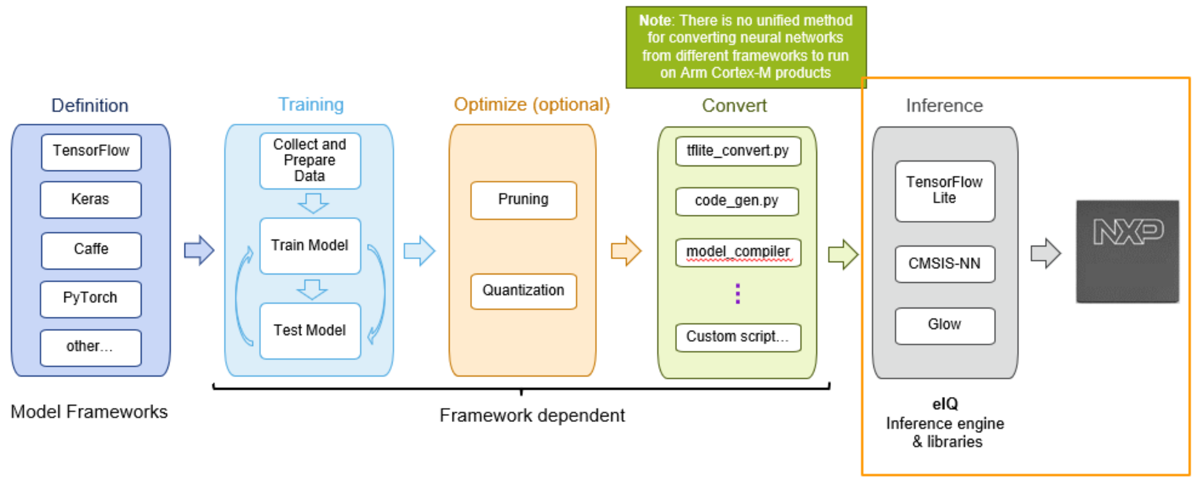


Figure 2. Machine learning process

For the Arm Cortex-M0/M0+ based targets, one important limitation to consider is the absence of the floating point unit (FPU) which is available for Arm Cortex-M33 and Arm Cortex-M7 (see Figure 3). The main consequence of this limitation is that the deep learning model should be fully quantized in order to be inferenced on an M0/M0+ core.

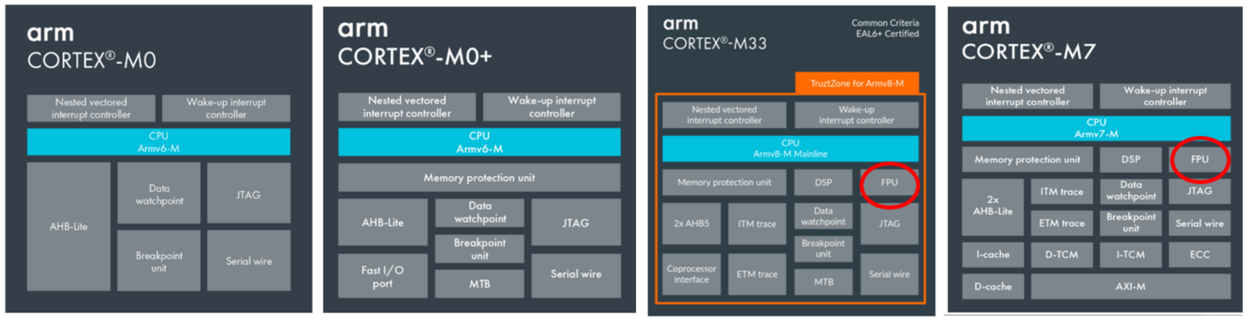


Figure 3. High-level architecture of different Cortex-M from Arm

During training of the model, the user can specify the data type of the model parameters. Usually model parameters as well as processing operations use floating point. Nevertheless it is possible to directly train a model in fixed points, such as int8 or int16, and this is often referred as quantization aware training. In the later case, the glow tool provides possibility to convert quantized model (for example, TFLite format) into glow conformant object file. In the case where the model is trained in floating point, glow provide all required tools to convert it to quantized glow model with different quantization options (see Workflow for porting the DL model on microcontroller). In both cases, the main challenge of the model designer is to ensure that the overall model accuracy, impacted by the quantization, is still satisfying the accuracy requirement.

The second requirement is about the embedded device memory (RAM and Flash) being large enough to fit the memory footprint required by the model processing. The total model size (model parameters and model instructions) has to be small enough for the given available embedded flash of the device. Same applies for the embedded RAM, since model processing requires internal buffers for saving intermediate operations output parsed from one layer to the next layer of the model. The bundle generated by glow during model conversion provides means to quantify the model memory footprint and this is deeply explained in the application note AN13001.

Another requirement that should be considered especially for real-time use cases is inference time; the total time required by the core to run the model inference. Since low end devices are characterized by relatively low performance, the inference time is expected to be high. Nevertheless for a lot of use cases with low complexity, a simple neural network model with few layers can very accurately solve the task even quantized. Besides, for some use cases like predictive maintenance, it is just needed to process the anomaly detection algorithm occasionally with no real-time requirement.

Results and performance comparison shows how this varies according to the core performance/frequency. Workflow for porting the DL model on microcontroller demonstrates how it is possible to design a deep learning model solving the handwritten digit classification task (based on MNIST data set) while satisfying all requirements mentioned in this section.

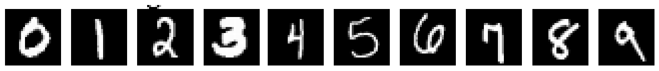
4 ML/DL Development: Design of a simple CNN solving handwritten classification task with MNIST data set

Since this application note aims at deploying machine learning models on Arm Cortex M0/M0+ microcontrollers, a very classic machine learning task is used as an example; MNIST handwritten digit classification. This is the machine learning “hello world” for images.

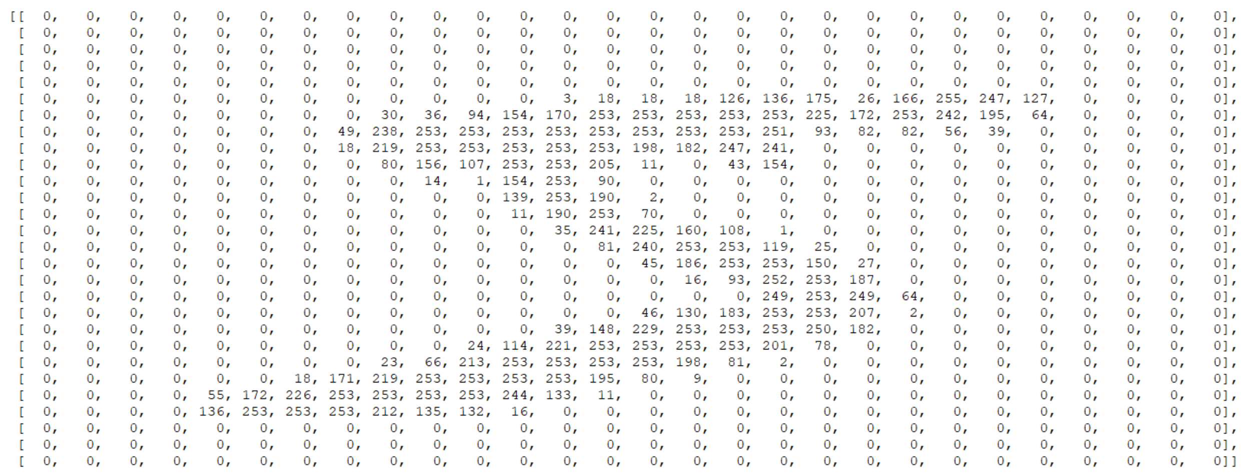
MNIST stands for Modified National Institute of Standards and Technology. The data set is made of 60000 labeled images for training and 10000 for validation and testing. The images are 28x28 pixels grayscale images of handwritten digits from 0 to 9. The task is to classify each image into the right category from 0 to 9. The original images were bitmap images. Each pixel was coded on 1 bit, black or white. Those images were converted to 28x28 format and an anti-aliasing filter was applied. The filter allows to smooth the boundaries between black and white pixels and the pixels are now coded on 8 bits. The “modified” word in the name of the data set comes from a new way to organize the training and testing sets. Before, the training set was not collected the same way as the testing set. They have now been shuffled.

