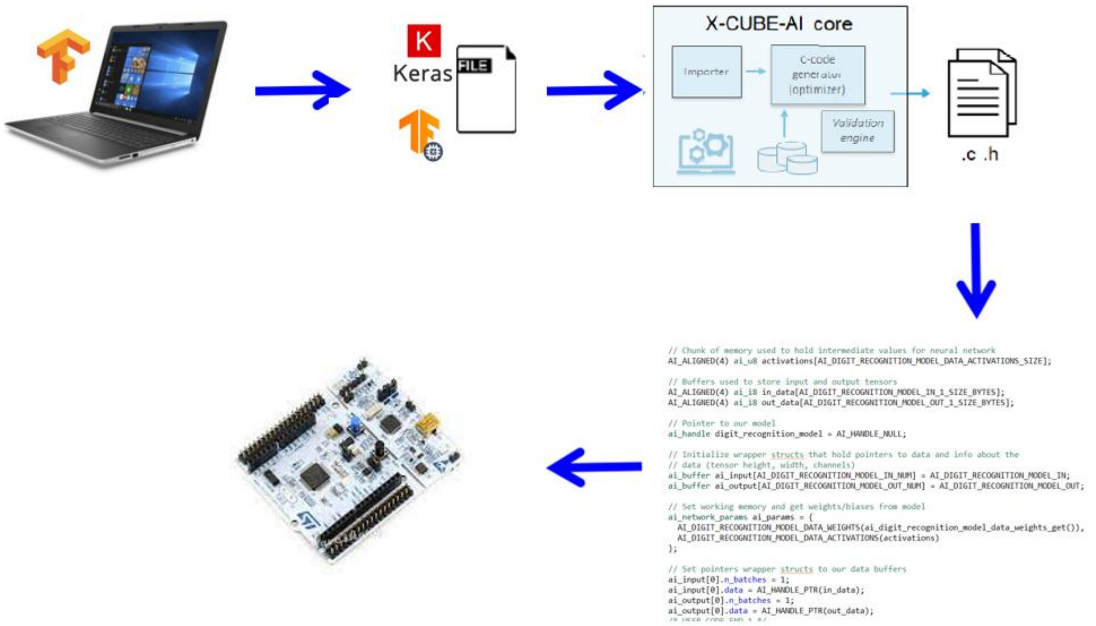
Documentation

Tiny machine learning (TinyML)

Tiny machine learning (TinyML) is a concept of embedding artificial intelligence on a small pieces of hardware. With TinyML, it is possible to deploy the machine learning algorithm to train the network on device and shrink their size to an optimization form for embedded device without the need of sending the data for cloud computing. Many problems regarding the significance of computing capabilities on data analyzing such as storage capacity, limited central processing unit (CPU) and reduced database performance can be solved through added latency from TinyML. In this tutorial, we will show you how we create a neural network model using TensorFlow platform and deploy the pre-trained model into STM32F446 chip to run inference for health condition recognition.



```
// Chunk of memory used to hold intermediate values for neural network
AI_ALIGNED(4) ai_u8 activations[AI_DIGIT_RECOGNITION_MODEL_DATA_ACTIVATIONS_SIZE];

// Buffers used to store input and output tensors
AI_ALIGNED(4) ai_i8 in_data[AI_DIGIT_RECOGNITION_MODEL_IN_1_SIZE_BYTES];
AI_ALIGNED(4) ai_i8 out_data[AI_DIGIT_RECOGNITION_MODEL_OUT_1_SIZE_BYTES];

// Pointer to our model
ai_handle digit_recognition_model = AI_HANDLE_NULL;

// Initialize wrapper structs that hold pointers to data and info about the
// data (tensor height, width, channels)
ai_buffer ai_input[AI_DIGIT_RECOGNITION_MODEL_IN_NUM] = AI_DIGIT_RECOGNITION_MODEL_IN;
ai_buffer ai_output[AI_DIGIT_RECOGNITION_MODEL_OUT_NUM] = AI_DIGIT_RECOGNITION_MODEL_OUT;

// Set working memory and get weights/biases from model
ai_network_params ai_params = {
  AI_DIGIT_RECOGNITION_MODEL_DATA_WEIGHTS(ai_digit_recognition_model_data_weights_get()),
  AI_DIGIT_RECOGNITION_MODEL_DATA_ACTIVATIONS(activations)
};

// Set pointers wrapper structs to our data buffers
ai_input[0].n_batches = 1;
ai_input[0].data = AI_HANDLE_PTR(in_data);
ai_output[0].n_batches = 1;
ai_output[0].data = AI_HANDLE_PTR(out_data);
/* USER CODE END 3 */
```

