POLITECNICO DI TORINO

Master of science in Computer Engineering

On the deployment of Artificial Neural Networks (ANN) in low cost embedded systems

Supervisor

Sanchez Sanchez Edgar Ernesto (DAUIN)

Candidate

Butt Usman Ali

23rd July 2021

Contents

List of Figures

List of Tables

List of Charts

List of Code Snippet

Abstract

In this thesis work we explored the deployment of AI solutions on microcontrollers. All the AI solutions currently available and thought off to be deployed on microcontrollers were explored, tested and verified on hardware. 32-bit microcontroller is made part of the development since 8-bit microcontrollers are not supported by any platform for AI applications. Stm32l432kc with 64Kbytes of SRAM and 256Kbytes of flash memory is chosen for testing AI models. Current AI solutions for microcontrollers differ from each other on vendor propriety, open source and compiler bases. Propriety Google TensorFlow lite for microcontroller, stm32 AI cube and open source library NNOM were used for building and training AI models. Results for each AI solution is compared with others on four core parameters flash occupancy, ram occupancy, time for inference and output deviation. Optimizing techniques such as quantization and using only c source code for microcontroller were emphasized. C and C++ yields a huge amount of difference in code size and inference time. At the end AI solutions were tested for compile-time injection faults by flipping the memory bits and altering the microcontroller code.

Introduction

This thesis work focuses on implementation of Artificial Intelligence (AI) on microcontrollers and custom designed SOC's. Every electronic device has some kind of processing and decision making put in to it. Microcontrollers and custom designed SOC's are the only parts which are programmed to perform tasks related to processing and decision making. These tasks are hardcoded in the microcontroller firmware.

Rise of AI in commercial products and the expected increase in edge and IOT devices made the researchers to explore the AI deployment on edge and IOT devices. AI advantages and success in commercial products is another catalyst to explore its usage in edge devices.

Edge and IOT devices central processing part is microcontroller or custom SOC's. So, AI compatibility with microcontrollers and SOC's is crucial for getting started with AI on micro's and Soc's.

Development and Deployment issues

AI applications are bulky and requires a lot of processing power, time and energy. Whereas microcontrollers have limited resources. Another issue is lack of development API's. For microcontrollers we develop and train model on another platforms (windows, Linux, mac etc.) and then deploy it on microcontrollers [1] along with some supporting libraries.

Let's look at the AI development API's for microcontrollers. So far only one Platform TensorFlow Lite for microcontroller's offers development of AI applications for micro's.

TensorFlow (TF) is a platform to develop Machine Learning (ML) models. Python and JavaScript can be used for developing, training and testing the models. You can export various predefined and pre-included ML datasets and API's to develop and train your model. Its opensource and works in the cloud.

TF lite is special variant of original TF which is designed for mobile, linux, edge and microcontrollers. You can develop, train and test your models using TF lite. TF lite has reduced functionality then main TF.

TF lite subvariant TF lite for microcontroller supports AI model development for edge and microcontrollers.

TF lite for microcontrollers

Not all the functions of original TF are supported by TF lite for microcontroller. Models created by TF lite for micro can only be deployed on 32-bit microcontrollers. In other words, only 32-bit operations are supported by TF lite. This leaves us to conclusion that 8-bit microcontrollers cannot be used for AI applications or no such platform exits to create and deploy AL applications on 8-bit microcontrollers. Only cortex cores are supported for AI model deployment. So RICS architecture is also set a side. TF for microcontroller's libraries are in C++11. To compile TF for micro we need a compiler which supports C++11.

TF lite for microcontroller is just a converter. Typical workflow of developing a model and porting the model to micro is

- First, we create our model in python using TF lite.
- Then model *flatbuffer* file is created. *Flatbuffer* is automatically generated by the TF lite after successfully construction, training and validation of model.

- This *flatbuffer* file is then translated into c code using TF lite for micro converter. TF lite for micro is provided as an API.
- Finally, we take the c code. Generate a new .c or .h header file in our micro application and place the c code in it.

TF didn't explain anything about the c code generation and how to use it. It only explains high level functions which automatically fetch the c code from the file and reconstruct the model in micro. Other high-level functions such as how to input and read output are also explained in the guide.

Compiler constraints

Usually microcontrollers firmware is written in c language and various c compilers and linkers comes with the development platform for code compilation. But in the TF lite for micro case we need a compiler which can translate both c++ and c code in to equivalent binary. Why both c++ and c code? TF lite micro package is in c++, whereas when we initialize any micro and include the vendor specific libraries for any particular peripheral of micro it is found that all of them are written in c. This puts an extra burden on the compiler.

Most vendors invest in compilers only to speed up the running time of microcontroller. The general theory behind it is that faster speeds consume less power. Well its true, but in case of low end devices like microcontrollers another two parameters program size and power consumed by peripherals in general and overall power consumed play a vital role[2].

Vendor specific AI converters dependent on TF and other AI model development platforms

Deploying AI on embedded platforms is a challenging task but recent advances in machine learning and hardware design is overcoming it[3]. More and more silicon vendors are innovating not only in hardware but also in software to gain competitive advantage on their industry peers.

Renesas e-AI converter

e-AI or embedded artificial intelligence converter offered by renesas is a proprietary AI model converter. It takes a trained model as input and outputs model equivalent c code. Model can be trained in TF, Caffe or other supported AI model development platforms. Outputted model equivalent code can further be included in the project wizard for a particular renesas microcontroller. Since e-AI is a proprietary you can only use the c code with renesas e^2 Studio Ide.

Typical work flow is

- Take pretrained model from TF, keras and caffe etc.
- Fed the model into e-AI translator
- Run translator, Optimized code for Renesas microcontroller is generated.
- Include the code in renesas e^2 studio project

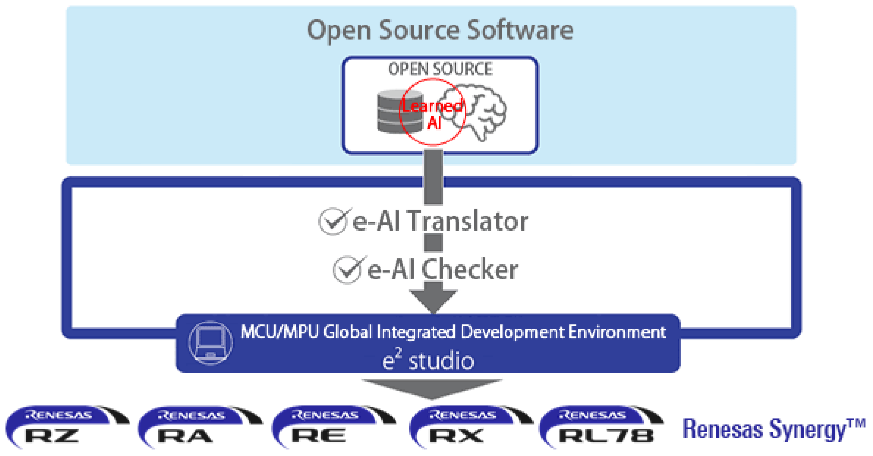


Figure 1 Renesas e-AI Anatomy [4]

Renesas recommends to use its microcontrollers with cortex-M3 core or above with suitable ROM and RAM in which one thinks its code can fit in. Not all the neural network (NN) layers are translatable by e-AI converter. Most popular and intensively used layers are supported[1].

ST microelectronics STM32Cube AI converter

ST microelectronics offered an AI solution for its line of microcontrollers named STM32Cube.AI converter. It also takes pretrained model as input and outputs equivalent c code. Code can then be included in to your microcontroller development project.

STM32Cube.AI is part of the STM32CubeMX. Cube MX offers GUI initialization of ST microcontrollers peripherals and with the click of a button initialization code for particular microcontroller is generated with all the necessary libraries included and startup file initialized. For Cube AI we just need to specify the path of the model file and initialize the micro peripherals in Cube MX. All done translate the project and all the files will be generated automatically.

Cube AI offers other functions for example you can view the model, compress or quantize it. These functions are easy to use Cube MX GUI makes it really simple and straight forward.

- Take pretrained model from TF, keras and caffe etc.
- Open STM32CubeMX. Select microcontroller. Initialize its peripherals.
- Open STM32Cube.AI from CubeMX and input the mode file. Load model (quantize if required).
- Click generate code for project and pre-initialized code generation.

[1] https://www.renesas.com/us/en/application/technologies/e-ai/translator

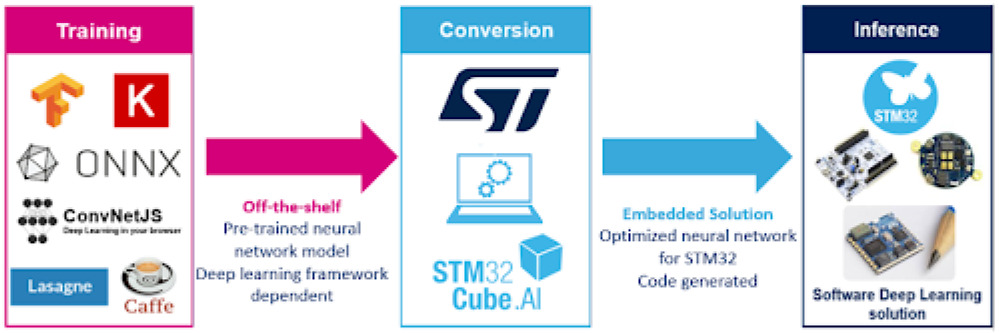

Training
ONNX
ConvNetJS
Lasagne
Caffe
Off-the-shelf
Pre-trained neural network model
Deep learning framework dependent
Conversion
STM32 Cube.AI
Embedded Solution
Optimized neural network for STM32
Code generated
Inference
STM32
Software Deep Learning solution

Figure 2 STM32 Cube AI Anatomy [5]

CubeAI supports cortex-M3 and above microcontrollers offered by stmicroelectronics. TF, Keras and Café deep learning frame works are supported by CubeAI. Output code is in c and inline assembly format.

ARM AI solutions

All the AI solutions offered by various vendors supports ARM cortex cores. ARM provides solutions for a common standard. Application developed using ARM libraries can be ported to any vendor device using the same ARM core.

Work flow is

- Create model in TF, Caffe or any supported neural network platform.
- Translate the model and make it compatible with ARM libraries.
- Run inference.

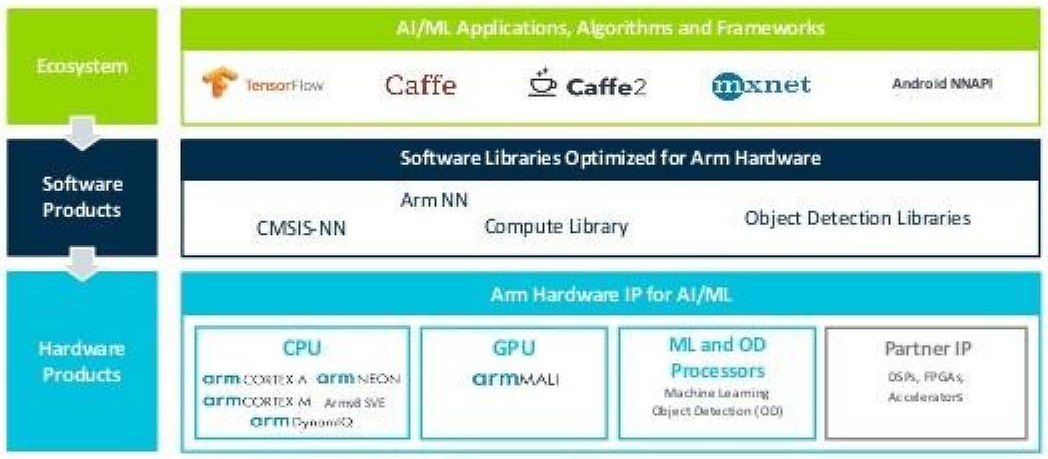


Figure 3 ARM AI core's flow [6]

Translating the model to CMSIS-NN or ARM-NN is manual. Each layer and activations must be translated according to the ARM[2] layers transform reference manual. Not all the AI neural network layers are

[2] https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/converting-a-neural-network-for-arm-cortex-m-with-cmsis-nn/single-page

translatable. If the layer is not supported by ARM other layers can be merged to achieve the functionality of the unsupported layers.