Here are examples of pictures in the training set.

The testing set looks similar.



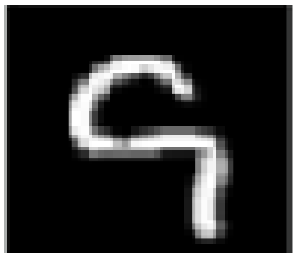
Images displayed on a screen usually have 3 dimensions; RGB for Red, Green, Blue. A 28x28 pixel RGB image is represented by a tensor of shape 28x28x3. This data set is in grayscale format so there is just one color channel. The shape of the image tensor is: 28x28x1. The images are coded on 8 bits, the values will be between 0 and 255. The first image of the data set is a 5. It is coded as the following matrix:



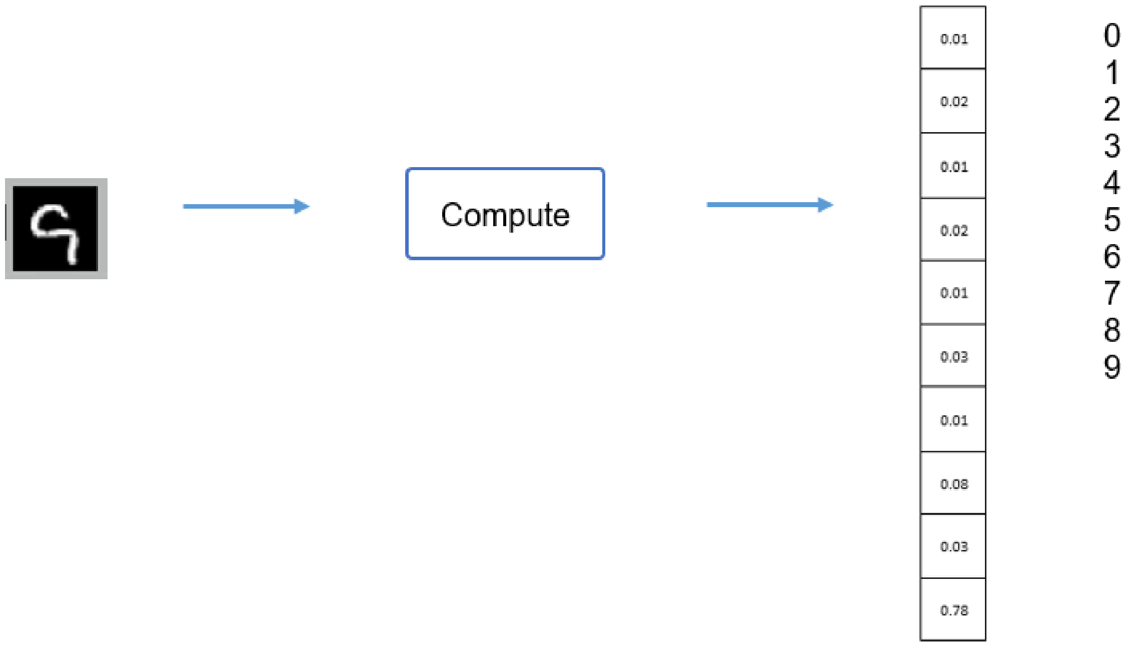
```
[[   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,  18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170, 253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253, 253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253, 253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253, 205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,  90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253, 190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190, 253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35, 241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39, 148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221, 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253, 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253, 195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,  11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
 [   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0]]
```

It is a matrix of dimension 28x28 filled with numbers between 0 and 255. A low value is a black pixel and a high value is a white pixel.

Our goal is to predict what number is on the picture. Here is another example from the training data set. It is a 9 in image format.



Our goal is to return a vector of size 10 containing the predictions from 0 to 9 like in the following figure. The highest value in the prediction vector corresponds to the predicted class. For this example, the last row of the vector is the biggest.



Let us have a look at how to build such a model. The model is trained using Python 3.8.9 and Jupyter Notebook IDE. Here are the different packages used.

- TensorFlow 2.0.0
- Numpy 1.20.2
- Matplotlib 3.4.2

TensorFlow is a machine learning framework written by Google. Numpy is a mathematical library. Matplotlib is a library used to plot graphics. The libraries can be imported with these lines.

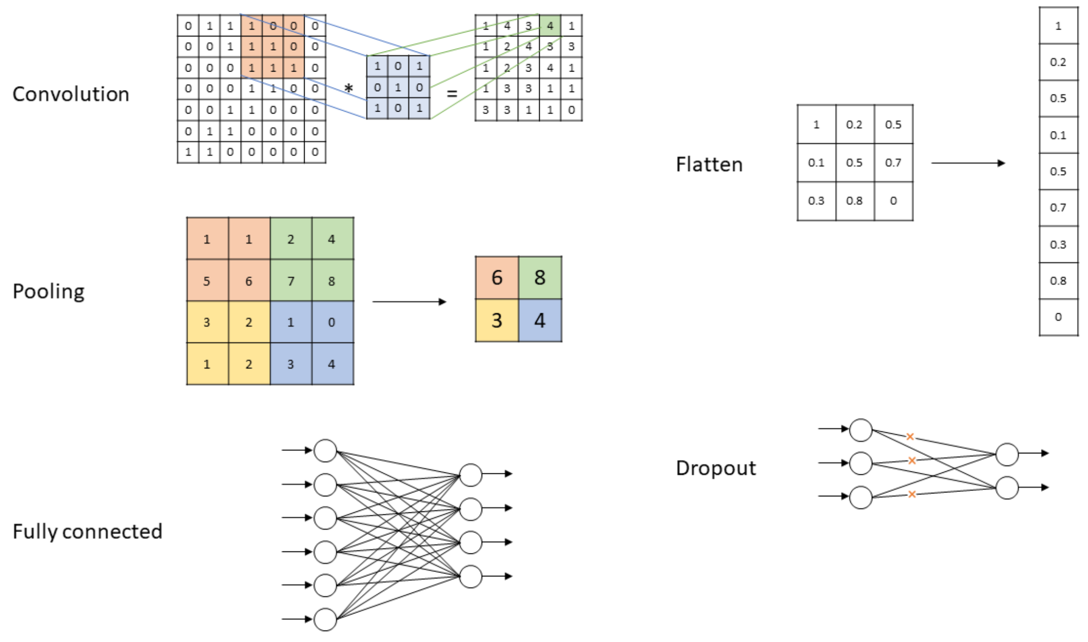
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

MNIST data set is already available inside TensorFlow. In order to get the dimension that TensorFlow expects for images, the data is loaded with these lines.