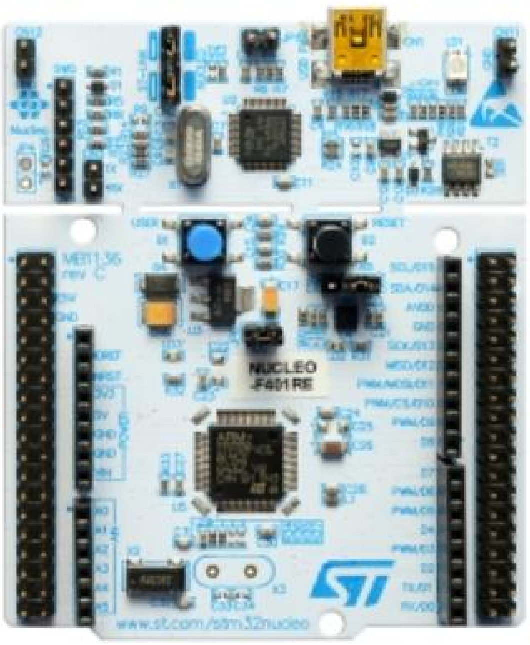
Tool Used

TensorFlow

An open-source library for numerical computation and large-scale machine learning.

NUCLEO-F446RE

STMicroelectronics Development Boards used to build our prototypes. The STM32 Nucleo-64 board provides an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 micro-controller. For the compatible boards, the external SMPS significantly reduces power consumption in Run mode. The ARDUINO® Uno V3 connectivity support and the ST morpho headers allow the easy expansion of the functionality of the STM32 Nucleo open development platform with a wide choice of specialized shields. The STM32 Nucleo-64 board does not require any separate probe as it integrates the ST-LINK debugger/programmer. The STM32 Nucleo-64 board comes with the STM32 comprehensive free software libraries and examples available with the STM32Cube MCU Package. The spec documents can be found in this link.



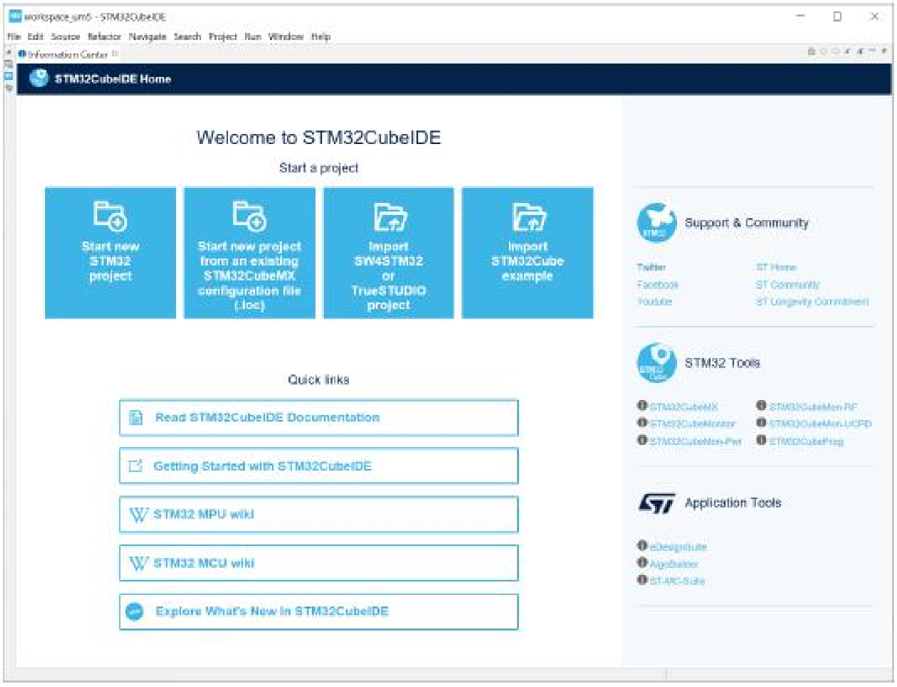
STM32CubeIDE

Advanced development platform with peripheral configuration, code generation, code compilation and debug features for micro-controllers.

