Lab 5: Practical Evaluation of Machine Learning Models

Learning outcome: Optimize a neural network model and evaluate memory usage, latency, and energy consumption

Lab overview

In this lab, we will optimize a neural network model using TensorFlow Lite and evaluate the model with the evaluation board in terms of latency, memory and energy using STM32CubeAI.

First, you will learn how to measure the performance of neural network models. Second, you will learn how to optimize a neural network model in order to improve the latency and reduce energy consumption. Finally, you will implement a deep learning model on your own. You can select any dataset which may be suitable for ARM Cortex-M processors. Try to optimize your model as best as possible.

Requirements

Software functions

Python Packages

You will be using tensorflow, matplotlib, numpy and pillow. Install the required packages using Anaconda.

Main API Functions

Tensorflow

`tf.lite.TFLiteConverter.convert()`
Converts a TensorFlow model into TensorFlow Lite model

`tf.lite.Interpreter.get_input_details()`
Gets model input tensor details (name, index, shape, dtype, quantization, etc.)

`tf.lite.Interpreter.get_output_details()`
Gets model output tensor details (name, index, shape, dtype, quantization, etc.)

Optimizing Model with Quantization

Model Loading

First, open Jupyter Notebook through Anaconda Prompt.

```
> jupyter notebook
```

Open 'lab5.ipynb' on the notebook. Execute the first three code blocks to import packages and load the dataset. These code blocks are almost the same as those of Lab 4.

Next, the following code block will load the model trained with CIFAR-10. Although we will use the provided model, but you can change the model path to load your own model.

```
# Load keras model
path_models = "./Data/models/"
path_keras_model = path_models+"custom_cifar10_model.h5"

model = tf.keras.models.load_model(path_keras_model)

# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

Quantization

Next, we are going to apply quantization to the model. Quantization is an optimization technique, which changes the degree of precision by reducing the number of bits for representing data. For example, rather than using 32-bit floating point numbers, we can make the model to use 8-bit integer numbers for calculation. Through quantization, we can reduce the amount of memory required for the model. Moreover, we can enhance the inference time because integer operations are cheaper than floating point operations in general.

We will use TFLite Converter to apply int8 quantization. You can see that the 'supported_ops' variable is set as TFLITE_BUILTINS_INT8. You can change this parameter to use a different precision. Execute the code block to generate the quantized model.

```
# Convert using integer-only quantization
# Now you have an integer quantized model that uses integer data for
# the model's input and output tensors, so it's compatible with integer-only hardware

def representative_data_gen():
  for input_value in tf.data.Dataset.from_tensor_slices(x_train).batch(1).take(100):
    yield [input_value]

path_models = "./Data/models/"
path_keras_model = path_models+"custom_cifar10_model.h5"
model = tf.keras.models.load_model(path_keras_model)

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
```

```
# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set the input and output tensors to uint8 (APIs added in r2.3)
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_model_quantInt = converter.convert()
```

Then, save the model with the next code block.

```
path_tflite_model = path_models + "custom_cifar10_model_quantInt.tflite"

# Save the quantized int model:
with open(path_tflite_model, 'wb') as f:
    f.write(tflite_model_quantInt)
```

Let's check how the input and output changed with quantization. You can see that the types of input and output changed to int8.

```
# Input and output details
tflite_interpreter = tf.lite.Interpreter(model_path=path_tflite_model)
input_details = tflite_interpreter.get_input_details()
output_details = tflite_interpreter.get_output_details()

print("== Input details ==")
print("Name:", input_details[0]['name'])
print("Shape:", input_details[0]['shape'])
print("Type:", input_details[0]['dtype'])
print("quantisation scale {}, zero_point
{}".format(input_details[0]['quantization'][0],
input_details[0]['quantization'][1]))

print("\n== Output details ==")
print("Name:", output_details[0]['name'])
print("Shape:", output_details[0]['shape'])
print("Type:", output_details[0]['dtype'])
print("quantisation scale {}, zero_point
{}".format(output_details[0]['quantization'][0],
output_details[0]['quantization'][1]))
```

Expected Output:

```
== Input details ==
Name: conv2d_42_input_int8
Shape: [ 1 32 32  3]
Type: <class 'numpy.int8'>
quantisation scale 0.003921568859368563, zero_point -128
```

```
== Output details ==
Name: Identity_int8
Shape: [ 1 10]
Type: <class 'numpy.int8'>
quantisation scale 0.00390625, zero_point -128
```

You can retrieve the original value using the quantization scale and zero_point values. This is the formula to calculate the original value with the numbers[1].

$$real\_value = (int8\_value - zero\_point) * scale$$

Static Model Evaluation

We will compare the original and quantized models in terms of model size and accuracy.

Model Size

The following code block will show the sizes of the original and quantized models.