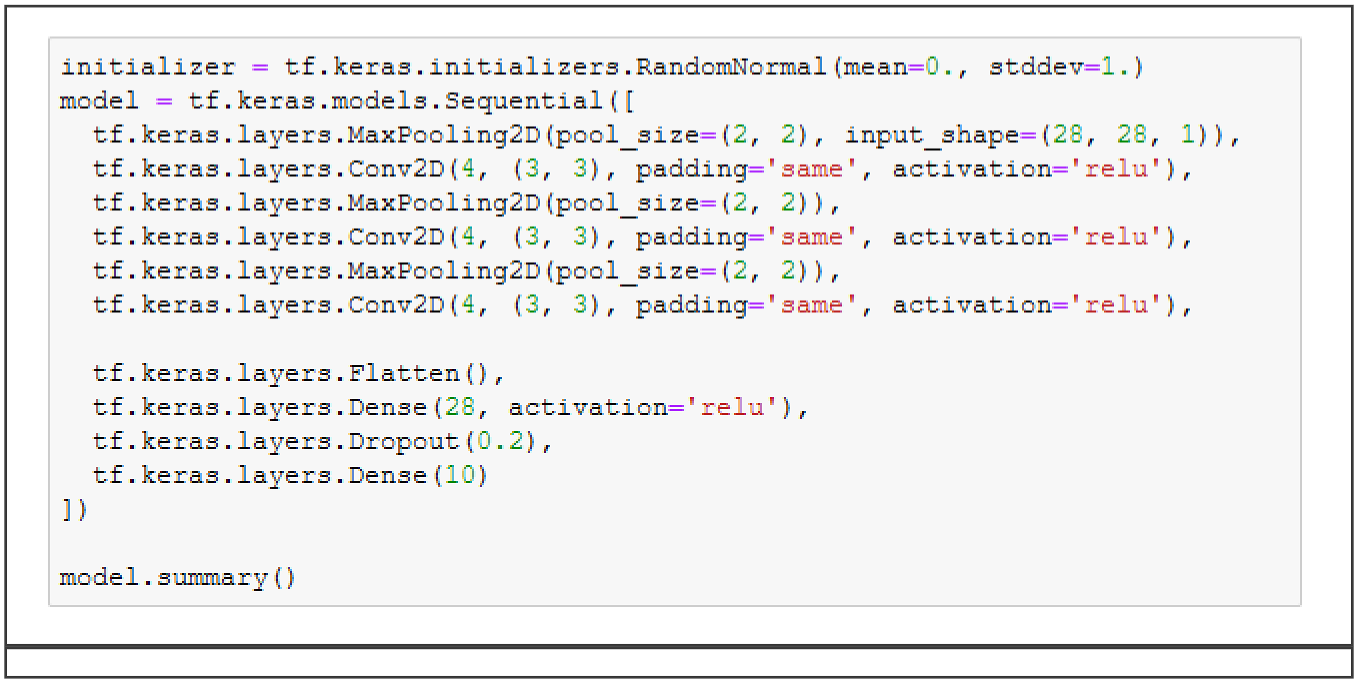
```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = np.expand_dims(x_train, 3)
x_test = np.expand_dims(x_test, 3)
```

The input data dimension is (batch, 28, 28, 1). Batch size is the number of samples to use for a training iteration.

The model uses the following layers; convolution, max pooling, fully connected, flatten and dropout. The description of these layers is out of the scope of this application note but you can find some information on the Internet.

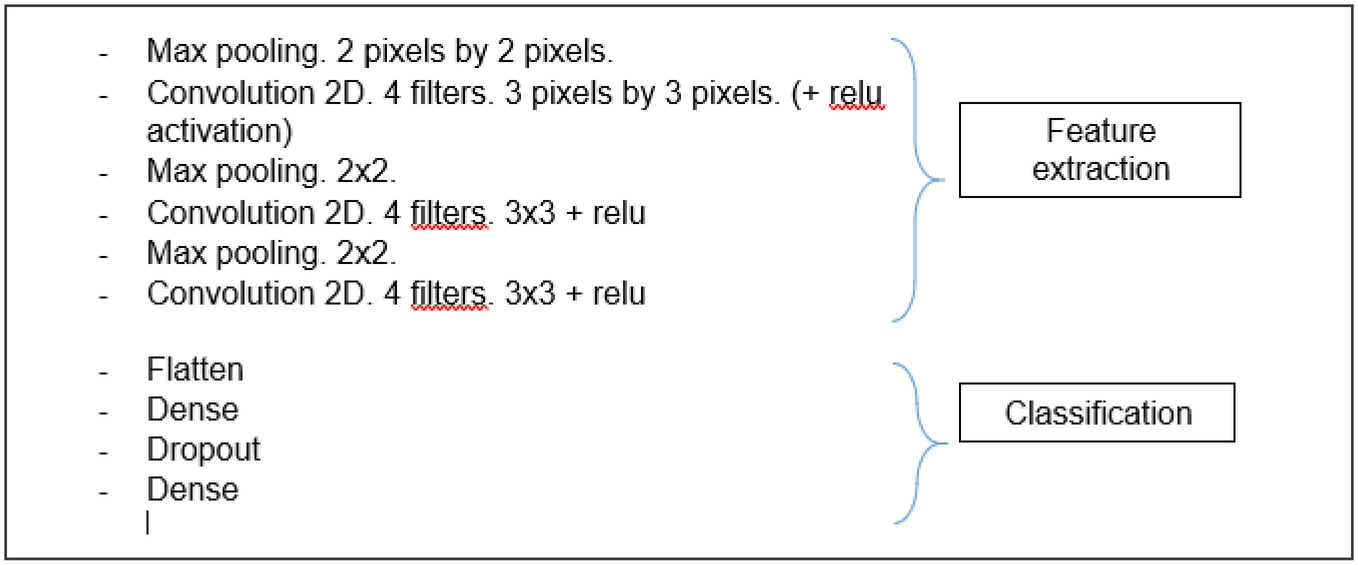


The model definition is done with the following code.



```
initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=1.)
model = tf.keras.models.Sequential([
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2), input_shape=(28, 28, 1)),
  tf.keras.layers.Conv2D(4, (3, 3), padding='same', activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
  tf.keras.layers.Conv2D(4, (3, 3), padding='same', activation='relu'),
  tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
  tf.keras.layers.Conv2D(4, (3, 3), padding='same', activation='relu'),

  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(28, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])

model.summary()
```

The layers come in this order:

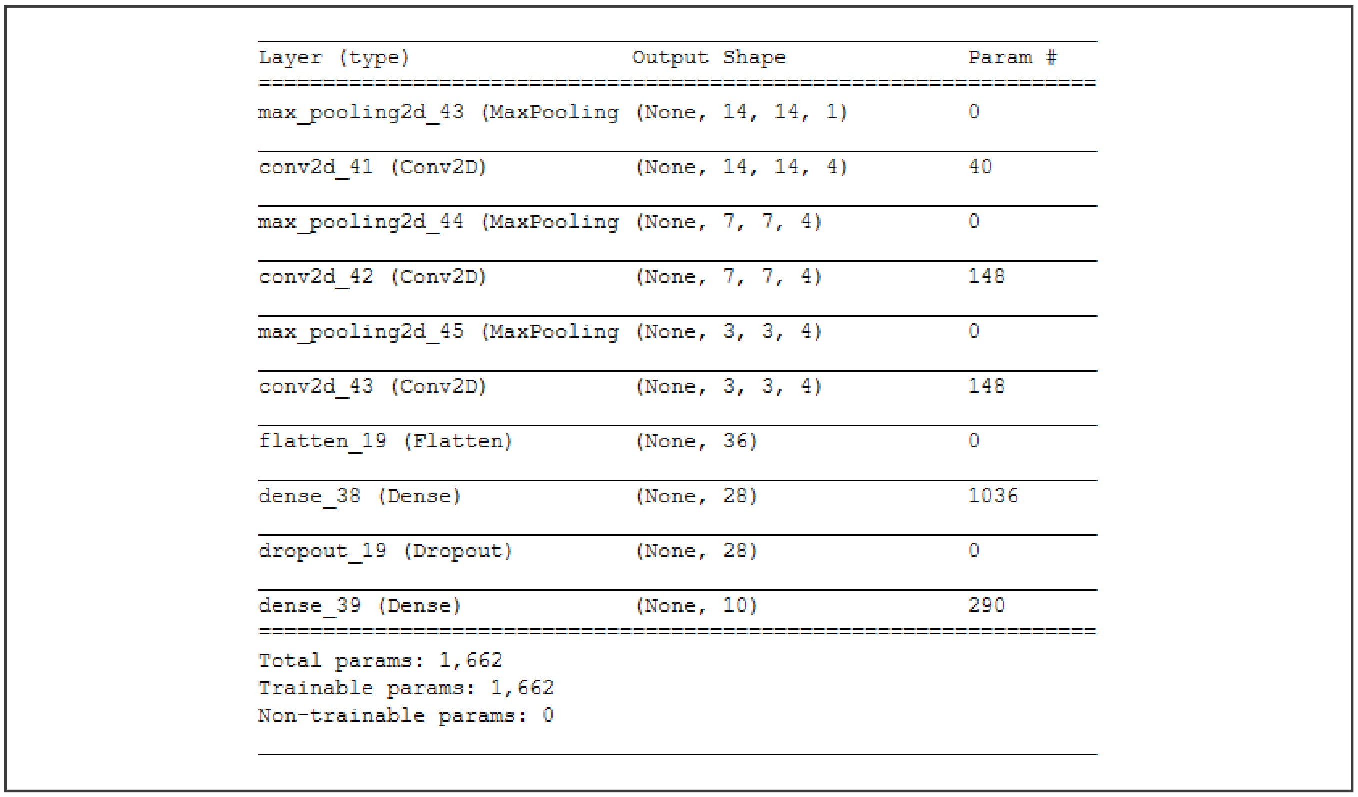


- Max pooling. 2 pixels by 2 pixels.
- Convolution 2D. 4 filters. 3 pixels by 3 pixels. (+ relu activation)
- Max pooling. 2x2.
- Convolution 2D. 4 filters. 3x3 + relu
- Max pooling. 2x2.
- Convolution 2D. 4 filters. 3x3 + relu

- Flatten
- Dense
- Dropout
- Dense

This model is a convolutional neural network. It is not usual to start a convolutional neural network by a max pooling layer. It is useful here because it helps to decrease the number of parameters. Our target MCUs for this application have a small memory. It is like converting the images from 28x28 to 14x14 as a pre-processing and skipping the first layer.

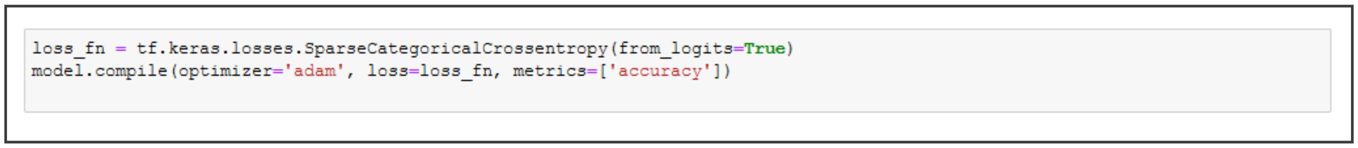
The input shape for max_pooling_1 is (None, 28, 28, 1). The output shape for each layer is given in the following table.



```
Layer (type)                 Output Shape              Param #
=================================================================
max_pooling2d_43 (MaxPooling (None, 14, 14, 1)         0
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 14, 14, 4)         40
_________________________________________________________________
max_pooling2d_44 (MaxPooling (None, 7, 7, 4)           0
_________________________________________________________________
conv2d_42 (Conv2D)           (None, 7, 7, 4)           148
_________________________________________________________________
max_pooling2d_45 (MaxPooling (None, 3, 3, 4)           0
_________________________________________________________________
conv2d_43 (Conv2D)           (None, 3, 3, 4)           148
_________________________________________________________________
flatten_19 (Flatten)         (None, 36)                0
_________________________________________________________________
dense_38 (Dense)             (None, 28)                1036
_________________________________________________________________
dropout_19 (Dropout)         (None, 28)                0
_________________________________________________________________
dense_39 (Dense)             (None, 10)                290
=================================================================
Total params: 1,662
Trainable params: 1,662
Non-trainable params: 0
_________________________________________________________________
```

The model has only 1662 parameters. This is a small number. Usually models have millions of parameters. Our model still can be trained to a high accuracy as the task is simple.

The next choice is about the loss, optimizer and metric.



```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])
```

Here, a loss function called sparse categorical cross-entropy is used. It is a loss that is useful for classification problems on data with exclusive labels. In our problem, an image can be either a 0 or a 7 but not both. Some details about this loss function can be found on TensorFlow's webpage: https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy.

The Adam optimizer is used. This is an updated version of the stochastic gradient descent. You can also read about Adam optimizer in TensorFlow's documentation.

The accuracy metric is used. This corresponds to the true positive plus the true negatives on the total number of samples.

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

You can read more about it on Wikipedia: https://en.wikipedia.org/wiki/Precision_and_recall.

The model is then trained for 100 epochs. The batch size is 32. The samples are processed by packets of 32.

```
history = model.fit(x_train,
                    y_train,
                    validation_data=[x_test, y_test],
                    batch_size=32,
                    epochs=100)
```

Here is a view of the training for the first 3 epochs and the last 3 epochs.

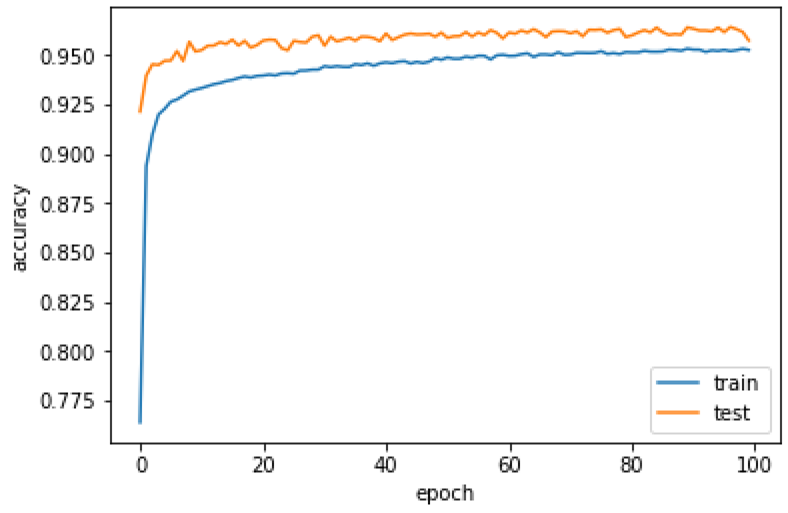
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/100
60000/60000 [==============================] - 13s 222us/sample - loss: 0.
7228 - acc: 0.7640 - val_loss: 0.2581 - val_acc: 0.9213
Epoch 2/100
60000/60000 [==============================] - 13s 210us/sample - loss: 0.
3421 - acc: 0.8937 - val_loss: 0.1958 - val_acc: 0.9395
Epoch 3/100
60000/60000 [==============================] - 13s 210us/sample - loss: 0.
2922 - acc: 0.9097 - val_loss: 0.1730 - val_acc: 0.9452
```

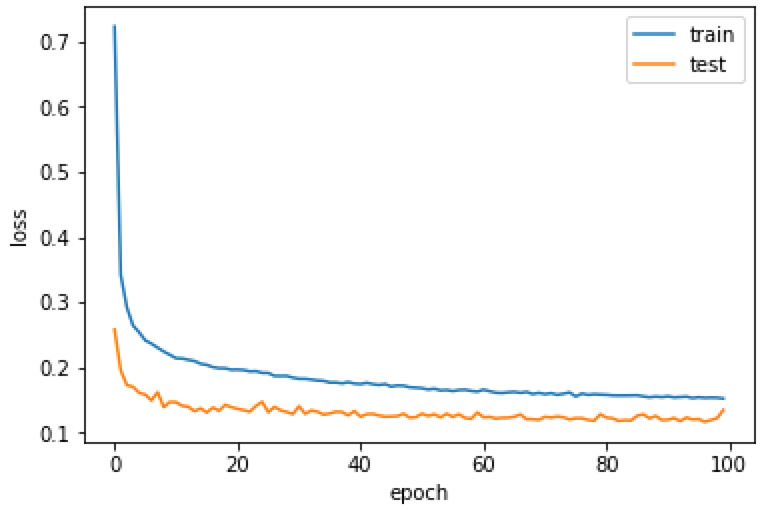
```
Epoch 98/100
60000/60000 [==============================] - 14s 239us/sample - loss: 0.
1538 - acc: 0.9524 - val_loss: 0.1194 - val_acc: 0.9630
Epoch 99/100
60000/60000 [==============================] - 14s 230us/sample - loss: 0.
1536 - acc: 0.9531 - val_loss: 0.1223 - val_acc: 0.9616
Epoch 100/100
60000/60000 [==============================] - 15s 248us/sample - loss: 0.
1524 - acc: 0.9525 - val_loss: 0.1347 - val_acc: 0.9573
```

Some progress can be seen in the loss and accuracy. Loss is decreasing and accuracy is rising close to 100%. At the end, a value in our validation set should be recognized 95% of the time. That's a good result.

Let us now visualize how the loss and accuracy evolve through the 100 epochs of the training.

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```





The accuracy is going up and the loss is going down for both training and test set. Usually the training curves are better than the test curves. Better meaning higher for the accuracy and lower for the loss. Here, the test curves are better than the training curves. It may be that the test images are easier to classify than the training examples due to the fact that the test set has less confusing examples than the training set.

Then, the model can be saved in h5 format with the following line.