STM32CubeIDE

Data brief

Integrated development environment for STM32 products



Product status link
STM32CubeIDE

Model Training

Our model is trained using Tensor Flow platform. To train the model, the 1000 samples.csv is collected containing the Oxygen level, Pulse rate, Temperature and Heart Rate variability, The final output is determined based on the below table:

If any column reaches the range, the result is reflected.

Result = **Blood Oxygen Level** OR **Heart Pulse Rate** OR **Body Temperature** OR **HRV**

Blood Oxygen Level (%)	**Heart Pulse Rate (/min)**	**Body Temperature (℃)**	**Heart Rate Variability**	**Result**
> 95	40 - 109	36.5 - 38.0	> 50	Normal (0)
93 - 94	110 - 130	38.1 - 39.0	< 50	Seek medical advice (1)
< 92	> 131	> 39.0	NA	Urgent medical advice (2)

Table 1

The features are extracted from the sample.csv and sample_features is created. The final column is contained in sample_labels as shown below:

```
#Data_collecting and pre_processing

headers=['oxygen', 'bpm', 'temp', 'hrv', 'final']

#data from csv

sample=pd.read_csv("C:\\Mtech\\advanced_microprocessor_systems\\Assignment\\sample.csv")

print(sample)

#create result set

sample_labels = sample.pop('final')

sample_features = sample.copy()

#create features set

sample_features = np.array(sample_features)

#print features and result

print(sample_features)

print(sample_labels)
```

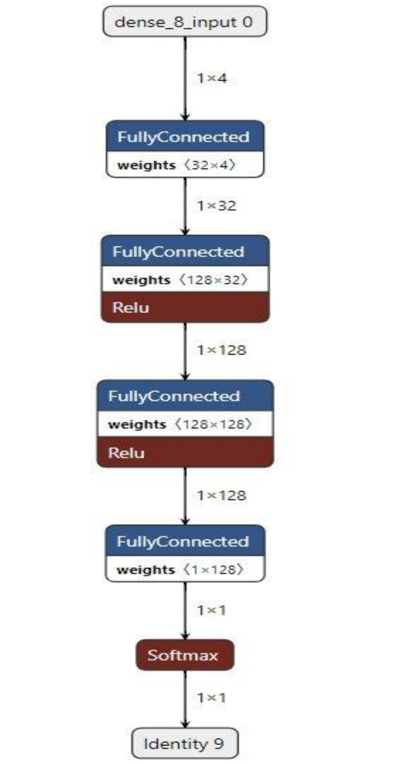
Then the sample_features and sample_labels split into train data and test data. The test ratio is 20% and the train split is 80%. Sample_features is split into Sample_features_train and Sample_features_test. Sample_labels is split into Sample_labels_train and Sample_labels_test as shown below:

```
#Data_collecting and pre_processing

headers=['oxygen', 'bpm', 'temp', 'hrv', 'final']

#data from csv

sample=pd.read_csv("C:\\Mtech\\advanced_microprocessor_systems\\Assignment\\sample.csv")

print(sample)

#create result set

sample_labels = sample.pop('final')

sample_features = sample.copy()

#create features set

sample_features = np.array(sample_features)

#print features and result

print(sample_features)

print(sample_labels)
```

Then the normalization and training of the sets are done as below: The model is created with 1 output.

```
#normalization

samples_features_train = tf.keras.utils.normalize(sample_features_train, axis=1)

sample_features_test = tf.keras.utils.normalize(sample_features_test, axis=1)

 #training

model = tf.keras.Sequential()

model.add(Dense(32, input_shape=(4,)))

model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))

model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))

model.add(tf.keras.layers.Dense(units=1, activation=tf.nn.softmax))
```