```
def get_gzipped_model_size(file):
    # Returns size of gzipped model, in bytes.
    _, zipped_file = tempfile.mkstemp('.zip')
    with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.ZIP_DEFLATED) as f:
        f.write(file)
    return os.path.getsize(zipped_file)

fp_size = get_gzipped_model_size(path_keras_model)
quant_size = get_gzipped_model_size(path_tflite_model)
print('Size of Full Precision Model: {} Bytes'.format(fp_size))
print('Size of quantised Model: {} Bytes'.format(quant_size))
print('Size reduction factor: {} times'.format(fp_size/quant_size))
```

Expected Output:

```
Size of Full Precision Model: 1175224 Bytes
Size of quantised Model: 101244 Bytes
Size reduction factor: 11.607838489194421 times
```

You can see that the quantized model is about 11 times smaller than the original model. This means that we can save more than 90% of memory to store the model.

[1] Reference: https://www.tensorflow.org/lite/performance/quantization_spec

Model Accuracy

Let's compare the accuracy of the original and quantized models. Since the quantized model uses a smaller number of bits to represent data, it may lose some information. Therefore, quantization may degrade the accuracy to some extent.

```
predictions = np.zeros((int(len(x_test)/100),), dtype=int)
input_scale, input_zero_point = input_details[0]["quantization"]
for i in range(int(len(x_test)/100)):
    val_batch = x_test[i]
    val_batch = val_batch / input_scale + input_zero_point
    val_batch = np.expand_dims(val_batch, axis=0).astype(input_details[0]["dtype"])
    tflite_interpreter.allocate_tensors()
    tflite_interpreter.set_tensor(input_details[0]['index'], val_batch)
    tflite_interpreter.invoke()

    tflite_model_predictions =
tflite_interpreter.get_tensor(output_details[0]['index'])
    #print("Prediction results shape:", tflite_model_predictions.shape)
    output = tflite_interpreter.get_tensor(output_details[0]['index'])
    predictions[i] = output.argmax()

sum = 0
for i in range(len(predictions)):
    if (predictions[i] == np.argmax(y_test[i])):
        sum = sum + 1
accuracy_score = sum / 100

full_precision_model = tf.keras.models.load_model(path_keras_model)
score = full_precision_model.evaluate(x_test, y_test, verbose=0)

print("Accuracy of quantized to int8 model is {}%".format(accuracy_score*100))
print("Compared to float32 accuracy of {}%".format(score[1]*100))
print("We have a change of {}%".format((accuracy_score-score[1])*100))
```

Expected Output:

```
Accuracy of quantized to int8 model is 83.0%
Compared to float32 accuracy of 82.70999789237976%
We have a change of 0.29000210762023526%
```

Surprisingly, you can see that the quantized model obtained higher accuracy than the original model. This means that quantization is successfully applied to the model.

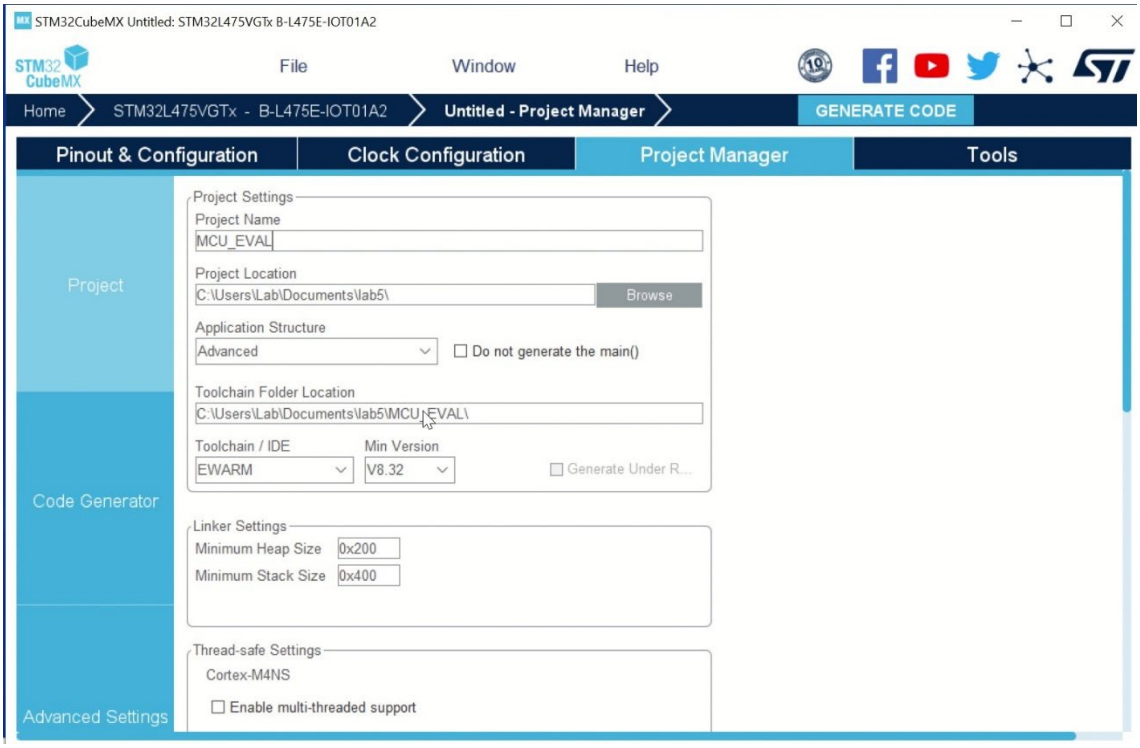
This code is also largely contributed by Pau Danilo and Carra Alessandro from STMicroelectronics.

Deploying Optimized Model

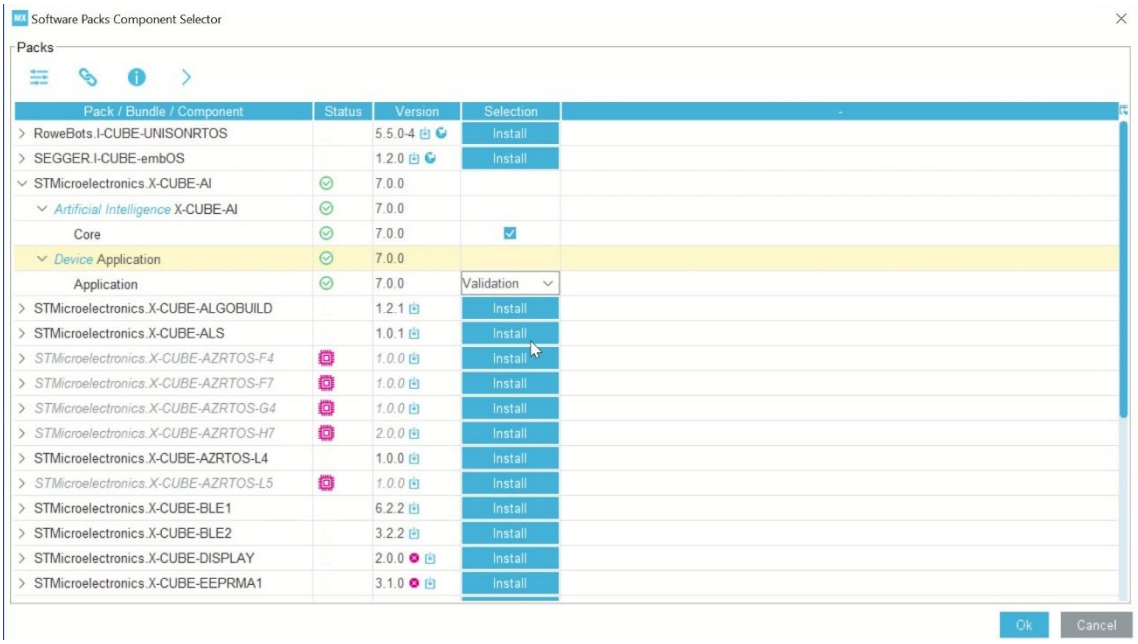