```
model.save('../../mnist.h5')
```

NOTE

This data set can achieve a higher performance when other networks are used (>98%). However, 95% is enough for this application note. LeNet is an example of neural network that performs better but requires more memory and more CPU cycles.

Table 1.

	Parameters	Accuracy
LeNet	1,256,080	98.08%
Our model	1,662	95.13%

NOTE

This trained model is included in the AN software zip.

5 Workflow for porting the DL model on microcontroller

The eiQ Glow tool could be directly downloaded from the corresponding NXP webpage (Glow installer for windows). The eIQ glow user guide details and explains all the steps of the included tools. If you are not already familiar with glow, see section **4) Model compilation** and its subsections **bundle generation** and **model profiling.**

Since the available model from the previous section is in Keras format, one prior step is to convert it from the Keras format (.h5) to the ONNX format (.onnx), since The Glow compiler currently has support only for Caffe2, ONNX, and TensorFlowLite model formats. For that conversion, we run a simple python script (attached in AN software zip) that uses onnxmltools library.

As documented, The main steps are to execute the model profiling tool in order to generate the quantization profile and then the model compiler in order to generate the bundle:

Generating quantization profiling

image-classifier.exe current_test_image.png -image-mode=0to1 -image-layout=NHWC -model=MNIST_NXP.onnx -onnx-define-symbol=N,1 -model-input-name=max_pooling_1_input -dump-profile=profile.yaml

Generating the bundle for M7

model-compiler.exe -backend=CPU -float-abi=hard -model=MNIST_NXP.onnx -target=arm -mcpu=cortex-m7 -onnx-define-symbol=N,1 -emit-bundle=bundle_m7 -load-profile=profile.yaml -network-name=mnist_nxp -convert-placeholders -dump-graph-DAG=quantized_graph.pdf

Generating the bundle for M0

model-compiler.exe -backend=CPU -model=MNIST_NXP.onnx -target=arm -mcpu=cortex-m0 -onnx-define-symbol=N,1 -emit-bundle=bundle_m0 -load-profile=profile.yaml -network-name=mnist_nxp -convert-placeholders -dump-graph-DAG=quantized_graph.pdf

NOTE

Main difference between both execution is the argument about the target CPU (-mcpu). Besides, the argument -float-abi=hard (specifying that the FPU can be used) is left for M0.

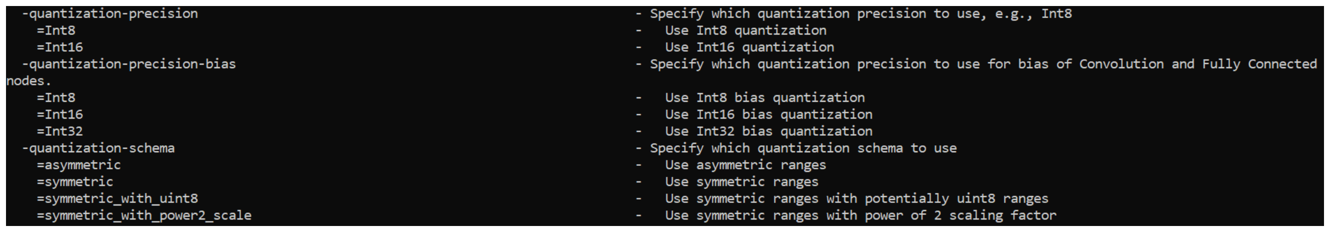
NOTE

-convert-placeholders is an important argument that ensures that the model input and output nodes are converted to the quantized format.

NOTE

-dump-graph-DAG=quantized_graph.pdf generates the graph of the bundle showing details of each layer including quantization parameters (used later).

The default (previously used) quantization specification is asymmetric with Int8 for the weights, Int32 for the biases, but this can be changed with user preferences as shown in Figure 4.

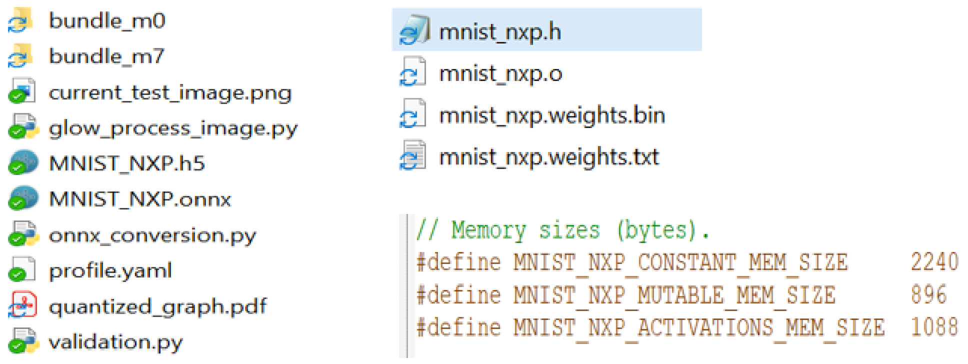


```
  -quantization-precision                                                   - Specify which quantization precision to use, e.g., Int8
    =Int8                                                                   -   Use Int8 quantization
    =Int16                                                                  -   Use Int16 quantization
  -quantization-precision-bias                                              - Specify which quantization precision to use for bias of Convolution and Fully Connected
nodes.
    =Int8                                                                   -   Use Int8 bias quantization
    =Int16                                                                  -   Use Int16 bias quantization
    =Int32                                                                  -   Use Int32 bias quantization
  -quantization-schema                                                      - Specify which quantization schema to use
    =asymmetric                                                             -   Use asymmetric ranges
    =symmetric                                                              -   Use symmetric ranges
    =symmetric_with_uint8                                                   -   Use symmetric ranges with potentially uint8 ranges
    =symmetric_with_power2_scale                                            -   Use symmetric ranges with power of 2 scaling factor
```

Figure 4. Model-compiler.exe -help

In order to obtain the image *current_test_image.png,* the attached python script *validation.py* selects a random image from Keras mnist data set, saves the current image, feeds it in the model as input and prints the model output for this image.

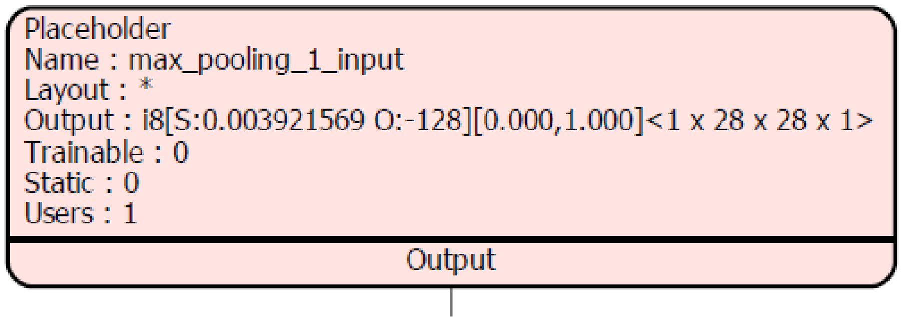
After executing the glow commands, you should be able to obtain two glow bundles (one four M0 and one for M7) with respectively four files as depicted in Figure 5 where the header file indicates the memory consumption.



