Enabling Intelligent edge devices with ultra low-power Arm MCUs and TensorFlow Lite

Introduction

In this guide, you will learn how you can perform machine learning inference on an Arm Cortex-M microcontroller with TensorFlow Lite for Microcontrollers. We are going to classify images from the CIFAR-10 dataset, an established computer-vision dataset used for object recognition. The dataset consists of 60,000(50,000 training images and 10,000 test images) 32x32 color images containing one of 10 object classes, with 6000 images per class.

Before you begin

Here is what you will need to complete the guide:

- STM32H747I discovery kit board
- Mini-USB cable
- Install Mbed CLI

  Linux/MacOS: `$sudo pip install mbed-cli`

  MacOS/Windows: https://os.mbed.com/docs/mbed-os/v5.15/tools/installation-and-setup.html

- Install Arm cross compilation(https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads). Please scroll down the download page and install version 6.x or 7.x. Higher version cross compilers will result in unexpected failures.

- Install Git
- (Optional) On Windows machines, install serial terminal programs like CoolTerm or Putty.

Getting started

TensorFlow Lite for Microcontrollers supports several devices out of the box, and is relatively easy to extend to new devices. For this guide, we will focus on the STM32H747I Discovery kit.

To run our image classification sample, we will show you how to complete the following steps:

1. Configure Arm cross compilation toolchain and Mbed CLI
2. Build and compile CIFAR-10 image classification sample
3. Dive into the code
4. Deploy the sample to your STM32H747I board
5. Run Image Classification

Configure Arm cross compilation toolchain and Mbed CLI

Install Arm cross compilation if you haven't (https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads). Please scroll down the page and install version 6.x or 7.x. Higher version cross compilers will result in unexpected failures.

1. To build and deploy the application, we will use the Mbed CLI.
   On Mac or Linux, you can install from the command line:

   ```
   $ sudo pip install mbed-cli
   ```

   On Windows, Mbed CLI is installed at **C:\mbed-cli\** by default. Type mbed from the command console. If this doesn't work, refer to Troubleshooting for additional configuration.

2. After Mbed CLI is installed, tell Mbed where to find the Arm embedded toolchain.

For Linux/Mac:

```
$ mbed config -G GCC_ARM_PATH <location_of_your_GCC_ARM_toolchain>
```

For Windows:

```
$ mbed config -G GCC_ARM_PATH C:\Program Files (x86)\GNU Tools ARM
Embedded\6 2017-q2-major\bin
```

Build and compile CIFAR-10 image classification example

1. Clone the TensorFlow repo:

Navigate to the directory where you keep code projects, and run the following command to download TensorFlow Lite source code.

```
$git clone https://github.com/wxarm/tinyML.git
```

2. Create the Mbed project:

After downloading TensorFlow, you can navigate into the TensorFlow directory and create a Mbed project for the image recognition sample.

For Linux/Mac:

```
     $cd tinyML/tensorflow

      $make -f tensorflow/lite/micro/tools/make/Makefile
      TAGS=disco_h747i generate_image_recognition_mbed_project
```

This will create a folder in `tensorflow/lite/micro/tools/make/gen/`**`osx_x86_64`**`/prj/image_recognition/mbed/` that contains the source and header files, Mbed driver files, and a README. `osx_x86_64` represents the OS & architecture combination of your host machine. On Linux, this is `linux_x86_64`.

For Windows:

Download the mbed.zip and unzip it to your Windows machine.

3. Navigate to the Mbed project folder: Replace …_`x86_64` in the command below with the architecture name of your host machine.

```
$cd
tensorflow/lite/micro/tools/make/gen/…_x86_64/prj/image_recognition/mbed/
```

4. Load the dependencies required, which includes Mbed OS, BSP files, run:

```
$ cat > .mbed << EOF

> ROOT=.

> TARGET=DISCO_H747I

> TOOLCHAIN=GCC_ARM

> EOF



$mbed config root .

$mbed deploy
```

5. After that setting is updated, you can compile your Mbed project as below:

```
$mbed compile -m DISCO_H747I -t GCC_ARM
```

Here is the description of the structure of the code base:

- **tensorflow/lite/micro/examples/image_recognition** is the original image_recognition example folder.

- **tensorflow/lite/micro/tools/make/gen/osx_x86_64/prj/image_recognition/mbed:** generated Mbed project folder.

- **tensorflow/lite/micro/tools/make/gen/osx_x86_64/prj/image_recognition/mbed/tensorflow/lite/micro/examples/image_recognition/:** Image recognition source code of the generated Mbed project. If you need to make further changes to the source code after generating the Mbed project, please change here.

- **tensorflow/lite/micro/tools/make/downloads/image_recognition_model/image_recognition_model.cc**: This file is the CIFAR-10 image classification machine learning model itself, represented by a large array of unsigned char values. This file is downloaded during the compilation stage.