For AI support ARM provides two library packages CMSIS-NN and ARM-NN. Various other stand-alone libraries can be added with CMSIS and ARM-NN for optimization purposes.

ARM-NN for AI on CPU's and GPU's

ARM-NN targets high end CPU's and GPU's. Thesis is on AI in microcontrollers so we will not go deeper in ARM-NN.

CMSIS-NN for AI

Common microcontroller software interface standard (CMSIS) is a set of libraries which can be used to develop firmware for ARM cores irrespective of the vendor. Firmware developed using CMSIS can be ported to any vendor product as long as the core is same. CMSIS-NN is a subset of original CMSIS which provides support for deploying AI models on ARM cores.

NXP e-IQ for ARM CMSIS-NN

Nxp e-IQ platform offers a machine learning environment on its high-end CPU's and GPU's. It utilizes CMSIS-NN and ARM-NN for machine learning.

NNOM Project - Open Source

Like other converters and translators an open source project is trying to foot in the AI field. NNOM takes a keras model as input and outputs a translated c code representing the inputted model. This c code contains the layers of model, its weights and biases. The code can then be deployed in the microcontroller project and NNOM libraries can be used to load the model in micro core and perform inference.

Getting Started

In this study I took three core AI solutions to start work on. All of the three are complete in most of the respects and can be deployed on 32-bit microcontrollers.

1. Tflite for micro
    a) Google solution to deploy AI on microcontrollers. Uses ARM CMSIS libraries.
2. Stm32Cube AI
    a) ST microelectronic solution. It also uses ARM CMSIS libraries.
3. NNOM
    a) Open source project. It also depends on ARM CMSIS libraries.

Target device chosen for the work is stm32l432kc microcontroller. It has 512Kb of flash and 64Kb of RAM. Internal clock oscillator of target is used whose clock is set to 80MHz. USART of microcontroller is also activated and inference results are output on a serial window using USART. Baud rate of USART is set to 115200 bits/s.

Solutions will be compared on four main parameters which define the microcontroller performance. Memory is a major constrain in microcontroller embedded solutions. Our code foot print must fit in the available memory. Time comes next, execution must be as quick as possible. These two parameters were given precedence in many embedded solutions so in ours.

- Flash occupancy
- Ram occupancy
- Inference time
- Output deviation

We have one more parameter in our case which is deviation from original output. Since AI models are not 100% accurate so we will also consider this in our experiments. How much the individual solution output differs from the original value.

General consideration that model will behave according to its construction and training is true. In our case model is created using two different techniques sequential and functional/layered. We will compare the output deviation by training the model on same data set. Another factor in our study is model translation using STM32Cube AI. We will also compare the output deviation if any when model is translated using cube AI.

Model Construction and Training

Getting started with TF lite for microcontroller lists a sine wave getting started example. The idea is if we input a number between sine(0-2**π**) output will be between -1 to 1. Equation for the function is y=sine(x). Where x is input and y is output. Plotting the graph between x and y gives a perfect sine wave.

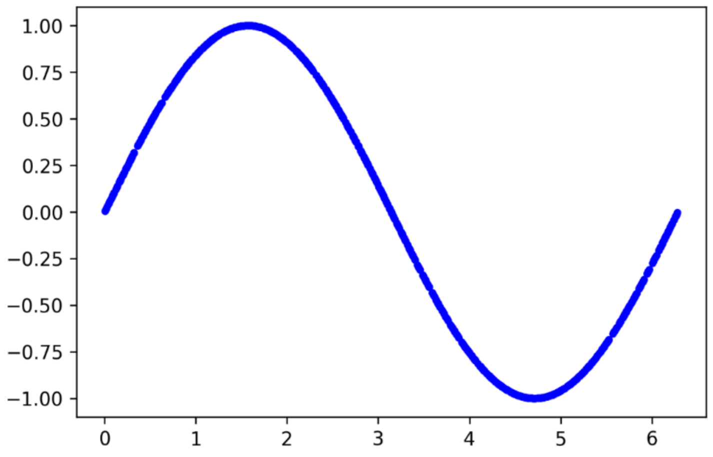


Figure 4 Sine wave 0-2pi

Google Collab is used for developing model. First, I imported TF library and all the other libraries/dependencies in the Collab project sheet. Such as NumPy for working with arrays, Math for exporting math functions, Keras for NN functions and NN model construction.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import math
from tensorflow.keras import layers
from tensorflow.keras import *
```

Code snippet 1 Imported libraries

Next a sample of 1000 is randomly generated from the input range 0-2π. Out of the total 1000, 20% samples are allotted for each test and validation sets. Remaining 60% are spared for training.

```
nsamples = 1000     # Number of samples to use as a dataset
val_ratio = 0.2     # % of samples for validation set
test_ratio = 0.2    # % of samples for test set
x_values = np.random.uniform(low=0, high=(2 * math.pi), size=nsamples)

# Split dataset training(60%), validation(20%), and test(20%)
val_split = int(val_ratio * nsamples)
test_split = int(val_split + (test_ratio * nsamples))
x_val, x_test, x_train = np.split(x_values, [val_split, test_split])
y_val, y_test, y_train = np.split(y_values, [val_split, test_split])
```

Code snippet 2 Data set test, train and validation

Visualizing the split train, test and validation dataset. Almost full range is covered by each dataset or uniform distribution.

Few values are above 1 and below -1 also the graph points are scattered. This is because when the sine function was created some random value is included with each sine output. This is important because if we don't include it we can get a sine wave with no randomness or a precise sine wave.

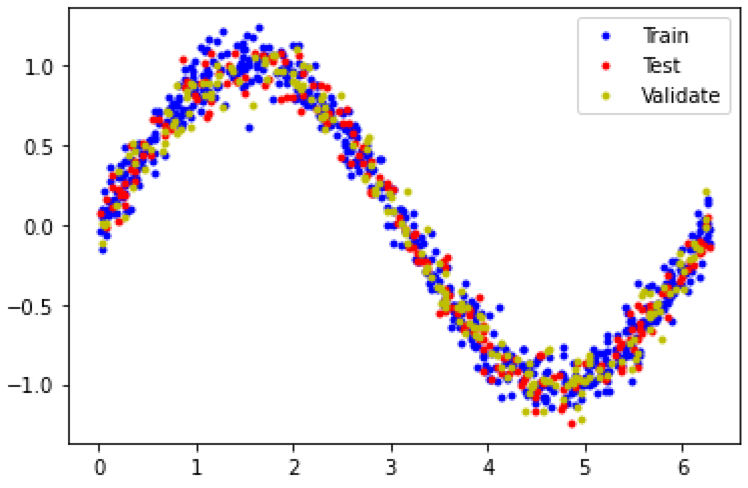


Figure 5 Model data points

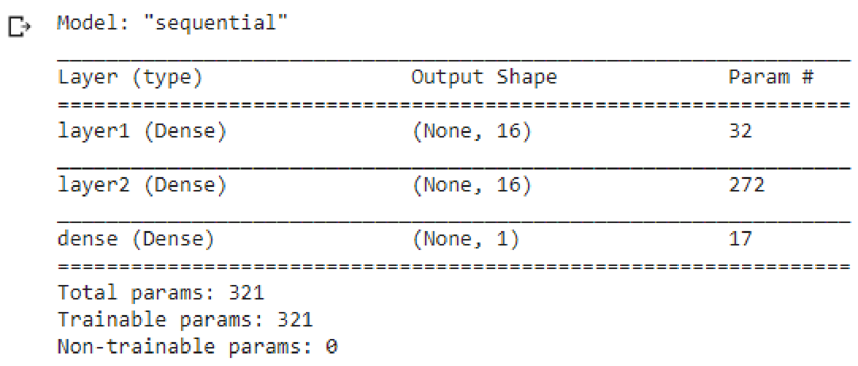
Model Construction

A sequential model having three fully connected dense neural network layers is created. Activation is relu. Input is a singe neuron and output also a single neuron. Sequential models are easy to construct and not much effort is required in programming. Input shape is a single neuron. Two hidden layers between input and output is named as layer 1 and 2. Both are dense layers. Dense layers are most commonly used layers in AI models and the simplest one's.

```
# Create a model
model = tf.keras.Sequential()
model.add(layers.Dense(16, activation='relu',name="layer1", input_shape=(1
,)))
model.add(layers.Dense(16, activation='relu',name="layer2"))
model.add(layers.Dense(1))
```

Code snippet 3 Sine wave sequential model

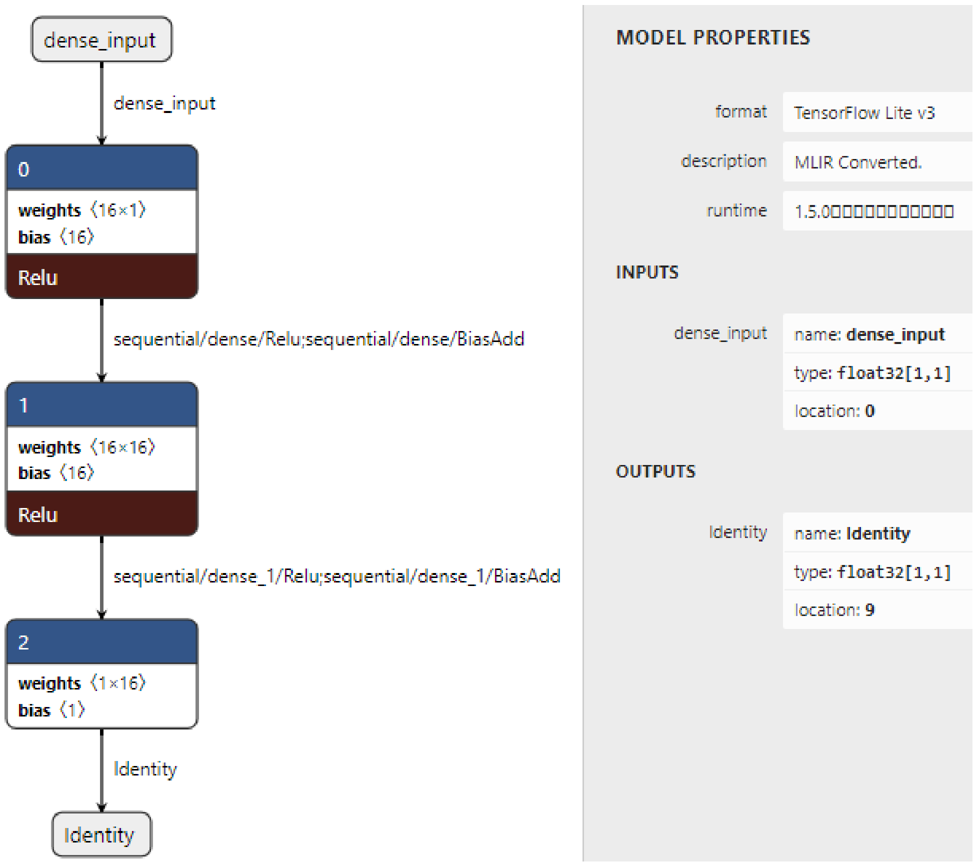
Layer-1 shape is 16 neurons with 32 parameters. Layer-2 16 neurons with 272 parameters and finally the output layer 1 neuron with 17 parameters. Model translation by google collab shows total 321 parameters.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
layer1 (Dense)               (None, 16)                32
_________________________________________________________________
layer2 (Dense)               (None, 16)                272
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
```

Figure 6 Model parameters

Going deeper in the model. Input and output are float(32-bit) type. Total weights for first layer are 16 and bias stands at 16. For second layer weights are 256 and bias 16. Output weights are 16 and bias 1. Activations are Relu and model is sequential. Netron is used to view the model and its parameters. On the right-hand side, we can see that model input and output are float types.