```
// Memory sizes (bytes).
#define MNIST_NXP_CONSTANT_MEM_SIZE     2240
#define MNIST_NXP_MUTABLE_MEM_SIZE      896
#define MNIST_NXP_ACTIVATIONS_MEM_SIZE  1088
```

Figure 5. Different screenshots of glow bundle files

At this stage, we can expect the model graph generated by Glow as PDF. This shows details of each layer including quantization parameters.



Placeholder
Name : max_pooling_1_input
Layout : *
Output : i8[S:0.003921569 O:-128][0.000,1.000]<1 x 28 x 28 x 1>
Trainable : 0
Static : 0
Users : 1

Figure 6. Screenshot from quantized_graph.pdf

For instance, the input placeholder, according to the graph of our model, is expected to have a quantization scale; S:0.003921569 (corresponds to 1/255) and an offset of O:-128. Given these parameters and knowing the original image input range (0to1), we can conclude that the image range for quantized model should be -128 to127.

Now we can convert one static test image into a C array that will be loaded into Flash and used as input for the model inference on the board. The **glow_process_image.py** python script can be found in the SDK at <SDK_dir>\middleware\eiq\glow\examples\common. This script generates a input_image_test.inc file that will be used in the next section. Note that the generated file is 28*28*1=784 bytes in size because of the 28x28 pixel monochromatic image which uses

1-byte (int8) fixed-point integer input that the quantized model requires. The following command converts the image to int8 and saves it in a txt format (.inc file):

```
python glow_process_image.py -image-path=current_test_image.png
-output-path=test_image.inc -image-layout=NHWC -image-channel-order=BGR -image-type=int8
-image-mode=neg128to127
```

Now that the M7 glow bundle is generated, the **section 5.2 integrating the bundle** of the eIQ glow user guide is the main reference in order to inference the model on one of the available NXP M7 MCUs (RT1050/RT1060/RT1064/RT1160/RT1170).

1. Open up MCUXpresso IDE and select a new workspace
2. Install the RT1170 SDK into the "Installed SDKs" tab by dragging-and-dropping the **RT1170 SDK .zip** file downloaded earlier into the Installed SDK window. A dialog box comes up. Click OK to continue the import.
3. In the Quickstart Panel in the lower left corner, click Import SDK examples(s), select your RT board and click Next, Expand the eiq_examples category and select the glow_lenet_mnist example and make sure UART is selected for the SDK Debug Console. Click Finish.
4. Now we need to add the files generated in the previous section into this project. There are 3 files that need to be included and are found in the "source" folder that was created by the compiler. The fourth file is the test image. Remember that the "mnist_nxp" name comes from the "-network-name" argument you gave when running model-compiler.
   a. mnist_nxp.h
   b. mnist_nxp.o
   c. mnist_nxp.weights.txt
   d. test_image.inc
5. The original "mnist" files could be deleted and the input_image.inc could be deleted. So the source folder looks as follows:

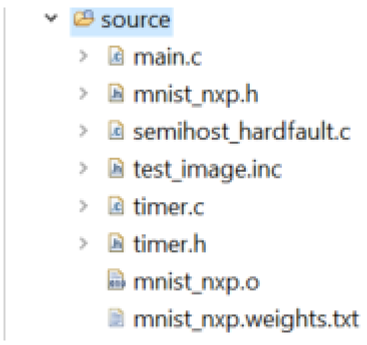
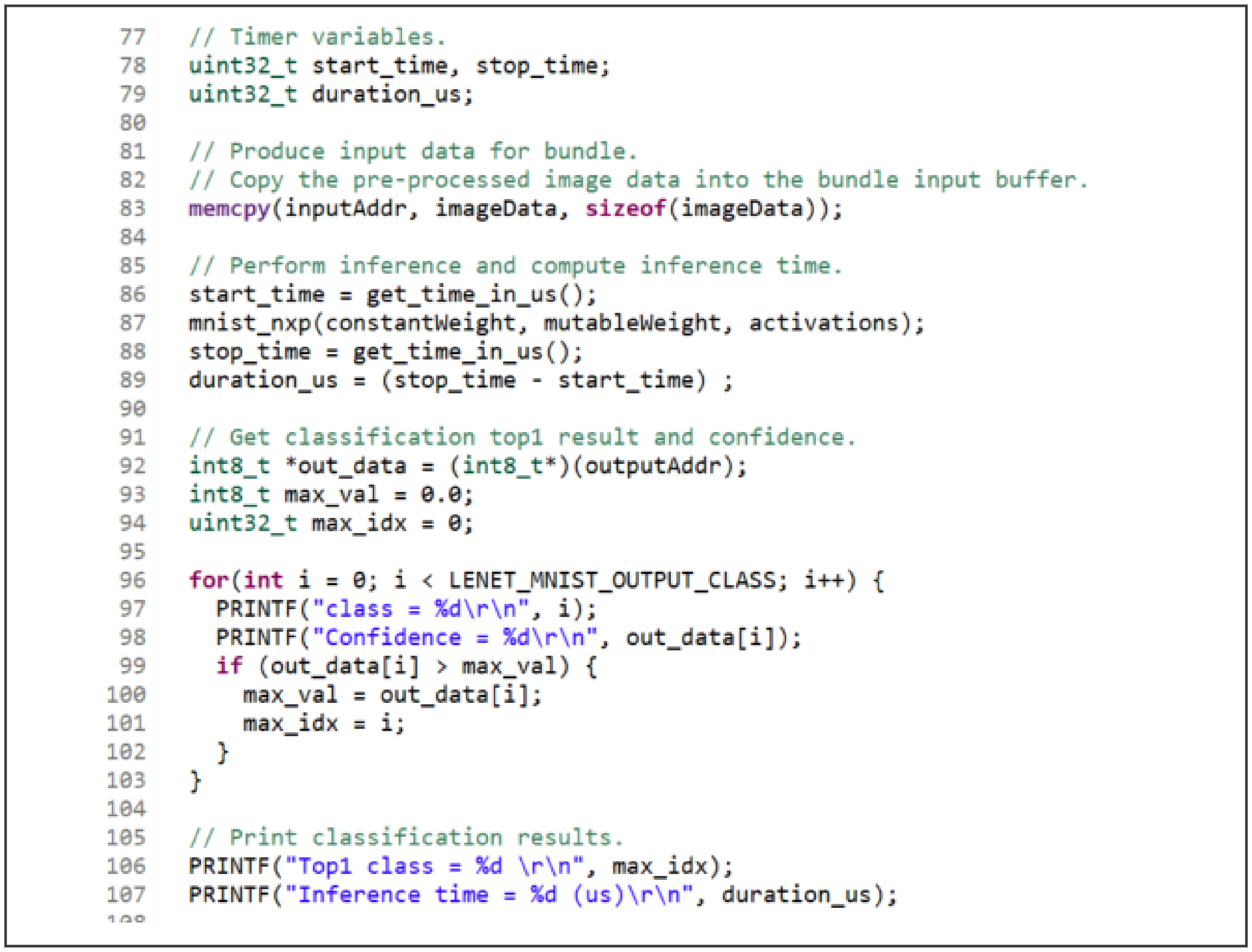


Figure 7. Project source folder

6. Change the project properties and modify the linker options such that it includes the bundle object file when linking the application: Right-click the project and select "Properties". b. Select "C/C++ Build" > "Settings". c. In the "Tool Setting" tab, select "MCU C++ Linker" > "Miscellaneous". d. Add the bundle to "Other objects". e. Click "Add..." and specify the relative path to the bundle in the project. That is, "../source/mnist_nxp.o".
7. Everywhere in the main.c file that lenet_mnist is used, it should be changed to just "mnist_nxp" and use the new variable names created by the generated files (MNIST_NXP_MEM_ALIGN, MNIST_NXP_CONSTANT_MEM_SIZE, MNIST_NXP_MUTABLE_MEM_SIZE, MNIST_NXP_ACTIVATIONS_MEM_SIZE and mnist_nxp)
8. Change input address to MNIST_NXP_max_pooling_1_input and output address to MNIST_NXP_dense_2__1. These are found in the minst_nxp.h file.
9. Change the input size to 28x28x1 and change the imageDate to test_image.inc
10. Change the output cast to int8_t *out_data = (int8_t*)(outputAddr);

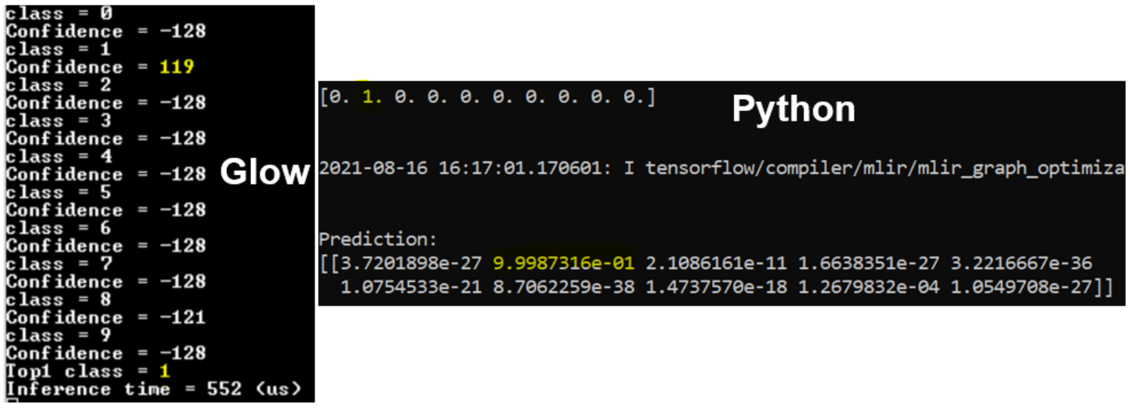
11. Change int8_t max_val = 0.0;
12. Change code to include printing of confidence of each class
13. Since the mnist_nxp model is very fast, change the printed duration to us instead of ms.