Let's deploy the models to measure the latency of the original and optimized models on the board.

Model Deployment

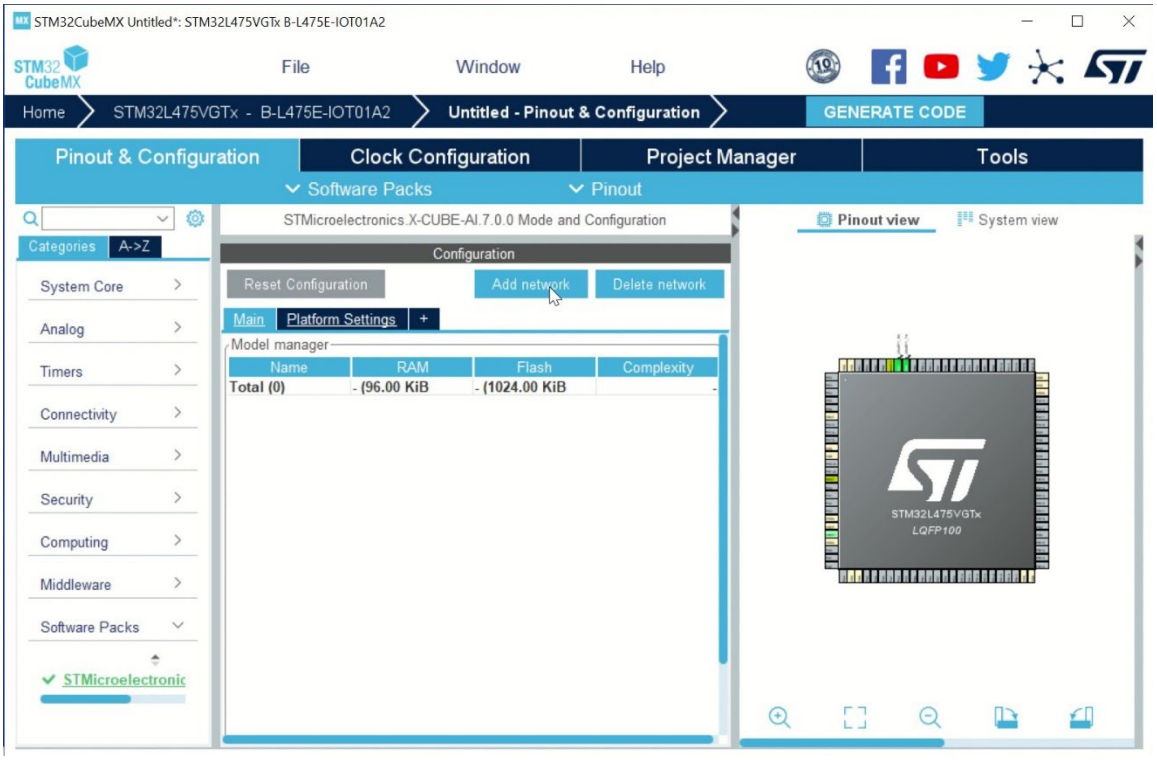
Open STM32CubeMX to create a Cube AI validation application. Click [Access to Board Selector]. Find the board and click [Start Project]. Go to [Project Manager]. Set the project name. Configure the project location and the firmware version.



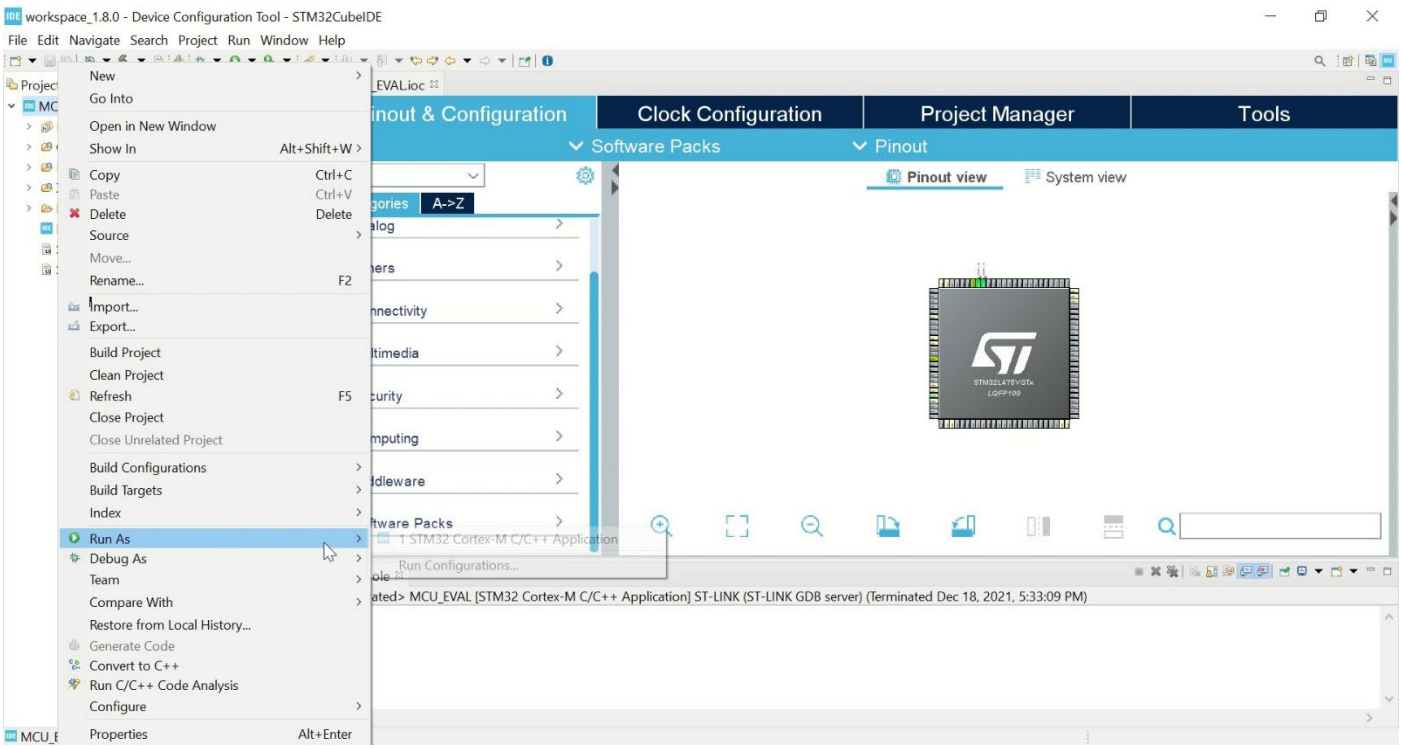
Clear pinouts by selecting 'Pinout > Clear Pinouts' and enable Cube AI. Choose [Validation] for device application.