MODEL PROPERTIES

format	TensorFlow Lite v3
description	MLIR Converted.
runtime	1.5.0□□□□□□□□□□□

INPUTS

dense_input	name: **dense_input**
	type: **float32[1,1]**
	location: **0**

OUTPUTS

Identity	name: **Identity**
	type: **float32[1,1]**
	location: **9**

Figure 7 Model individual layers view

First layer of model has 16 neuros all of them are getting input from a single input range between 0-2π. Second layer has 16 neurons and each neuron is getting input from the previous layer 16 neurons. Finally, the output is a single neuron.

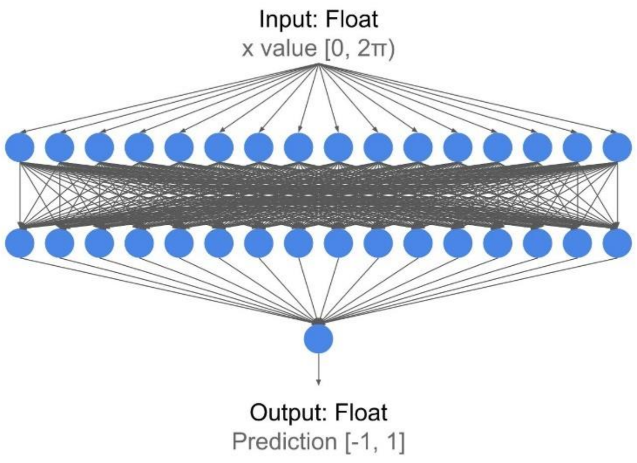


Figure 8 Model neurons view

Rmsprop optimizer is used for compiling, loss and metrics is calculated as a mean absolute error. To train the model batch size of 100 is used. Which effectively translates to (samples/batch size) 10 batches of 100 samples. 10 batches each with 100 samples is passed through the model 500 times.

```
# Add optimizer, loss function, and metrics to model and compile it
model.compile(optimizer='rmsprop', loss='mae', metrics=['mae'])


# Train model
history = model.fit(x_train,
                    y_train,
                    epochs=500,
                    batch_size=100,
                    validation_data=(x_val, y_val))
```

Code snippet 4 Training model epochs, batch size etc

After training analyzing the training and validation loss graph shows us decrease in the loss linearly. Both losses reduce simultaneously with the increasing number of epochs. This is important because if one of them is not synced with other our model will not perform correctly and huge deviation in result is obvious.

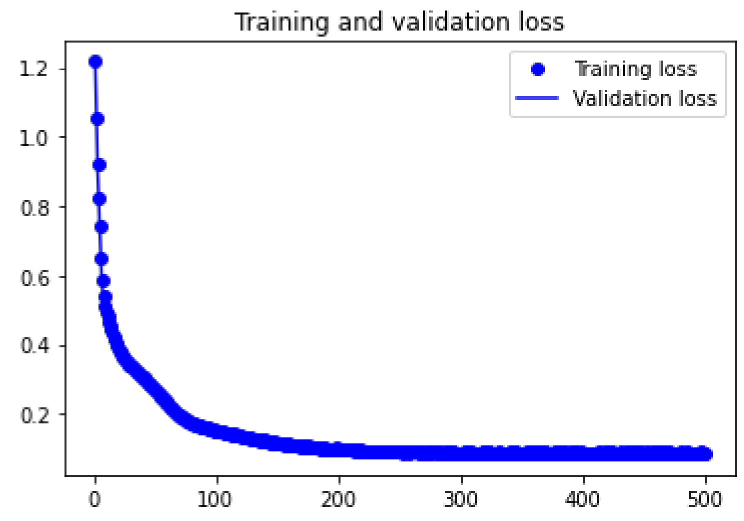


Figure 9 Model training and validation loss

Running a simple test prediction shows us a perfect sine wave. Test values are the one which were taken out from the initial sample of 1000 account 20% of the sample. Predicted values are almost in between the scattered sample and are continuous which is a positive sign and indicates that our model is behaving as expected.

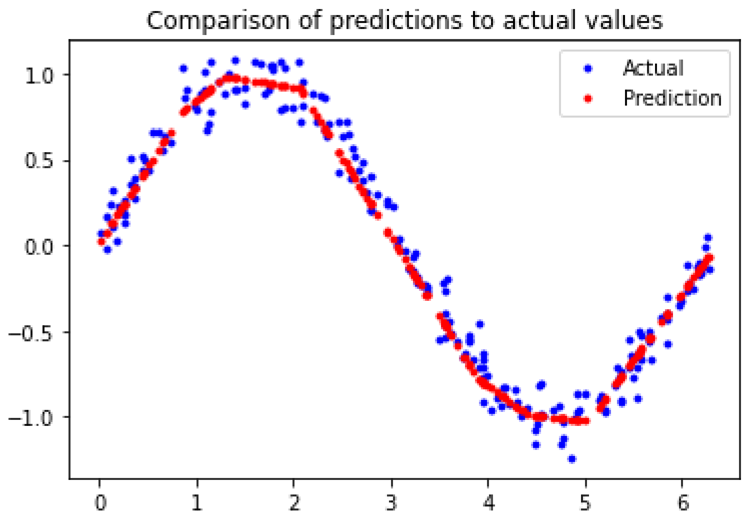


Figure 10 Model prediction

NNOM – Model construction

NNOM is an open source project. Which allows you to build and train your AI model in python and with its converter you can translate your model in c code for microcontroller. Few key points of NNOM are

- Works with models built and trained using keras Only.

- Supports only Functional/Layered model unlike TF which supports sequential.
- Must include the NNOM libraries (Python) to Keras project for model interpretation and conversion to NNOM format.
- Not all layers and activations are supported by NNOM.
- Libraries for microcontroller are in C language.

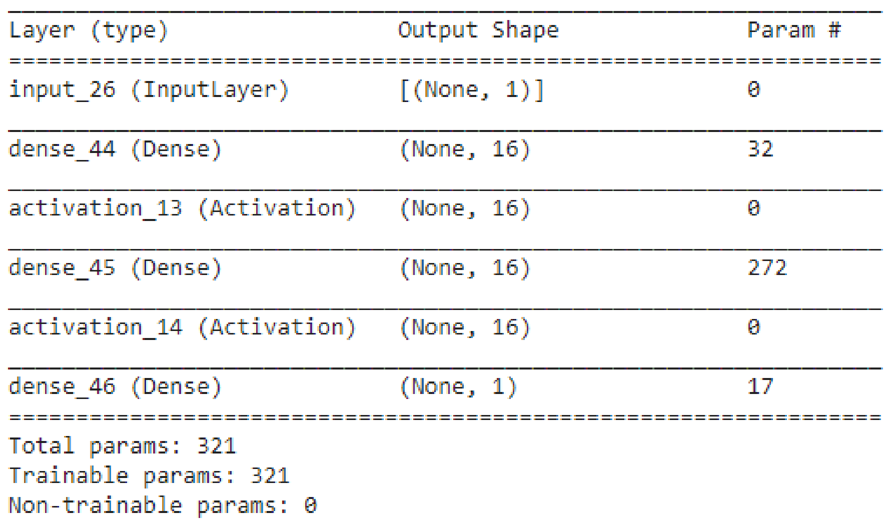
Below is the same sine wave written in both sequential and functional types. Sequential is for TF lite and functional is for NNOM.

Functional model input shape is 1. Which means single input. Next is a dense layer of 16 neurons. Which is activated by a relu function. After that another dense layer of 16 neurons activated by relu function. At last the final output layer of single neuron.

```
# Functional or layer model with independent activation functions
input = tf.keras.Input(shape=(1,))
X = tf.keras.layers.Dense(16)(input)
X = Activation("relu")(X)
X = tf.keras.layers.Dense(16)(X)
X = Activation("relu")(X)
output = tf.keras.layers.Dense(1)(X)
modelNNOM = tf.keras.Model(input, output);

# Create a model - Sequential Model
model = tf.keras.Sequential()
model.add(layers.Dense(16, activation='relu',name="layer1", input_shape=(1
,)))
model.add(layers.Dense(16, activation='relu',name="layer2"))
model.add(layers.Dense(1))
```

Code snippet 5 Sine wave Functional vs Sequential model

Functional Model produces same results as sequential. Only the activations are now considered as separate layers. Where as in sequential activation functions are part of layers. Total parameters remain same 321 as of sequential model.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_26 (InputLayer)        [(None, 1)]               0
_________________________________________________________________
dense_44 (Dense)             (None, 16)                32
_________________________________________________________________
activation_13 (Activation)   (None, 16)                0
_________________________________________________________________
dense_45 (Dense)             (None, 16)                272
_________________________________________________________________
activation_14 (Activation)   (None, 16)                0
_________________________________________________________________
dense_46 (Dense)             (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
```

Figure 11 NNOM model layer parameters

Training the model and running predictions on the same data set which was previously used for construction sequential model yields a perfect match. In this case the predictions seem more averaged and following a continuous sine wave with few bends.

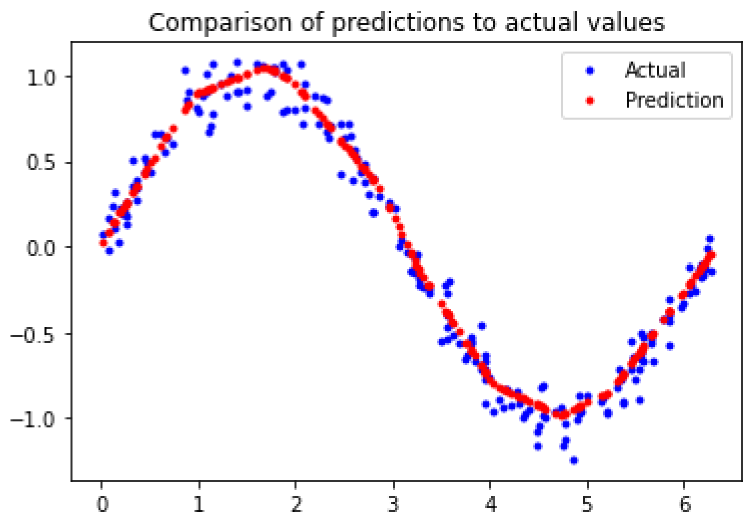


Figure 12 NNOM model predictions

NNOM model training results and layers max min output is below. Test loss and mean absolute error for model stands at 0.0816. Layers input and output range is between

Table 1 NNOM max min layer input/output

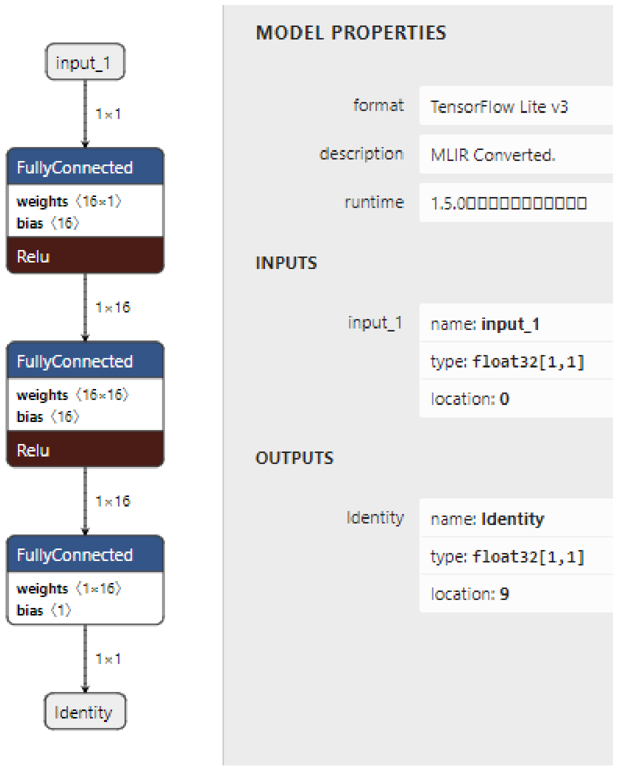
	Max	Min
Input	6.28	-0.0137
Second Layer	2.80	-3.44
Third Layer	1.89	-2.7
Output Layer	1.04	-0.98