The model likes as shown below:



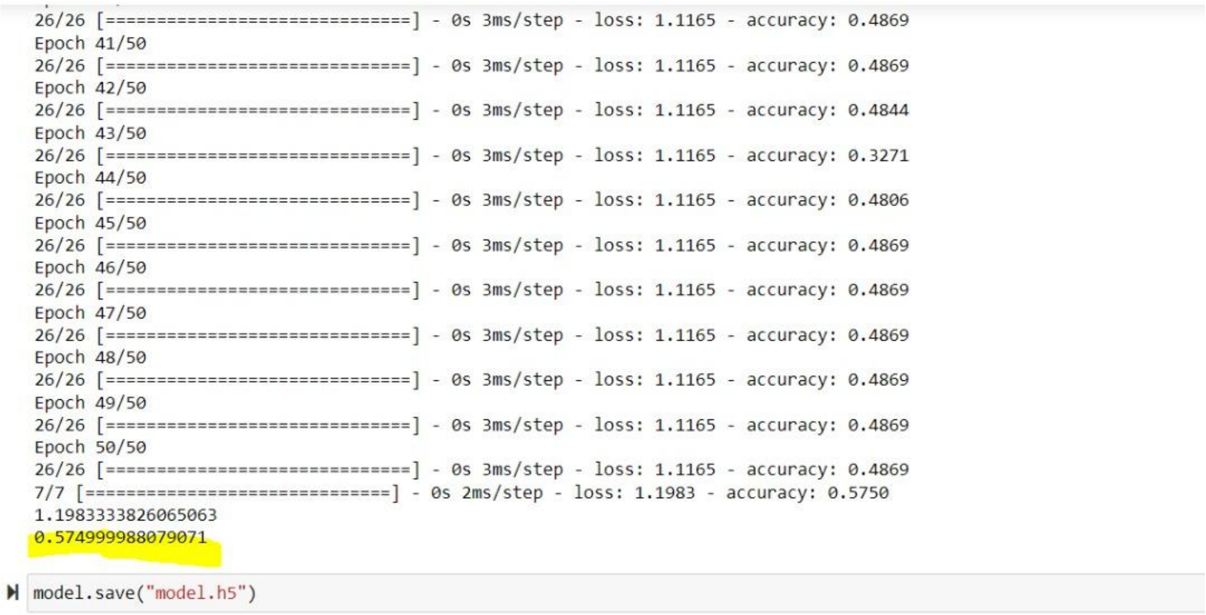
Then the compilation is done.

```
#compilation

model.compile(optimizer='adam', loss='mae', metrics=['accuracy'])
```

Then the model is trained in 50 epochs which means 50 runs for network training.

```
#train the model

model.fit(sample_features_train, sample_labels_train, epochs=50)

#testing the model for accuracy

loss, accuracy = model.evaluate(sample_features_test, sample_labels_test)

print(loss)

print(accuracy)
```

The accuracy obtained is as shown below:



```
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 41/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 42/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4844
Epoch 43/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.3271
Epoch 44/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4806
Epoch 45/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 46/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 47/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 48/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 49/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
Epoch 50/50
26/26 [==============================] - 0s 3ms/step - loss: 1.1165 - accuracy: 0.4869
7/7 [==============================] - 0s 2ms/step - loss: 1.1983 - accuracy: 0.5750
1.1983333826065063
0.574999988079071
```

```
model.save("model.h5")
```

The trained model is saved and converted into Keras(.h5) and Tensorflow(.tflite) format files.