After enabling Cube AI, add the original model clicking the ‘Add network’ button. We are going to measure the latency using [Validate on Target]. To do so, we need to first install the code to the board. So, click generate code.



Open STM32CubeIDE and import the generated project. Click [Run As] to install the code.

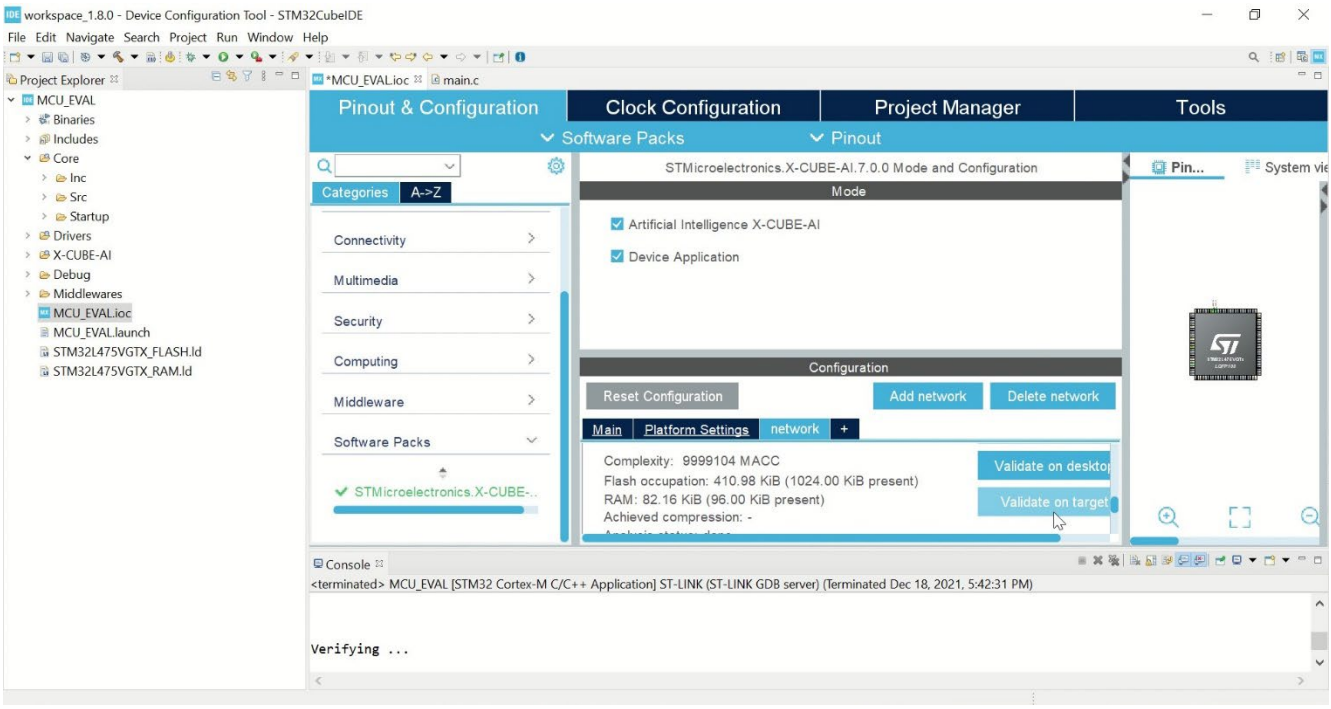


If you got the ‘undefined reference’ error, open ‘Core/Src/main.c’ and modify it as follows.

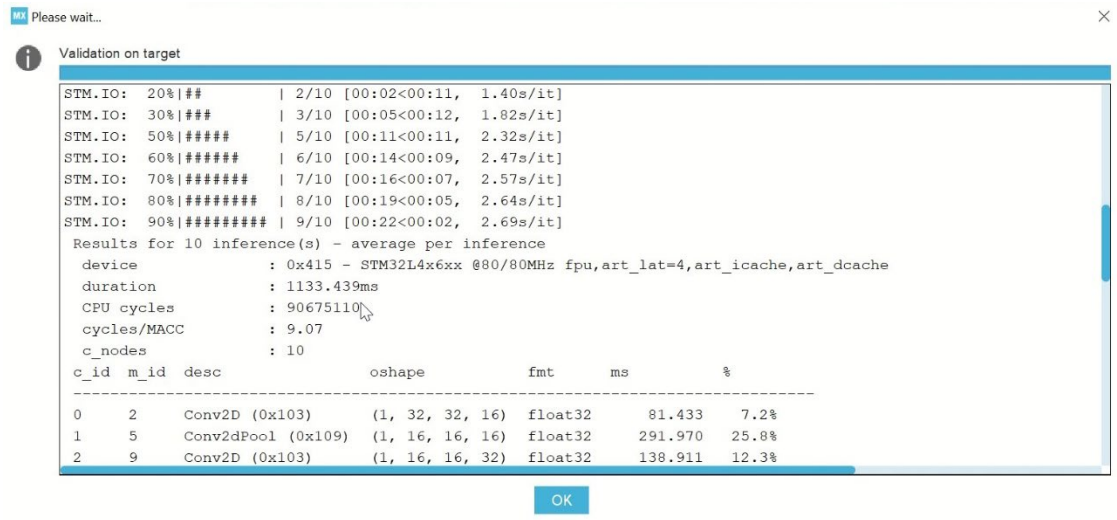
```
static void MX_USART1_UART_Init(void);