```
7/7 - 0s - loss: 0.0816 - mean_absolute_error: 0.0816
Test loss: 0.08161722123622894
Top 1: 0.08161722123622894
input_26 Quantized method: max-min  Values max: 6.280983639904923 min:
0.013756599578923862 dec bit 4
dense_44 Quantized method: max-min  Values max: 2.803634 min: -3.4467075
dec bit 5
activation_13 Quantized method: max-min  Values max: 2.803634 min: -
3.4467075 dec bit 5
dense_45 Quantized method: max-min  Values max: 1.8984458 min: -2.7345867
dec bit 5
activation_14 Quantized method: max-min  Values max: 1.8984458 min: -
2.7345867 dec bit 5
dense_46 Quantized method: max-min  Values max: 1.0484817 min: -0.98121214
dec bit 6
quantisation list {'input_26': [4, 0], 'dense_44': [5, 0],
'activation_13': [5, 0], 'dense_45': [5, 0], 'activation_14': [5, 0],
'dense_46': [6, 0]}
quantizing weights for layer dense_44
    tensor_dense_44_kernel_0 dec bit 7
    tensor_dense_44_bias_0 dec bit 7
quantizing weights for layer dense_45
    tensor_dense_45_kernel_0 dec bit 6
    tensor_dense_45_bias_0 dec bit 7
quantizing weights for layer dense_46
    tensor_dense_46_kernel_0 dec bit 6
    tensor_dense_46_bias_0 dec bit 7
```

Figure 13 NNOM layers output range & mean absolute error

Layers weights width is also printed by the NNOM. This is too important. Since the NNOM don't support fractional inputs. We can multiply the fractional number with width of weight and input the whole number. At output we must do the same divide the output by the weight width of the layer to convert back the number in fractional format.

Model file when fed to netron results in the same output. Input and output types are floats. Layers names are replayed by fully connected. Fully connected is another name for dense layers. So, no difference. Each and every thing is identical to previous sequential model.



MODEL PROPERTIES

format	TensorFlow Lite v3
description	MLIR Converted.
runtime	1.5.0□□□□□□□□□□□□

INPUTS

input_1	name: **input_1**
	type: `float32[1,1]`
	location: **0**

OUTPUTS

Identity	name: **Identity**
	type: `float32[1,1]`
	location: **9**

Figure 14 NNOM individual layers anatomy

Mean absolute error (MAE) for the both models after training is listed below. Both models were trained on the same dataset, same number of epochs and batch sizes are used for training and validation. Not much difference in MAE is seen.

Table 2 Mean absolute error sequential and functional model

Model	Mean absolute error during training
Functional	0.0895
Sequential	0.0838

TensorFlow – Project (c array)

After constructing and training the model in TensorFlow. It is translated to equivalent c code using TensorFlow lite for microcontroller translator. Translated c code file is then made part of the project.

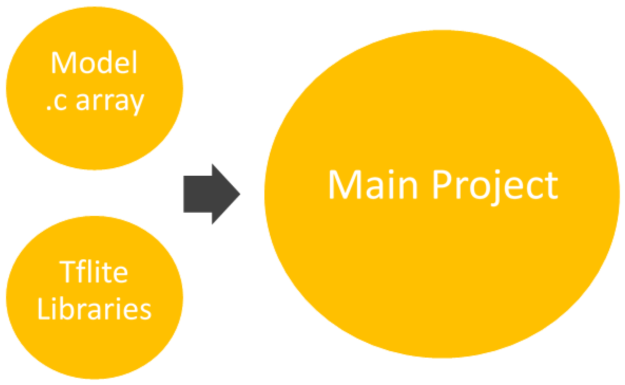


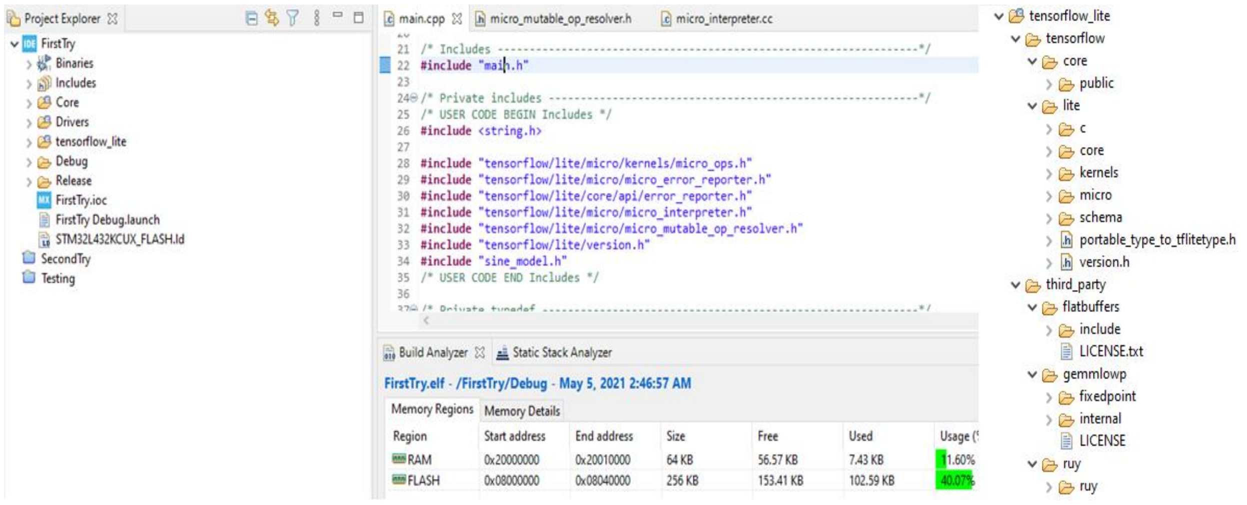
Figure 15 TFlite libraries with .c array

Work flow of project is shown above. Model c array and Tflite external libraries are made part of the project. Since we are using ST microcontroller we must also import the ST dependent libraries in the project.

I configured the STM32L432Kc using STM32CubeMX. Cube MX is a GUI package by STM which can be used to initialize the individual peripherals of STm32 microcontrollers. After setting up the micro peripherals you can generate initialized code for differed IDE's. Currently Cube MX produces HAL libraries for micro peripherals. HAL are the updated peripheral libraries for st microcontrollers.

I generated code for STM32Cube IDE and keil uvision IDE. To test both, since both uses different compilers. Tweaked GNU-GCC compiler.

TensorFlow project using STM32CubeIDE: In the project tree all the included folders can be seen. Tensorflow libraries are in tensorflow_lite folder. Tree is further expanded on the right side. Plenty of files are in package and all of them are in c++. So, our main application file must be a c++ source file. Project code comprised of both c and c++ sources. In below window main file extension is .cpp or c++ file.



```
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <string.h>

#include "tensorflow/lite/micro/kernels/micro_ops.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/core/api/error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/version.h"
#include "sine_model.h"
/* USER CODE END Includes */
```

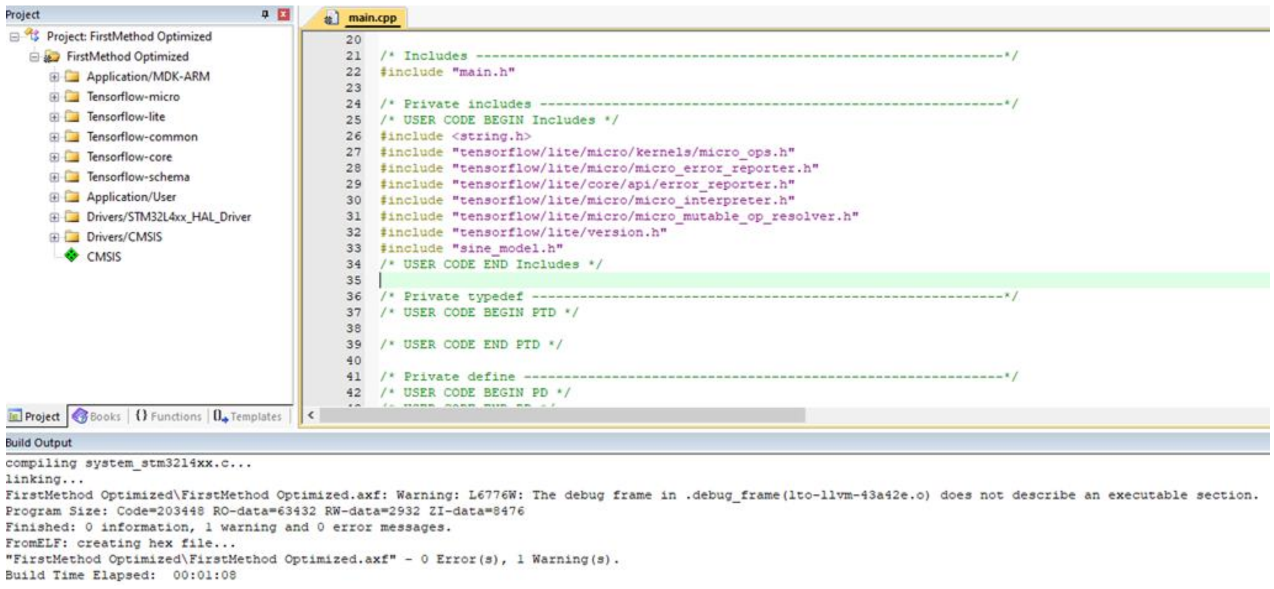
FirstTry.elf - /FirstTry/Debug - May 5, 2021 2:46:57 AM

Region	Start address	End address	Size	Free	Used	Usage (%
RAM	0x20000000	0x20010000	64 KB	56.57 KB	7.43 KB	11.60%
FLASH	0x08000000	0x08040000	256 KB	153.41 KB	102.59 KB	40.07%

Figure 16 TensorFlow project using STM32CubeIDE

Stm32cubeide uses GCC compiler with GNU arm embedded tool chain integrated in it. It's a free distribution.

TensorFlow project using keil uvision ide*:* With keil uvision IDE the libraries are same. Small difference is in the startup file, changed heap and stack size. The main difference is in compiler. ARM compiler 6 with ARMclang technology is used to compile the code. HAL and CMSIS libraries are same which were previously used in the cube ide. Project tree can be seen on the left side. Tensorflow folders are individually made part of the project. Keil doesn't support folder in a folder tree structure. Our main source file ends with .cpp extension. Which means our code is in c and c++.



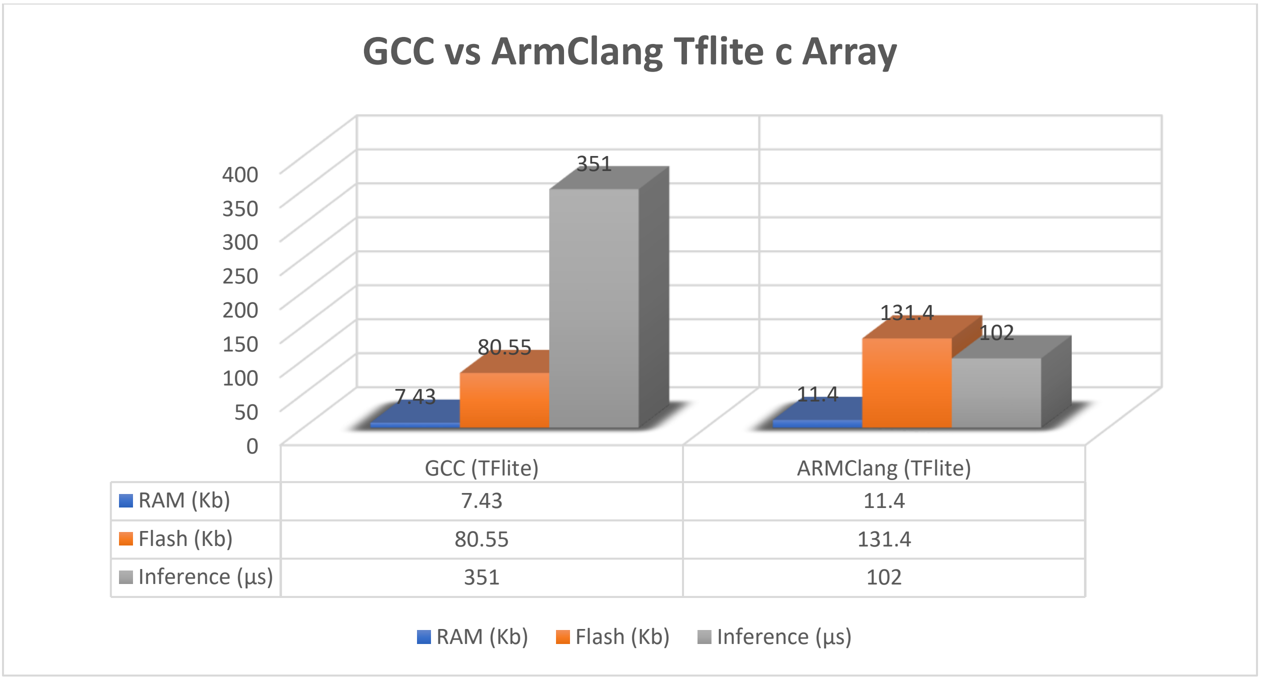
```
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <string.h>
#include "tensorflow/lite/micro/kernels/micro_ops.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/core/api/error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/version.h"
#include "sine_model.h"
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
```