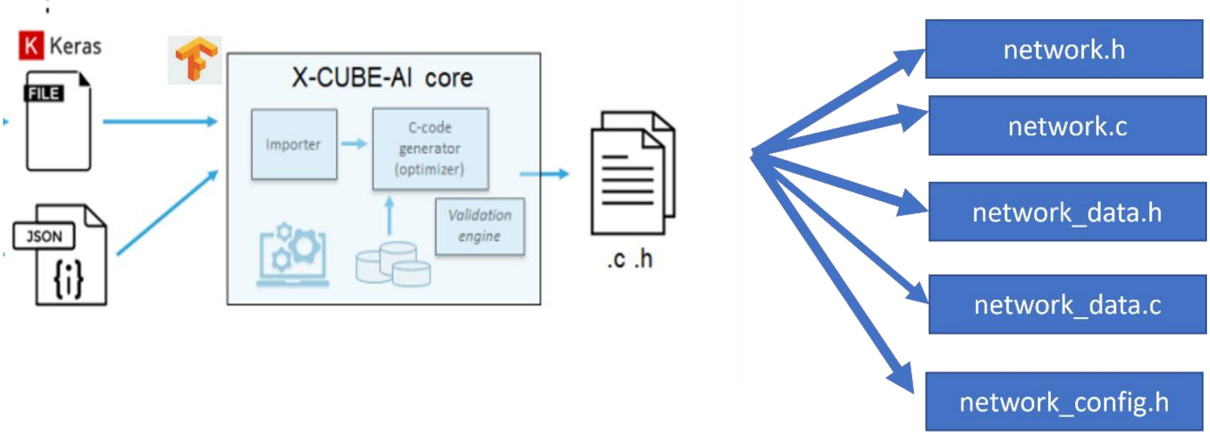
```
model.save("new_model.h5")

#convert keras model into tensorflowlite model

converter= tf.lite.TFLiteConverter.from_keras_model(model)

#converter.optimizations=[tf.lite.Optimize.OPTIMIZE_FOR_SIZE]

tflite_model=converter.convert()

with open('new_model.tflite', 'wb') as f:

    f.write(tflite_model)
```

The full script is done in jupyter notebook and run using TensorFlow. The full script "health_model.ipynb" is uploaded.

Generating C-model using X-CUBE-AI

The X-CUBE-AI core engine is part of the X-CUBE-AI Expansion Package. It provides an automatic and advanced NN mapping tool to generate and deploy an optimized and robust C-model implementation of a pre-trained Neural Network for the embedded systems with limited and constrained hardware resources. The generated STM32 NN library (both specialized and generic parts) can be directly integrated in an IDE project or makefile-based build system. The user manual of X-CUBE-AI can be referred in this link.

Refer to this tutorial, the X-CUBE-AI is installed, and the new project is created. Then, our C-model with STM32 NN library is generated by importing the pre-trained Keras or TensorFlow Lite model file into the X-CUBE-AI core engine. After generating the C-model, 5 files should be occurred in our working directory "X-CUBE-AI/App". Since our network's name is network, the 5 files generated are network.h, network.c, network_data.h, network_data.c and network_config.h.



The header file network.h consists of declaration on the input and output tensor size as well as tensor dimension (width, height and channel). It also consists of declaration of the main NN functions used to run reference. The network.c consists of declaration on the weights and biases parameters of the network. The network_data.c file is where our neural network which consists of huge array of weights stored in. The network.h file consists of functions that initialize the pointer to our activation and weights.