static void MX_USART1_UART_Init(void)
{
```

After the installation is finished, open the ioc file (MCU_EVAL.ioc) on STM32CubeIDE, go to the Cube AI menu, and click [Validate on Target].



Expected Output:



In this validation report, you can check the average number of cycles and the average inference time, taken to process the 10 samples. I would copy this result into a text file. Here, you can also check the complexity, flash occupation, and the amount of memory required for the model.

Now the original model is validated. Remove the original network and add the quantized model. Select the TFLite option because the quantized model is in TFLite format. Then, press [Ctrl] + [S], then the code will be automatically updated. Modify the main file again and click [Run As] to install the new model. The installation is done. Go back to the ioc file and click [Validate on Target] to get the report.

Dynamic Model Evaluation

With the validation results, compare the original and optimized models in terms of latency and energy consumption.

Sample results:

	Original Model	**Optimized Model**	**Difference (%)**
Inference Time	1133.439 ms (90675110 cycles)	318.464 ms (25477095 cycles)	**71.9%**
Memory Size (RAM)	82.16 KB	25.82 KB	**68.6%**
Energy Consumption	43 * 1.133 mJ	43 * 0.318 mJ	**71.9%**

In terms of latency, the quantized model obtained 72% better latency than the original model. Second, the quantized model requires 69% less memory than the original model. Third, since energy consumption is proportional to the latency, the quantized model obtained 72% less energy consumption than the original model.