```
compiling system_stm32l4xx.c...
linking...
FirstMethod Optimized\FirstMethod Optimized.axf: Warning: L6776W: The debug frame in .debug_frame(lto-llvm-43a42e.o) does not describe an executable section.
Program Size: Code=203448 RO-data=63432 RW-data=2932 ZI-data=8476
Finished: 0 information, 1 warning and 0 error messages.
FromELF: creating hex file...
"FirstMethod Optimized\FirstMethod Optimized.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:01:08
```

Figure 17 TensorFlow project using keil uvision ide

Arm clang is a propriety and a paid subscription is required for full code compilation.

Source code for both compliers were optimized at the maximum code and performance level which is given as -o3. For keil clang, link time optimization was also enabled.



	GCC (TFlite)	ARMClang (TFlite)
RAM (Kb)	7.43	11.4
Flash (Kb)	80.55	131.4
Inference (µs)	351	102

Chart 1 GCC vs ARM Clang -TFlite

ArmClang flash and ram occupancy is way higher than open source GCC. But the inference speed of ARM Clang is higher than GCC.

Table 3 GCC to ARM Clang - TFlite

TFlite - GCC to ARMClang	
Flash	+63%
RAM	+53%
Inference	-244%

TensorFlow – project (.tflite flat buffer file)

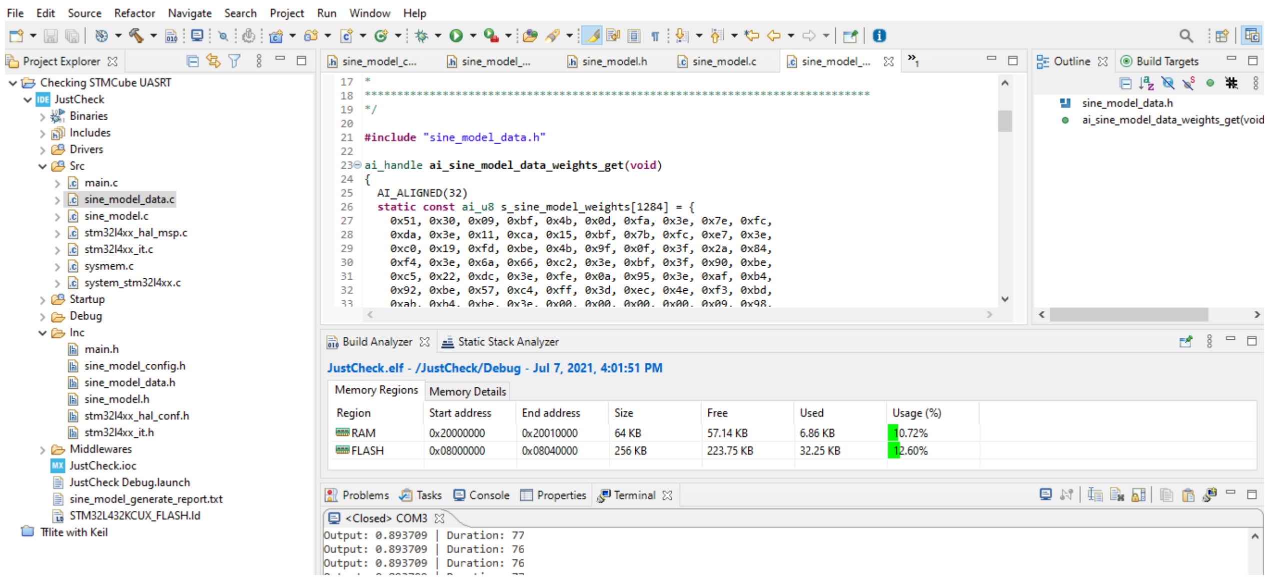
The c array generated in the above two methods is dependent on tflite file. This file is a cluster of flat buffers scalers and vectors representing our trained model. We generate c code for TFlite for microcontroller project using this file.

ST microelectronics offers their own model translator under the name of STM32CubeMX.AI or AI package for ST microelectronics. Translator takes input a trained model file such as tflite or .h5 model file of keras. It then translates the model and outputs equivalent c files for the target STM32 microcontroller.

In our case, first I selected the target microcontroller which is STM32l432kc. Next open the STM32CubeMX to initialize the peripherals. Then opened the STM32Cube AI and provided the tflite file as input. Compression and optimization can be performed on the inputted model file using cube AI package. Next the model is translated and pre-initialized code for target MCU is generated. In this case code is generated for both cube ide and keil uvision as well.

STM32CubeMX.AI with .tflite model file: In the left pane of the above window project tree can be seen. Model files translated by STM32Cube AI are present in src (sine_model_data.c and sine_model.c) and inc (data, config and model header files) folder. Neurons their weights and biases are defined in these files. In the middleware's folder STM32 AI libraries are residing. Together with the model files in src folder these libraries reconstruct the model in micro.

Main file extension in this case is .c, which means our source code is in c language. All project libraries are in c.



```
*
******************************************************************************
*/

#include "sine_model_data.h"

ai_handle ai_sine_model_data_weights_get(void)
{
  AI_ALIGNED(32)
  static const ai_u8 s_sine_model_weights[1284] = {
    0x51, 0x30, 0x09, 0xbf, 0x4b, 0x0d, 0xfa, 0x3e, 0x7e, 0xfc,
    0xda, 0x3e, 0x11, 0xca, 0x15, 0xbf, 0x7b, 0xfc, 0xe7, 0x3e,
    0xc0, 0x19, 0xfd, 0xbe, 0x4b, 0x9f, 0x0f, 0x3f, 0x2a, 0x84,
    0xf4, 0x3e, 0x6a, 0x66, 0xc2, 0x3e, 0xbf, 0x3f, 0x90, 0xbe,
    0xc5, 0x22, 0xdc, 0x3e, 0xfe, 0x0a, 0x95, 0x3e, 0xaf, 0xb4,
    0x92, 0xbe, 0x57, 0xc4, 0xff, 0x3d, 0xec, 0x4e, 0xf3, 0xbd,
    0xab, 0xb4, 0xbe, 0x3e, 0x00, 0x00, 0x00, 0x00, 0x09, 0x98,
```

JustCheck.elf - /JustCheck/Debug - Jul 7, 2021, 4:01:51 PM

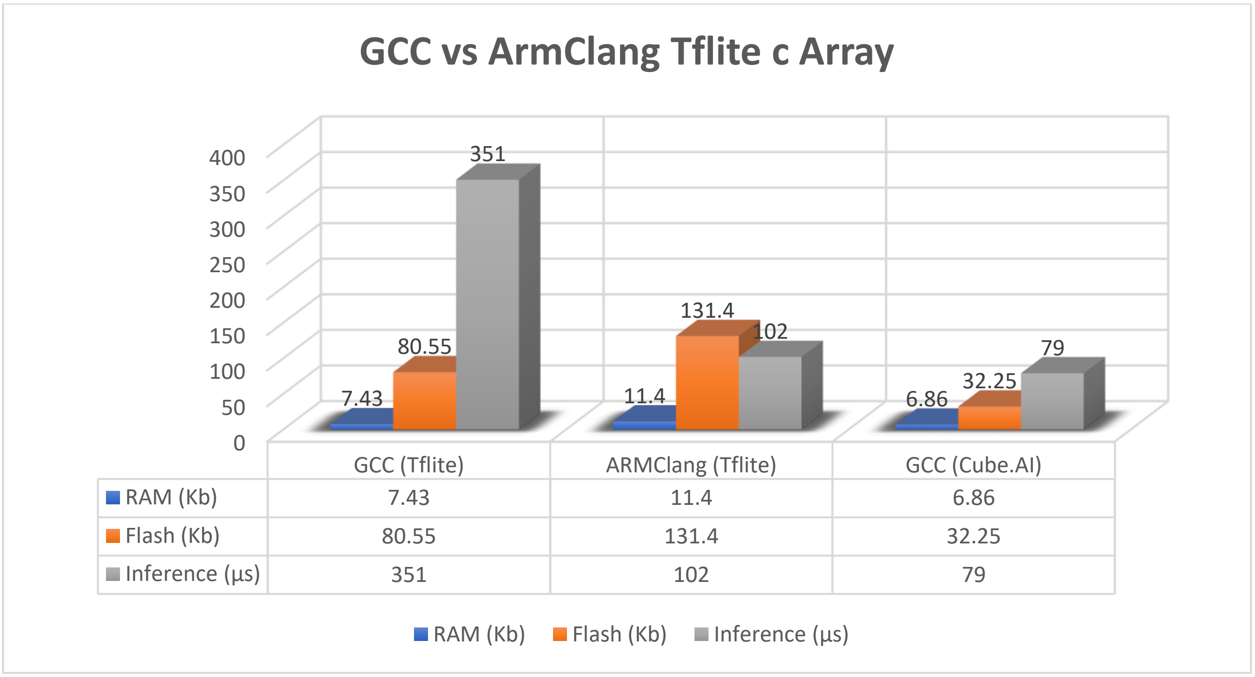
Region	Start address	End address	Size	Free	Used	Usage (%)
RAM	0x20000000	0x20010000	64 KB	57.14 KB	6.86 KB	10.72%
FLASH	0x08000000	0x08040000	256 KB	223.75 KB	32.25 KB	12.60%