```
// Timer variables.
uint32_t start_time, stop_time;
uint32_t duration_us;

// Produce input data for bundle.
// Copy the pre-processed image data into the bundle input buffer.
memcpy(inputAddr, imageData, sizeof(imageData));

// Perform inference and compute inference time.
start_time = get_time_in_us();
mnist_nxp(constantWeight, mutableWeight, activations);
stop_time = get_time_in_us();
duration_us = (stop_time - start_time) ;

// Get classification top1 result and confidence.
int8_t *out_data = (int8_t*)(outputAddr);
int8_t max_val = 0.0;
uint32_t max_idx = 0;

for(int i = 0; i < LENET_MNIST_OUTPUT_CLASS; i++) {
  PRINTF("class = %d\r\n", i);
  PRINTF("Confidence = %d\r\n", out_data[i]);
  if (out_data[i] > max_val) {
    max_val = out_data[i];
    max_idx = i;
  }
}

// Print classification results.
PRINTF("Top1 class = %d \r\n", max_idx);
PRINTF("Inference time = %d (us)\r\n", duration_us);
```

14. Clean, build, and debug the project.

Finally we can see the model output of the inferenced image on the terminal. We should be able to conclude a match between the output of the model running with glow on the embedded device and the output of the model running on PC using python script. For more testing you can iterate; run validation.py (to save new random digit image) then run python glow_process (to convert image) and finally include the new generated test_image in the project.



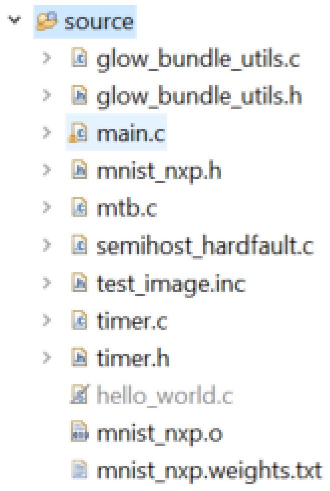
```
class = 0
Confidence = -128
class = 1
Confidence = 119
class = 2
Confidence = -128
class = 3
Confidence = -128
class = 4
Confidence = -128
class = 5
Confidence = -128
class = 6
Confidence = -128
class = 7
Confidence = -128
class = 8
Confidence = -121
class = 9
Confidence = -128
Top1 class = 1
Inference time = 552 (us)
```

```
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]

2021-08-16 16:17:01.170601: I tensorflow/compiler/mlir/mlir_graph_optimiza

Prediction:
[[3.7201898e-27 9.9987316e-01 2.1086161e-11 1.6638351e-27 3.2216667e-36
  1.0754533e-21 8.7062259e-38 1.4737570e-18 1.2679832e-04 1.0549708e-27]]
```

Figure 8. Match between quantized glow model on embedded device and original model running on PC

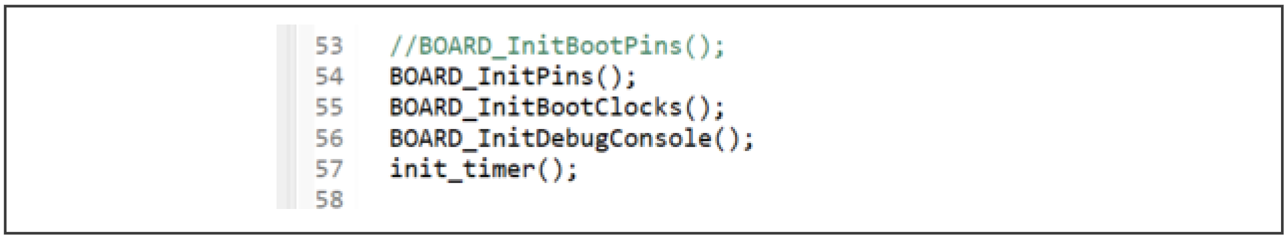
Now that you are able to run the fully quantized model solving MNIST on an RT device, the next steps show how to inference the model on an M0 MCU using MCUXpresso IDE and extending eIQ on SDKs of different family boards, such as SDK_x_FRDM-KE02Z40M, SDK_x_FRDM-KE04Z, SDK_x_FRDM-KE16Z, SDK_x_LPC845BREAKOUT, SDK_x_FRDM-K32L2B, SDK_x_FRDM-KV11Z.

1. On the same previously used workspace, install the SDK for the preferred device (list above) from www.mcuxpresso.nxp.com
2. Import a simple hello_world project from the corresponding SDK, still in the same SDK used for the RT project
3. Include the bundle files inside the source folder of your project
4. Copy the glow bundle utils found under eiq<glow<bundle_utils and the main source file from the RT project (glow_bundle_utils.c, glow_bundle_utils.h and main.c) to the source folder of your project. The test image should be copied as well. In order to calculate the inference time, timer.c and timer.h should be copied. It should now look the following:



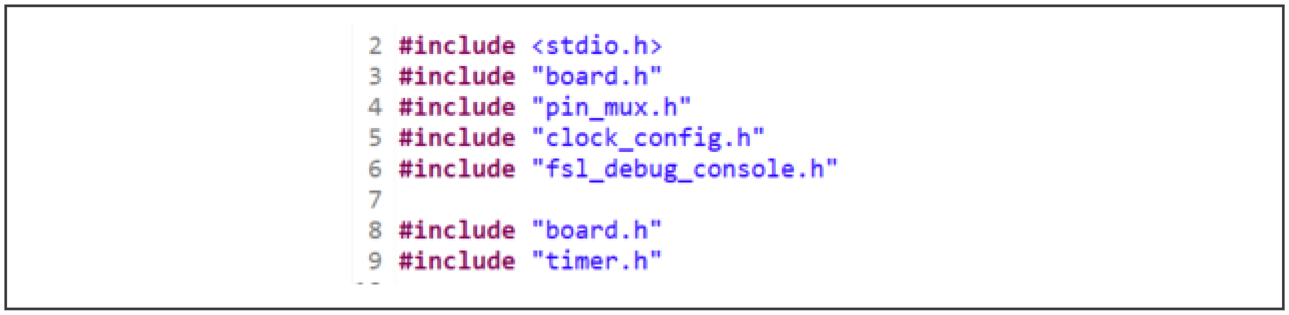
```
source
  glow_bundle_utils.c
  glow_bundle_utils.h
  main.c
  mnist_nxp.h
  mtb.c
  semihost_hardfault.c
  test_image.inc
  timer.c
  timer.h
  hello_world.c
  mnist_nxp.o
  mnist_nxp.weights.txt
```

5. Add the mnist_nxp.o compiled code from the Glow compiler output by clicking the Miscellaneous settings→Other objects→Add button to include the object file "${workspace_loc:/${ProjName}/source/NXP_MNIST.o}"
6. Adapt the board initialization of the main.c (first code lines of the main function) to the target board copying from the original main function of the hello_world.c (this will vary from one board to another).