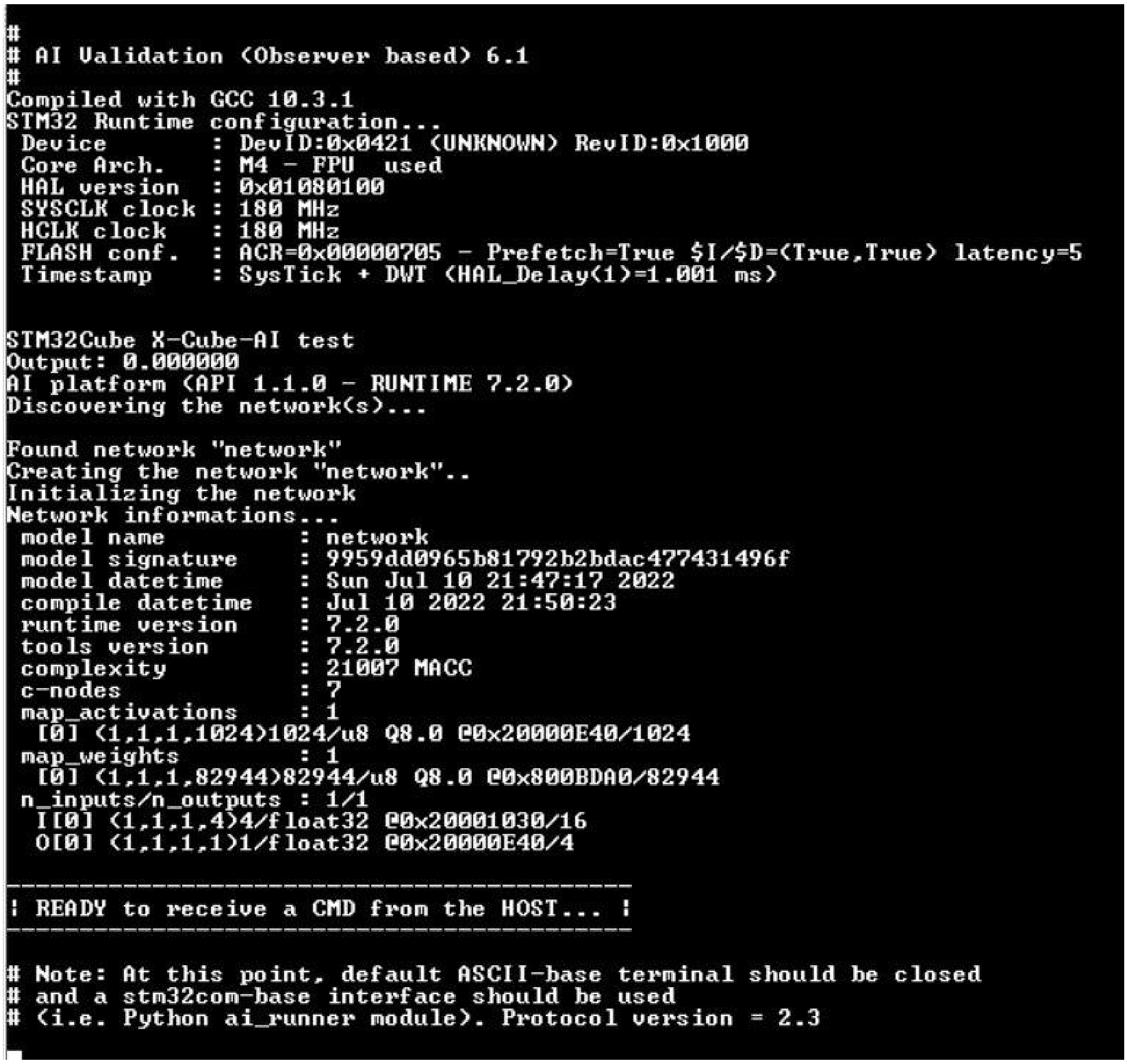
Run Inference for health model

The following code is written in main. c to perform inference for digit recognition on NUCLEO-F446RE. The input is given for the values {96,70,38,67} in our code and the output is obtained as 0 which is according to our sample output table 1 as mentioned above in this document.

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  char buf[50];
  uint32_t arr[4]={96,70,38,67};
  int buf_len = 0;
  uint32_t timestamp;
  float y_val;
  ai_error ai_err;
  ai_i32 nbatch;

  // Chunk of memory used to hold intermediate values for NN
  AI_ALIGNED(32) ai_u32 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];

  // Buffers used to store input and output tensors
  AI_ALIGNED(32) ai_i32 in_data[AI_NETWORK_IN_1_SIZE_BYTES];
  AI_ALIGNED(32) ai_i32 out_data[AI_NETWORK_OUT_1_SIZE_BYTES];

  // Pointer to our model
  ai_handle network = AI_HANDLE_NULL;

  // Set working memory and get weights/biases from model
  ai_network_params ai_params = {
     { AI_NETWORK_DATA_WEIGHTS(ai_network_data_weights_get()),
       AI_NETWORK_DATA_ACTIVATIONS(activations)} };

  // Set pointers wrapper structs to our data buffers
  ai_input[0].data = AI_HANDLE_PTR(in_data);
  ai_output[0].data = AI_HANDLE_PTR(out_data);