```
Output: 0.893709 | Duration: 77
Output: 0.893709 | Duration: 76
Output: 0.893709 | Duration: 76
```

Figure 18 STM32CubeMX.AI with .tflite model file

After compiling, the used Flash and RAM occupancy can be seen in the memory region window. It can be seen that 6.86Kb of RAM is used and 32.35Kb of flash is used. I am outputting inference result as serial UART output.

Stm32cubeide offers a terminal window. we can establish a serial connection to this window by specifying communication parameters. I set it up for 115200 bits/s, parity none and 1 stop bit. Inference results are outputted in this window. Inference time for prediction is recorded as 79 micro seconds.



	GCC (Tflite)	ARMClang (Tflite)	GCC (Cube.AI)
RAM (Kb)	7.43	11.4	6.86
Flash (Kb)	80.55	131.4	32.25
Inference (µs)	351	102	79

Chart 2 TFlite to Cube AI - ARM Clang

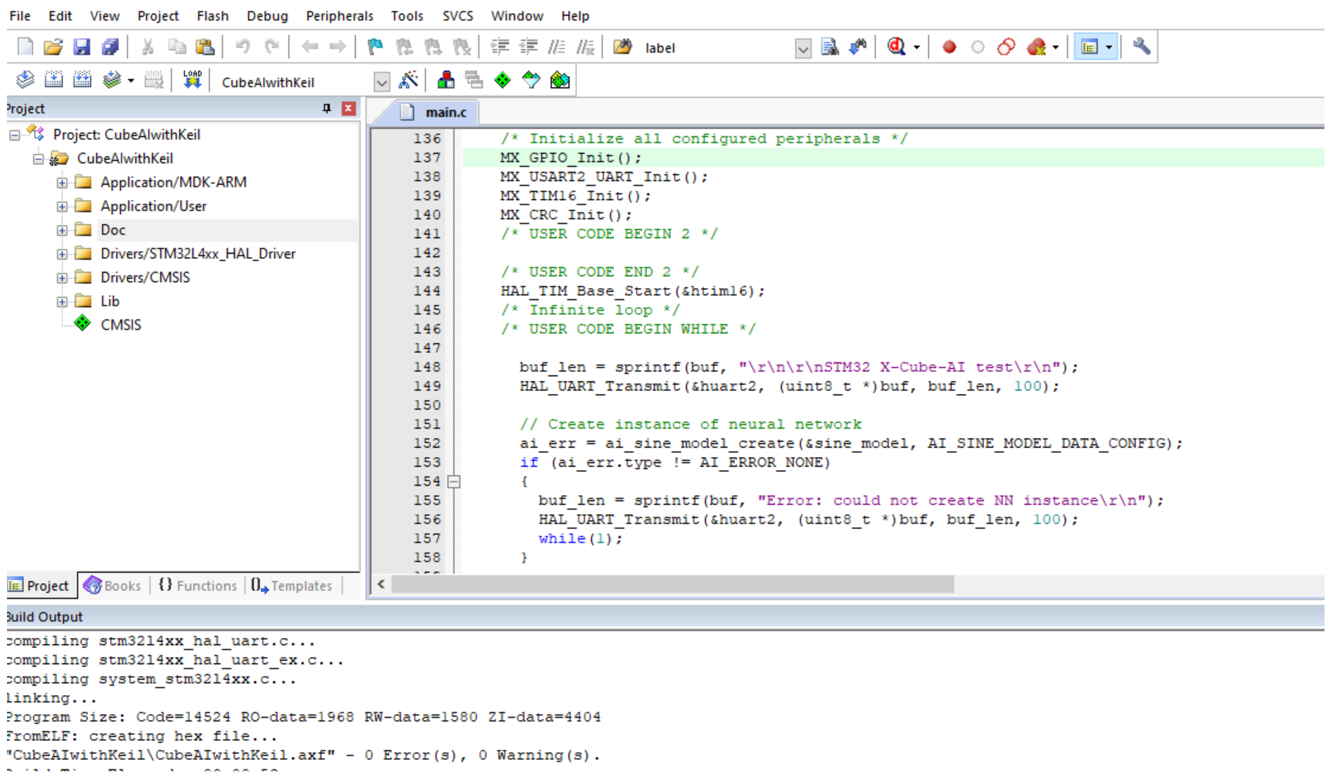
Inference time and flash occupancy is significantly reduced using Cube AI translator with GCC compiler.

Table 4 TFlite to Cube AI - GCC

GCC - TFlite to Cube AI	
Flash	-149.2%
RAM	-8.3%
Inference	-344.3%

Cube AI code compiled with GCC reduces ram and flash size occupancy compared with GCC-Tflite.

STM32CubeAI .tflite model file with keil IDE: In this case Cube MX is used to generate peripheral code. Cube AI translates model. Translated model code is same like the one generated in Cube IDE case because Tflite file is same. Project tree can be seen on the left side in the main window. AI libraries are in lib folder. Main source file is in c. we are in same situation like above one. Our all libraries are in c language.

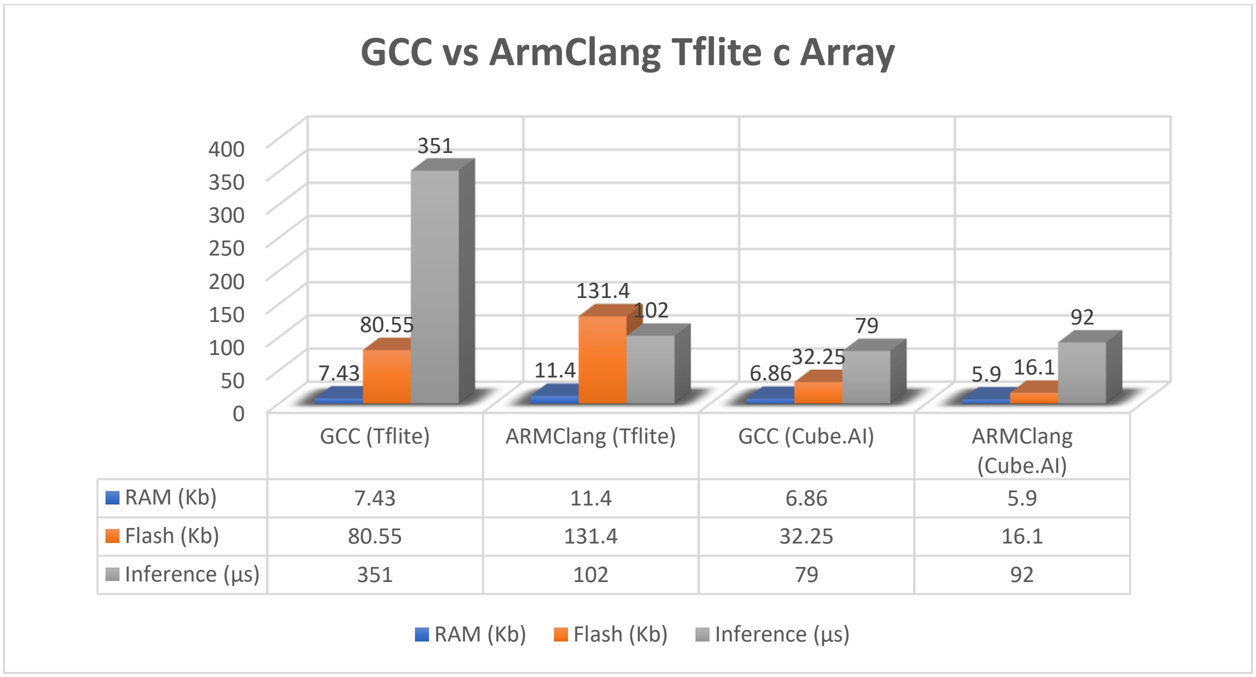


```c
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM16_Init();
  MX_CRC_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */
  HAL_TIM_Base_Start(&htim16);
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */

    buf_len = sprintf(buf, "\r\n\r\nSTM32 X-Cube-AI test\r\n");
    HAL_UART_Transmit(&huart2, (uint8_t *)buf, buf_len, 100);

    // Create instance of neural network
    ai_err = ai_sine_model_create(&sine_model, AI_SINE_MODEL_DATA_CONFIG);
    if (ai_err.type != AI_ERROR_NONE)
    {
      buf_len = sprintf(buf, "Error: could not create NN instance\r\n");
      HAL_UART_Transmit(&huart2, (uint8_t *)buf, buf_len, 100);
      while(1);
    }
```

```
compiling stm32l4xx_hal_uart.c...
compiling stm32l4xx_hal_uart_ex.c...
compiling system_stm32l4xx.c...
linking...
Program Size: Code=14524 RO-data=1968 RW-data=1580 ZI-data=4404
FromELF: creating hex file...
"CubeAIwithKeil\CubeAIwithKeil.axf" - 0 Error(s), 0 Warning(s).
```

Figure 19 STM32 Cube AI .tflite model file with keil IDE

Project is compiled successfully and results can be seen in the build output window above. Building time is slightly higher than the GCC. The use of LLVM technique in ARM Clang takes longer to build the project files.



	GCC (Tflite)	ARMClang (Tflite)	GCC (Cube.AI)	ARMClang (Cube.AI)
■ RAM (Kb)	7.43	11.4	6.86	5.9
■ Flash (Kb)	80.55	131.4	32.25	16.1
■ Inference (µs)	351	102	79	92

Chart 3 GCC vs ARM Clang - Tflite vs Cube AI

First comparing the ARM Clang switching from Tflite to Cube AI we see reduction in all parameters (flash, ram and inference time). Next switching from GCC to ARM Clang we see reduction in flash and ram but the inference time is increased.

Table 5 TFlite to Cube AI - ARM Clang

ARM Clang – Tflite to Cube AI	
Flash	-716.3%
RAM	-93.2%
Inference	-10.82%

Table 6 GCC to ARM Clang - Cube AI

Cube AI - GCC to ARM Clang	
Flash	-100.3%
RAM	-16.27%
Inference	+16.45%

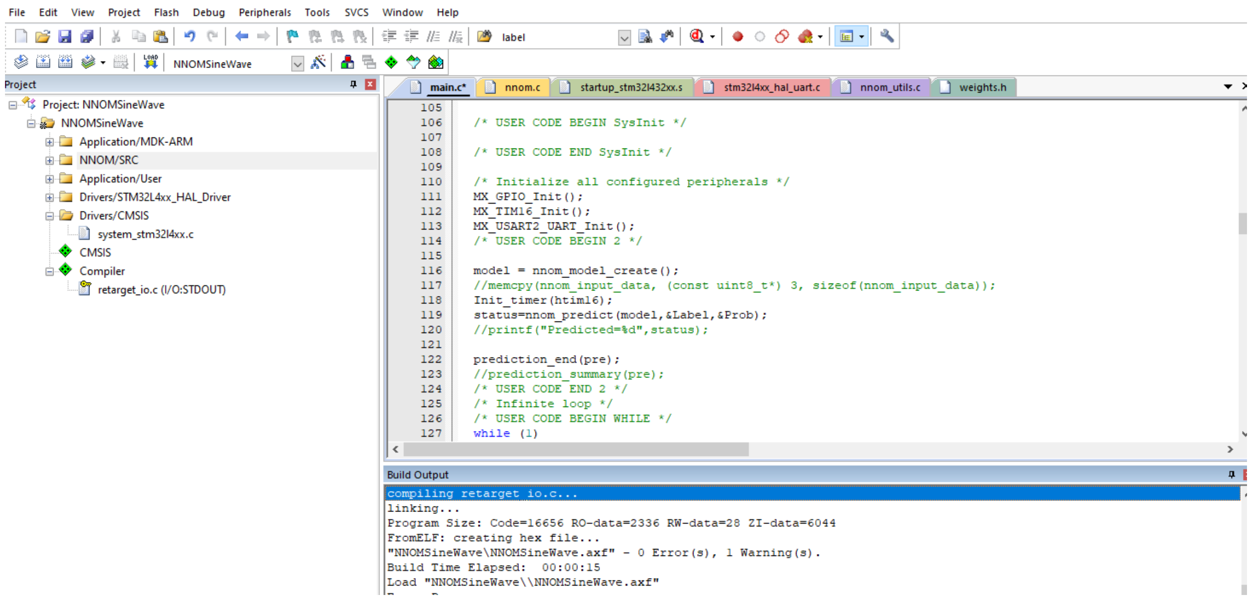
NNOM - Project (Keras)

NNOM for sine wave model. NNOM is a light weight open source project on GitHub which perform inference using a model constructed and trained in Keras. Unlike TFlite, NNOM model must be in sequential/layered architecture. Sine model which was previously used for TFlite and Cube AI is transformed into layered architecture. Process shown in the introduction chapter.

NNOM provides python dependencies and translator which can translate Keras AI model into equivalent NNOM c code. The concept is same like other solutions. Only the translator is changed and translated code is equivalent to NNOM supported architecture.

On target side NNOM provides a package of c libraries which can be made part of the project for inference. The major drawback of NNOM is it doesn’t support fractional numbers as input.

Project peripherals were initialized with stm32CubeMX and project files were imported for keil uvision ide. So. ARM Clang compiler is used to compile code for NNOM.

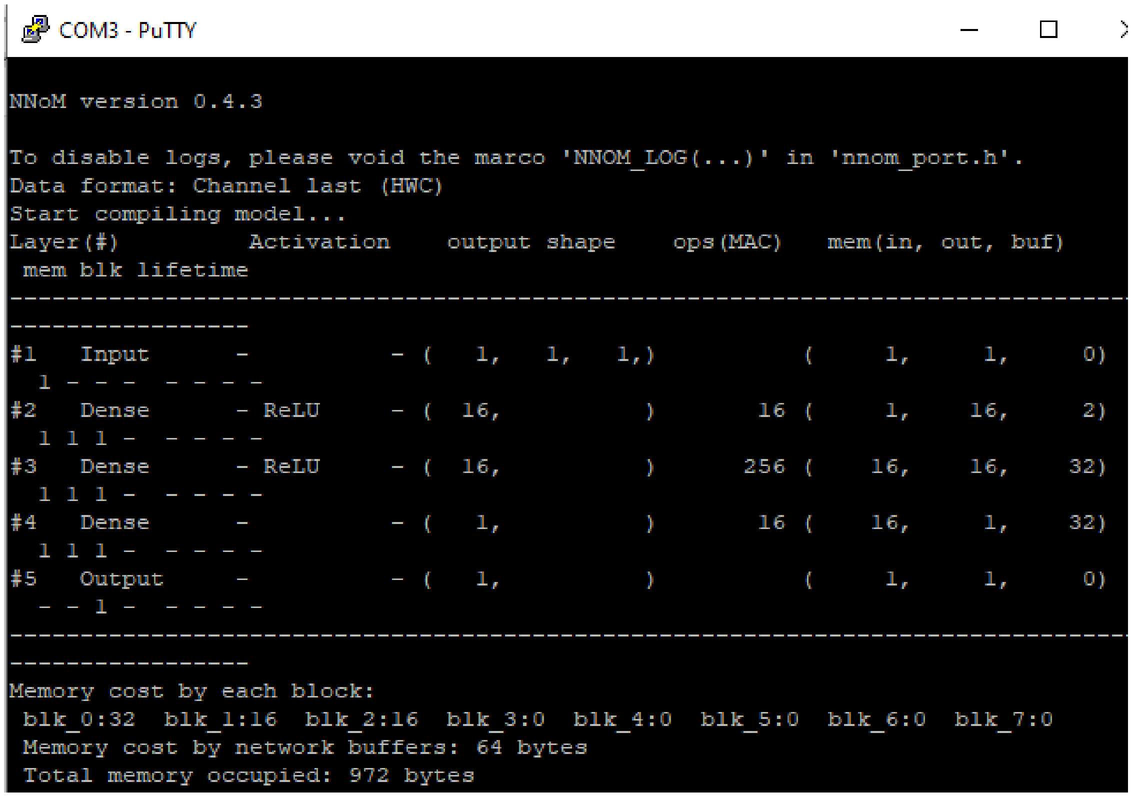