```
//BOARD_InitBootPins();
BOARD_InitPins();
BOARD_InitBootClocks();
BOARD_InitDebugConsole();
init_timer();
```

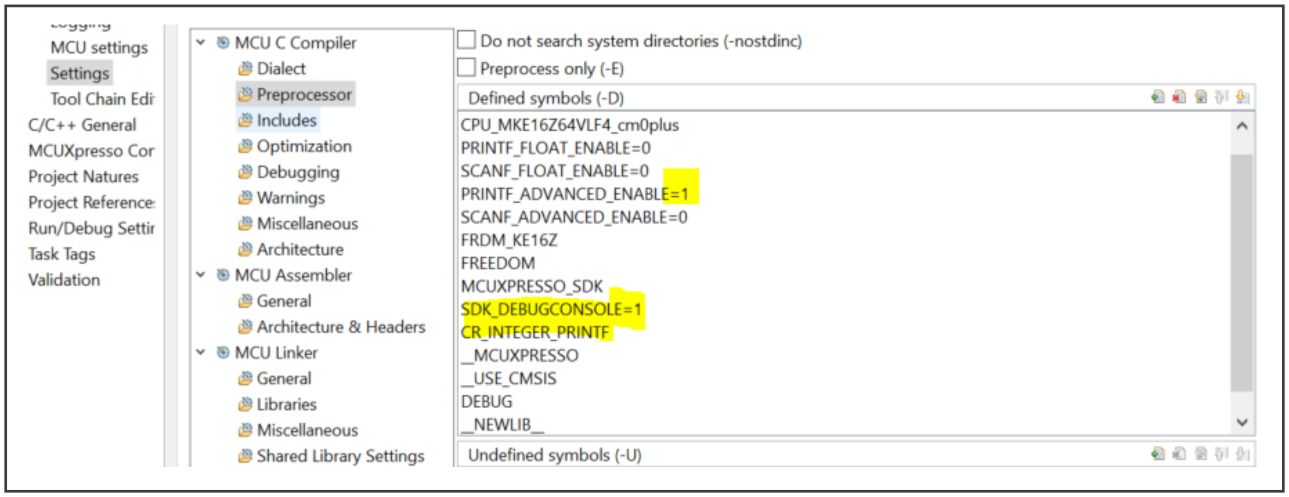
7. Do the same for the includes of the header files inside main.c (example: #include peripherals.h should be removed) (this will vary from one board to another).



```
#include <stdio.h>
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_debug_console.h"

#include "board.h"
#include "timer.h"
```

8. Change the used library under Settings, managed Linker Script to NewLib (nohost (for UART) or semihost depending on console preference)

9. Verify the following preprocessor macros to enable correct printing of the output



☐ Do not search system directories (-nostdinc)
☐ Preprocess only (-E)

Defined symbols (-D)

```
CPU_MKE16Z64VLF4_cm0plus
PRINTF_FLOAT_ENABLE=0
SCANF_FLOAT_ENABLE=0
PRINTF_ADVANCED_ENABLE=1
SCANF_ADVANCED_ENABLE=0
FRDM_KE16Z
FREEDOM
MCUXPRESSO_SDK
SDK_DEBUGCONSOLE=1
CR_INTEGER_PRINTF
__MCUXPRESSO
__USE_CMSIS
DEBUG
__NEWLIB__
```

10. Finally, prevent the hello_world.c file from compiling by right clicking on that file and selecting Properties. In the dialog box that comes up, select the C/C++ Build category, and check the Exclude resource from build option. Then click Apply and Close.

11. Clean, build, and debug project

6 Results and performance comparison

This final session lists the memory consumption (Flash and RAM) and the inference time of the MNIST NXP model running on different tested Arm Cortex-M0 based microcontrollers.

- LPC845 with M0+ core clocked at 30 MHz (board is LPC845-BRK with 64 kB Flash and 16 kB RAM)
- KE16Z with M0+ core clocked at 48 MHz (board is FRDM-KE16Z with 64 kB Flash and 8 kB RAM)
- KE02Z64 with M0+ core clocked at 40 MHz (board is FRDM-KE02Z40M with 64 kB Flash and 4 kB RAM)
- K32L2 with M0+ core clocked at 48 MHz (board is FRDM-K32L2B with 256 kB Flash and 32 kB RAM)

- KV11Z with M0+ core clocked at 75 MHz (board is FRDM-KV11Z with 128 kB Flash and 16 kB RAM)

The memory consumption values correspond to the footprint of the complete software project (and not only the deep learning model) including all required essential drivers for running the MCU and communicating output to serial COM. All these software projects are attached in the AN zip. Although the ported model is exactly the same (so its memory consumption is exactly the same), we still see differences in memory consumption between the different boards (see Table 2). This is explained by the difference of the included low-level drivers which are lighter for small microcontrollers compared to the RT microcontroller for example.

For each device, the inference time is calculated for two different configurations; either the model parameters are read directly from non-volatile Flash memory during model inference (this could be ensured by adding “const” in front of the weight array definition) or by copying these parameters to the volatile RAM during booting time. The difference is about a trade-off between lower RAM consumption (model kept in Flash) or faster inference time (model copied to RAM).

Table 2. Performance comparison table

Model	Flash	RAM	Inference time
RT1060 (Flash)	36312 B	11 KB	0.63 ms
RT1060 (RAM)	36312 B	13504 B	0.55 ms
KE16Z (Flash)	25308 B	4012 B	13.5 ms
KE16Z (RAM)	25308 B	6252 B	12.9 ms
KE02 (Flash)	22204 B	3372 B	16.7 ms
KE02 (RAM)	Not possible (exceeding available RAM)		
LPC845 (Flash)	18412 B	5144 B	36.7 ms
LPC845 (RAM)	18412 B	7384 B	35.6 ms
K32L2 (Flash)	22748 B	7084 B	13.3 ms
K32L2 (RAM)	22748 B	9324 B	12.9 ms
MKV11Z (Flash)	23192 B	5032 B	8.5 ms
MKV11Z (RAM)	23192 B	7272 B	8.3 ms

Despite the low performances of the M0 based microcontrollers, we see that the inference time in all cases significantly lower than 100 ms. These results confirm that it is possible to design lightweight deep learning models that are convenient for low end microcontrollers.

7 Conclusion

This application note demonstrates that NXP low end microcontrollers (Arm Cortex-M0 based) are capable of inferencing machine learning models on the edge. Starting from an available data set, achieving an embedded model running on a simple microcontroller, the AN shows all the required steps to follow (See Workflow for porting the DL model on microcontroller), leveraging the NXP eIQ toolkit for AI/ML.

ML/DL Development: Design of a simple CNN solving handwritten classification task with MNIST data set depicts the methodology of designing a lightweight model convenient for small edge device for a particular application and Results and performance comparison concludes that the achieved performances of NXP small MCUs running such model are impressive.

8 Revision history

The following table lists the substantive changes done to this document since the initial release.

Table 3. Revision history

Revision number	Date	Substantive changes
0	18 November 2021	Initial release

Date of release: 18 November 2021
Document identifier: AN13438