  /* USER CODE END 1 */

  /* MCU Configuration*/
  /* Reset of all peripherals, Initializes the Flash interface and the
     Systick.*/
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_CRC_Init();
  MX_TIM14_Init();
  MX_X_CUBE_AI_Init();

  /* USER CODE BEGIN 2 */
```

```
//Greetings!
buf_len=sprintf(buf, "\r\n\r\nSTM32Cube X-Cube-AI test\r\n");
 HAL_UART_Transmit(&huart2,(uint8_t *)buf, buf_len,100);

// Create instance of NN
ai_err = ai_network_create(&network, AI_NETWORK_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);
 }

// Initialize NN
 if (!ai_network_init(network, &ai_params))
 {
  buf_len = sprintf(buf, "Error: could not initialize NN\r\n");
  HAL_UART_Transmit(&huart2, (uint8_t *)buf, buf_len, 100);
  while(1);
 }

 // structs that point to our input and output data and some meta data
 ai_input = ai_network_inputs_get(network, NULL);
 ai_output= ai_network_outputs_get(network, NULL);

 /* USER CODE END 2 */

 /* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
   // Fill input buffer (use test value)
   for (uint32_t i = 0; i <  AI_NETWORK_IN_1_SIZE; i++)
   {
     ((ai_float *)in_data)[i] = (ai_float)arr[i];
   }
   timestamp=htim14.Instance->CNT;

   // Perform inference
   nbatch = ai_network_run(network, &ai_input[0], &ai_output[0]);
   if (nbatch != 1)
   {
      buf_len = sprintf(buf, "Error: could not run inference\r\n");
      HAL_UART_Transmit(&huart2, (uint8_t *)buf, buf_len, 100);
   }

   // Read output (predicted y) of NN
   y_val = ((float *)out_data)[4];

   // Print output of NN along with inference time (microseconds)
   buf_len = sprintf(buf, "Output: %f ", y_val,
        htim14.Instance->CNT - timestamp);
   HAL_UART_Transmit(&huart2, (uint8_t *)buf, buf_len, 100);
 }
/* USER CODE END WHILE */
```

The screenshot of console window is as shown below:



```
#
# AI Validation (Observer based) 6.1
#
Compiled with GCC 10.3.1
STM32 Runtime configuration...
 Device         : DevID:0x0421 (UNKNOWN) RevID:0x1000
 Core Arch.     : M4 - FPU  used
 HAL version    : 0x01080100
 SYSCLK clock : 180 MHz
 HCLK clock     : 180 MHz
 FLASH conf.    : ACR=0x00000705 - Prefetch=True $I/$D=(True,True) latency=5
 Timestamp      : SysTick + DWT (HAL_Delay(1)=1.001 ms)


STM32Cube X-Cube-AI test
Output: 0.000000
AI platform (API 1.1.0 - RUNTIME 7.2.0)
Discovering the network(s)...

Found network "network"
Creating the network "network"..
Initializing the network
Network informations...
 model name          : network
 model signature     : 9959dd0965b81792b2bdac477431496f
 model datetime      : Sun Jul 10 21:47:17 2022
 compile datetime    : Jul 10 2022 21:50:23
 runtime version     : 7.2.0
 tools version       : 7.2.0
 complexity          : 21007 MACC
 c-nodes             : 7
 map_activations     : 1
  [0] (1,1,1,1024)1024/u8 Q8.0 @0x20000E40/1024
 map_weights         : 1
  [0] (1,1,1,82944)82944/u8 Q8.0 @0x800BDA0/82944
 n_inputs/n_outputs : 1/1
  I[0] (1,1,1,4)4/float32 @0x20001030/16
  O[0] (1,1,1,1)1/float32 @0x20000E40/4

-------------------------------------------
| READY to receive a CMD from the HOST... |
-------------------------------------------

# Note: At this point, default ASCII-base terminal should be closed
# and a stm32com-base interface should be used
# (i.e. Python ai_runner module). Protocol version = 2.3

```

Validation on results

The input array is given as {96,70,38,67} and the output is obtained as 0 since the sample output is within the 0 values as per the table 1 shown above. The serial port is set at 115200 baud rate to display the output