```c

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM16_Init();
  MX_USART2_UART_Init();
  /* USER CODE BEGIN 2 */

  model = nnom_model_create();
  //memcpy(nnom_input_data, (const uint8_t*) 3, sizeof(nnom_input_data));
  Init_timer(htim16);
  status=nnom_predict(model,&Label,&Prob);
  //printf("Predicted=%d",status);

  prediction_end(pre);
  //prediction_summary(pre);
  /* USER CODE END 2 */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
```

```
compiling retarget_io.c...
linking...
Program Size: Code=16656 RO-data=2336 RW-data=28 ZI-data=6044
FromELF: creating hex file...
"NNOMSineWave\NNOMSineWave.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:00:15
Load "NNOMSineWave\\NNOMSineWave.axf"
```

Figure 20 NNOM with ARM Clang

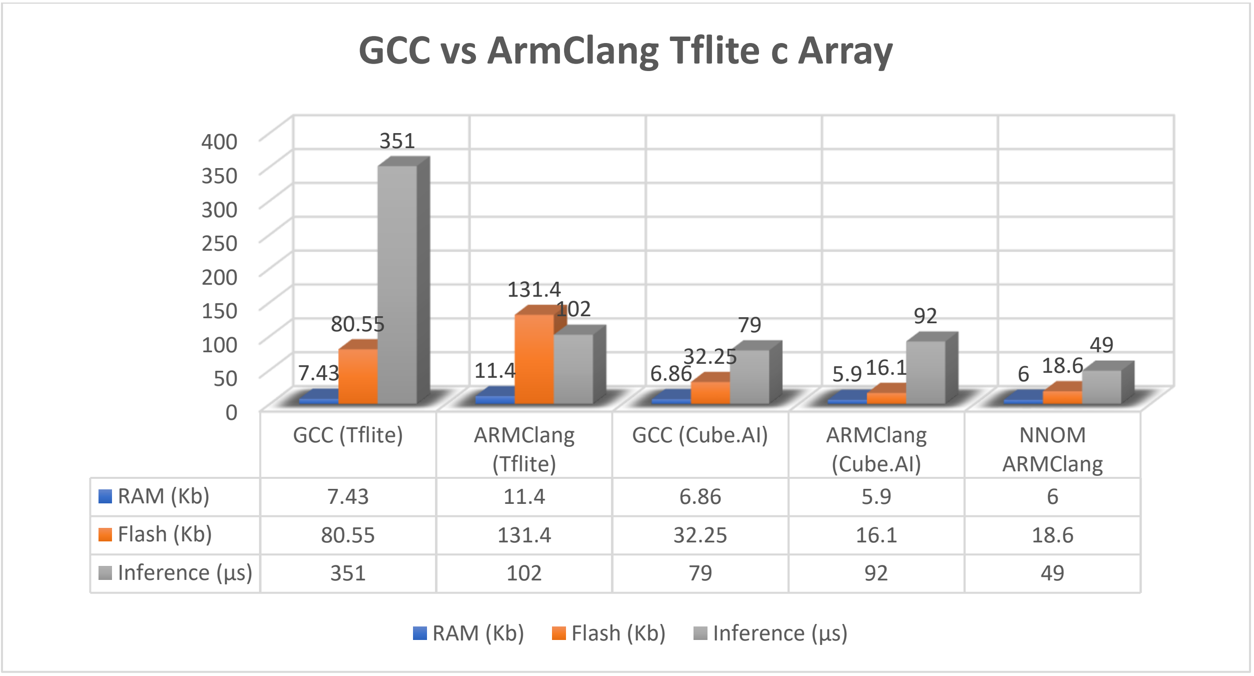
In the above window on left pane project file tree is present. NNOM libraries are in NNOM/SRC folder. Weights header file in project window is the c code generated by the NNOM translator. Rest HAL and CMSIS libraries are included default by Cube MX when peripherals were initialized. Compiling the code is error free, build details and resources used are listed in the build output window.



```
NNoM version 0.4.3

To disable logs, please void the marco 'NNOM_LOG(...)' in 'nnom_port.h'.
Data format: Channel last (HWC)
Start compiling model...
Layer(#)         Activation    output shape    ops(MAC)   mem(in, out, buf)
 mem blk lifetime
-------------------------------------------------------------------------------
-----------------
#1   Input      -          - (   1,   1,   1,)          (     1,      1,      0)
  1 - - -  - - - - -
#2   Dense      - ReLU     - (  16,          )       16 (     1,     16,      2)
  1 1 1 -  - - - - -
#3   Dense      - ReLU     - (  16,          )      256 (    16,     16,     32)
  1 1 1 -  - - - - -
#4   Dense      -          - (   1,          )       16 (    16,      1,     32)
  1 1 1 -  - - - - -
#5   Output     -          - (   1,          )          (     1,      1,      0)
  - - 1 -  - - - - -
-------------------------------------------------------------------------------
-----------------
Memory cost by each block:
 blk_0:32  blk_1:16  blk_2:16  blk_3:0  blk_4:0  blk_5:0  blk_6:0  blk_7:0
 Memory cost by network buffers: 64 bytes
 Total memory occupied: 972 bytes
```

Figure 21 NNOM model anatomy UART view

Verifying the model if translated correctly is possible with microcontroller. We can output the model details on USART port. Layers shape's, input and outputs, tensors are same. Memory buffers are saved weights and biases. Total parameters are also same 321.



	GCC (Tflite)	ARMClang (Tflite)	GCC (Cube.AI)	ARMClang (Cube.AI)	NNOM ARMClang
🟦 RAM (Kb)	7.43	11.4	6.86	5.9	6
🟧 Flash (Kb)	80.55	131.4	32.25	16.1	18.6
⬜ Inference (µs)	351	102	79	92	49

Chart 4 NNOM vs GCC vs ARM Clang

Comparing NNOM with all the other solutions clearly shows it's the best optimum solution. Inference time is way better than all the other solutions. Regarding memory its slightly higher than the ARM Clang Cube AI solution.

Output Deviation: So far, we compared only the memory and inference time. Another and most important parameter is output. Does the input produce the required output? And how much time each input takes for inference?

Deviation highly depends on the model and its training. Model must be trained to achieve lowest MAE. Model can be checked for MAE. This error is expected in the inference results.

I took 6 random numbers (0.7, 1.6, 2, 3.4, 4.9, 6.2) from input range and their output is compared with original sine function output. Individual and average deviation from original output is calculated for each solution.

Recall

Table 7 Recall MAE

Model	Mean absolute error during training
Functional	0.0895
Sequential	0.0838

	Original	GCC(TFlite)			ARM Clang(TFlite)			NNOM (ARM Clang)		
Input	Output	Output	Deviation	Inference µ	Output	Deviation	Inference µ	Output	Deviation	Inference
0.7	0.644	0.654	1.55%	351	0.654	1.55%	102	0.683	6.06%	49
1.6	0.999	1.023	2.40%	351	1.023	2.40%	102	1.041	4.20%	49
2	0.909	0.913	0.44%	351	0.913	0.44%	102	0.925	1.76%	49
3.4	-0.255	-0.247	-3.14%	351	-0.247	-3.14%	102	-0.206	-19.22%	49
4.9	-0.982	-0.985	0.31%	351	-0.985	0.31%	102	-0.968	-1.43%	49
6.2	-0.083	-0.104	25.30%	351	-0.104	25.30%	102	-0.073	-12.05%	49
Av Deviation			4.48%			4.48%			-3.44%	
		GCC(Cube AI)			ARM Clang(Cube AI)					
		Output	Deviation	Inference µ	Output	Deviation	Inference µs			
		0.607	-5.75%	79	0.607	-5.75%	92			
		0.997	-0.20%	79	0.997	-0.20%	92			
		0.893	-1.76%	79	0.893	-1.76%	92			
		-0.273	7.06%	79	-0.273	7.06%	92			
		-1	1.83%	79	-1	1.83%	92			
		-0.234	181.90%	79	-0.234	181.90%	92			
Av Deviation			30.51%			30.51%				

Analyzing the above results, we see that inference time is constant for every input in each individual solution. If MAE is taken as 0.08 for both models. We see that most of the outputs are in range of MAE, few are exceeding the it. NNOM MAE is far less than others. Sin(6.2) is deviation is worst in all the solutions. In Cube AI case it produces output which deviates 181% from original output.

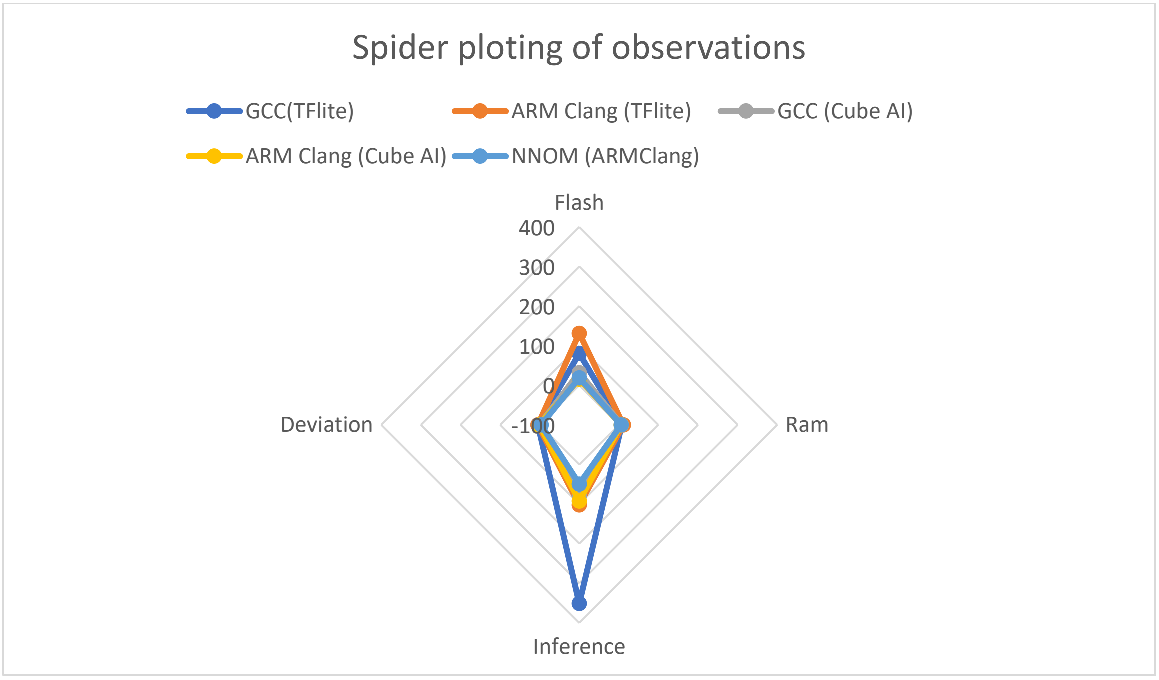


Chart 5 GCC vs ARM Clang vs NNOM - Spider view

Spider graph is the best representation of the observed results. We see that inference time for GCC Tflite is approaching its limits whereas on flash side it is less than ARM Clang Tflite. Less than GCC Tflite area is covered by ARM clang Tflite. So, this solution is better compared to previous one. Compared to the above two solutions the GCC and Arm Clang Cube AI are covering less area. So, Cube AI with Clang and GCC is better than Tflite. We can decide between the two depending on the memory and inference precedence, what we want in our project.

Minimum area occupied is by NNOM which means that NNOM is the best solution. Major draw backs of NNOM project is it does not support fractional number as input. You have to multiply the input fractional number with the width of the input layer to convert it in to whole number. At output we must divide the number with the width of the output layer to convert it back to fractional number.

Compile time injection faults to test robustness

Compile time injection faults are introduced in the source code to test the robustness of AI deployment on microcontroller. Source code of the model will be flipped changed to test the behavior of the model under stress.

I decided to use a bigger model for compile time injection faults testing. Sine wave model can also be tested. But a model consisting of images is better. We can test if images can be predicted by microcontrollers. Off course our model size will be bigger. On the other hand, we will be able to put stress on micro to test its robustness.

MNIST Dataset: MNIST is a collection of hand written digits images. Total images in training set are 60,000. Testing set has total of 10,000 images. Images size is 28x28. We must convert this array to 4 dimensions in order to use it with keras or tensorflow.

We learnt from the sine wave model that NNOM solution is perfect in all dimensions. So, I decided to use NNOM libraries with ARM Clang compiler to test MNIST dataset.

Individual pixels of images range between 0 to 255. Which is quite bigger in memory cost. We can quantize our images. Individual pixel is divided by 255 to bring down the pixel range to 0-1. We can train our model on this range. This can not only reduce the weights and bias but also saves us a lot of memory.

Now the usual NNOM operation is performed and model c files are generated. The final model in c format is below

```c
static nnom_model_t* nnom_model_create(void)
{
    static nnom_model_t model;
    nnom_layer_t* layer[16];

    check_model_version(NNOM_MODEL_VERSION);
    new_model(&model);

    layer[0] = input_s(&input_1_config);
    layer[1] = model.hook(conv2d_s(&conv2d_config), layer[0]);
    layer[2] = model.active(act_relu(), layer[1]);
    layer[3] = model.hook(maxpool_s(&max_pooling2d_config), layer[2]);
    layer[4] = model.hook(conv2d_s(&conv2d_1_config), layer[3]);
    layer[5] = model.active(act_relu(), layer[4]);
    layer[6] = model.hook(maxpool_s(&max_pooling2d_1_config), layer[5]);
    layer[7] = model.hook(conv2d_s(&conv2d_2_config), layer[6]);
    layer[8] = model.active(act_relu(), layer[7]);
    layer[9] = model.hook(maxpool_s(&max_pooling2d_2_config), layer[8]);
    layer[10] = model.hook(flatten_s(&flatten_config), layer[9]);
    layer[11] = model.hook(dense_s(&dense_config), layer[10]);
    layer[12] = model.active(act_relu(), layer[11]);
    layer[13] = model.hook(dense_s(&dense_1_config), layer[12]);
    layer[14] = model.hook(softmax_s(&softmax_config), layer[13]);
    layer[15] = model.hook(output_s(&output_config), layer[14]);
    model_compile(&model, layer[0], layer[15]);
    return &model;
}
```

Code snippet 6 MNIST model in c

Model consists of three convolution layers, each layer used RELU for activation. Followed by two dense or fully connected layers. Final output goes through the softmax layer. So, in total 6 layers. But since NNOM works on layered architecture so all the individual functions are also considered layers.

To perform inference ten random images were selected from the database and their pixel array values were saved in header file. These ten images will be used to test the MNIST dataset in microcontroller core. They will be inputted to model for inference. Size of the images are 28x28 which translates in to an array of 784 integer values. Furthermore, NNOM input layer weights width is 7 bits so our images pixel values will be in range from 0 to 127.

```
#define IMG8 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,
#define IMG8_LABLE 6

#define IMG9 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0,
0, 0, 0,
#define IMG9_LABLE 9

#define TOTAL_IMAGE 10

static const int8_t img[10][784] =
{IMG0,IMG1,IMG2,IMG3,IMG4,IMG5,IMG6,IMG7,IMG8,IMG9};
```

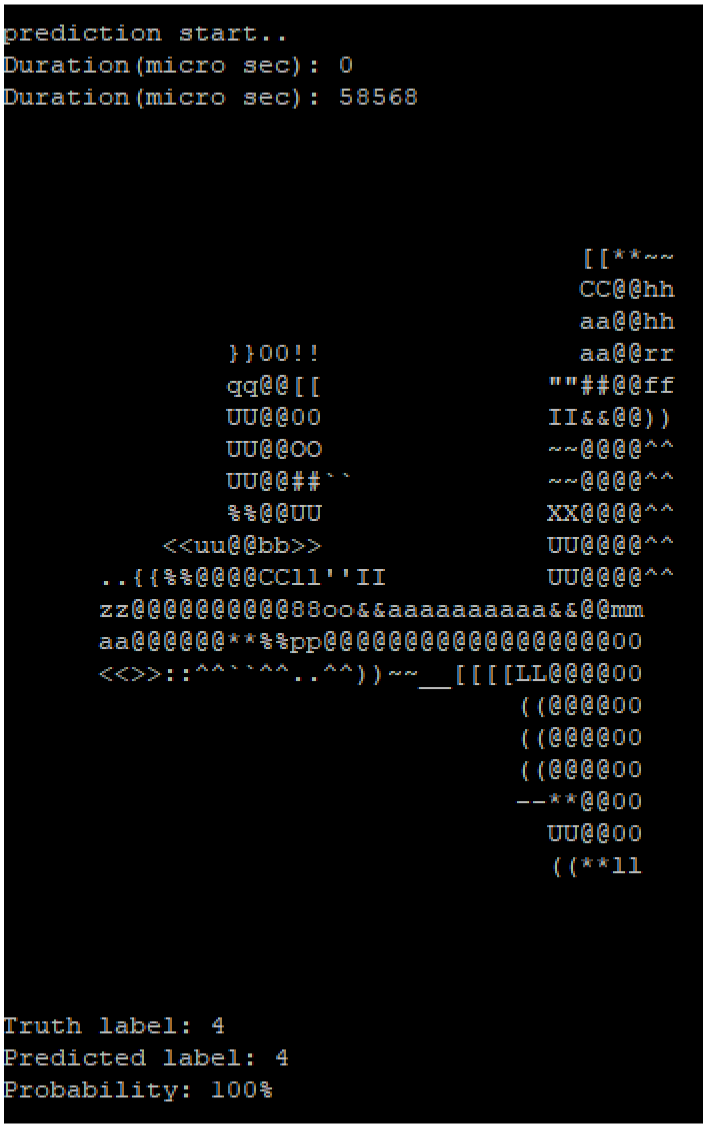
Code snippet 7 MNIST images arrays

Compiling the project and looking in the build process, all the libraries are successfully compiled. Flash and Ram occupancy is listed in the below table.

Table 8 MNIST flash and ram occupancy

Parameter	Memory kilo Byte
Flash	145.772 Kb
RAM	26.824 Kb

To run inference input image array is feed to the model. Output is printed on serial window. USART at 9600 bits/s is initialized. Putty is used to make a serial connection and print the USART output by microcontroller.



```
prediction start..
Duration(micro sec): 0
Duration(micro sec): 58568




                                    [[**~~
                                    CC@@hh
                                    aa@@hh
              }}OO!!                aa@@rr
              qq@@[[              ""##@@ff
              UU@@OO              II&&@@))
              UU@@OO              ~~@@@@^^
              UU@@##``            ~~@@@@^^
              %%@@UU              XX@@@@^^
          <<uu@@bb>>              UU@@@@^^
      ..{{%%@@@@CCll''II          UU@@@@^^
      zz@@@@@@@@@@88oo&&aaaaaaaaaa&&@@mm
      aa@@@@@@**%%pp@@@@@@@@@@@@@@@@@@OO
      <<>>::^^``^^..^^))~~__[[[[LL@@@@OO
                                ((@@@@OO
                                ((@@@@OO
                                ((@@@@OO
                                --**@@OO
                                  UU@@OO
                                  ((**ll




Truth label: 4
Predicted label: 4
Probability: 100%
```

Figure 22 MNIST inference results

It is concluded that all the 10 images are successfully identified by the model in microcontroller.

Average inference time recorded

Inference time	**5.8568 milli seconds**

- Significant amount of bit flips in weights and biases results in output totally inaccurate. But still there is an output. Unless the width of weight/bias arrays are manipulated.
- No significant change in prediction is found after flipping few image array bits.
- Prediction is altered by introducing numeric values at array positions which are previously standing at 0.
- Compiler automatically corrects the input size of a single element of image array during compiling if the input value size increases the datatype.

Conclusion

It’s obvious from the results that the c++ libraries use in the microcontroller domain is not providing any edge when compared to their c counterparts. C++ produces binaries which are enormous in size. A single advantage (compiler comparison GCC vs ARM Clang) is seen using c++ which is the reduction in inference time but still way higher compared to c. Open source GCC when compared with ARM Clang takes much time for inference. ARM Clang uses LLVM (low level virtual machine) for object and binaries creation which with the help of ARM compiler reduces the time for inference.

Open source NNOM and freeware STM32Cube AI have libraries in c and translates model in to equivalent c code. Both are better than google support for AI on microcontrollers (c++ TFlite). NNOM which supports functional model is found better than all. No difference is found in final model between functional and sequential but the performance of NNOM is better. This performance increase is due to NNOM translator and core inference engine.

Compile time faults injection in model and input data on MNIST dataset suggests that in order to alter results a big portion of bits must be flipped. Especially the pixels with zero value if flipped to 1, alters the result. Weights and bias also alter the result on the same condition of drastic change in values. If a bit flip increases the size of the individual array element compiler automatically adjusts the size to datatype and generates just a warning.

References

[1] F. Sakr, F. Bellotti, R. Berta, and A. De Gloria, “Machine Learning on Mainstream Microcontrollers,” *Sensors*, vol. 20, no. 9, Art. no. 9, Jan. 2020, doi: 10.3390/s20092638.

[2] “Wolfe - 2005 - How compilers and tools differ for embedded system.pdf.” Accessed: Jul. 15, 2021. [Online]. Available: https://my.eng.utah.edu/~cs5785/reading/pgi_article_cases.pdf

[3] P.-E. Novac, G. Boukli Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, “Quantization and Deployment of Deep Neural Networks on Microcontrollers,” *Sensors*, vol. 21, no. 9, Art. no. 9, Jan. 2021, doi: 10.3390/s21092984.

[4] “What is Renesas’ e-AI Solution? | Renesas.” https://www.renesas.com/us/en/application/technologies/e-ai/about-e-ai (accessed Jul. 15, 2021).

[5] “x-cube-ai.pdf.” Accessed: Jul. 15, 2021. [Online]. Available: https://www.stmicroelectronics.com.cn/resource/en/data_brief/x-cube-ai.pdf

[6] “ai-platform-solution-product-brief.pdf.” Accessed: Jul. 15, 2021. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/ai/ai-platform-solution-product-brief.pdf