How is the model file generated?

CIFAR-10 TensorFlow model can be trained by using the scripts here (https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10). Starting from the trained model to obtain a converted model that can run on the controller itself, we need to run the TensorFlow Lite converter script: `tflite_convert`. This tool uses tricks to make the model as small and efficient as possible, and to convert it to a TensorFlow Lite FlatBuffer.

To reduce the size of the model, we use a technique called quantization. All weights and activations in the model get converted from 32-bit floating point format to an 8-bit and fixed-point format.

This conversion will not only reduce the size of the network, but will avoid floating point computations that are more computationally expensive.

The final step in the process is to convert this model into a C file that we can drop into our Mbed project. To do this conversion, we will use a tool called xxd. Issue the following command:

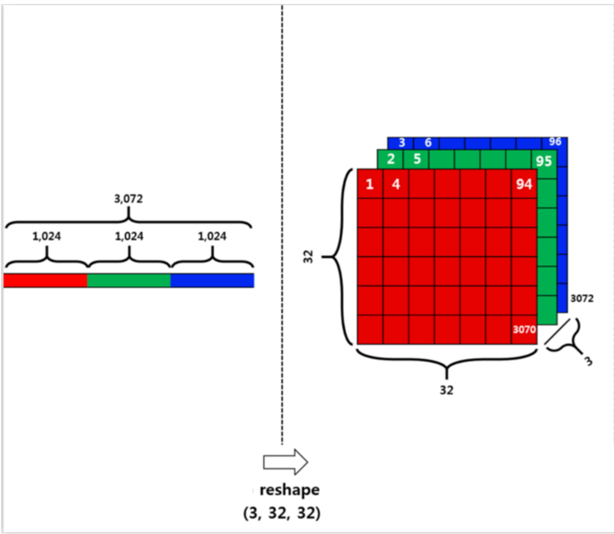
```
xxd -i image_recognition.tflite > image_recognition_model.cc
```

- **main.cc**: This file is the entry point for the image classification program, which runs the machine learning model using TensorFlow Lite for Microcontrollers.

- **first_10_cifar_images.h**: The *.h file is converted from CIFAR-10 test images binary version test_batch.bin. The test batch data consists of 10000 sample images in the format below.

```
<1 x label><3072 x pixel>
...
<1 x label><3072 x pixel>
```

The first byte is the label of the first image, which is a number in the range 0-9. The next 3072 bytes are the values of the pixels of the image, which represents a color image of 32x32 pixel with 3 channels(R, G, B). The vector (3072) has the exact same number of elements if you calculate 32*32*3==3072. In order to process the image data, we reshape the row vector (3072) as below.



The file contains 10 images of 3073-byte. Therefore the file size should be exactly 30730 bytes long.

Dive into the code

Main.cc is the main entry point of our program, which demonstrates how to run inference using TensorFlow Lite for microcontrollers.

Include the library headers

To use the TensorFlow Lite for Microcontrollers library, we must include the following header files:

```
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

- `all_ops_resolver.h` provides the operations used by the interpreter to run the model.
- `micro_error_reporter.h` outputs debug information.
- `micro_interpreter.h` contains code to load and run models.
- `schema_generated.h` contains the schema for the TensorFlow Lite `FlatBuffer` model file format.
- `version.h` provides versioning information for the TensorFlow Lite schema.

Include the model

The TensorFlow Lite for Microcontrollers interpreter expects the model to be provided as a C++ array. In our example, the model is defined in `image_recognition_model.h` and `image_recognition_model.cc`. The header is included with the following line:

```
#include
"tensorflow/lite/micro/examples/image_recognition/image_recognition_model.h"
```

Set up the unit test

The code we are walking through is a unit test that uses the TensorFlow Lite for Microcontrollers unit test framework. To load the framework, we include the following file:

```
#include "tensorflow/lite/micro/testing/micro_test.h"
```

The test is defined using the following macros:

```
TF_LITE_MICRO_TESTS_BEGIN

TF_LITE_MICRO_TEST(LoadModelAndPerformInference) {
```

The remainder of the code demonstrates how to load the model and run inference.

Set up logging

To set up logging, a `tflite::ErrorReporter` pointer is created using a pointer to a `tflite::MicroErrorReporter` instance:

```
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* error_reporter = &micro_error_reporter;
```

This variable will be passed into the interpreter, which allows it to write logs. Since microcontrollers often have a variety of mechanisms for logging, the implementation of `tflite::MicroErrorReporter` is designed to be customized for your particular device.

Load a model

In the following code, the model is instantiated using data from a `char` array `image_recognition_model_data`, which is declared in `image_recognition_model.h`. We then check the model to ensure its schema version is compatible with the version we are using:

```
const tflite::Model* model
= ::tflite::GetModel(image_recognition_model_data);
if (model->version() != TFLITE_SCHEMA_VERSION) {
  error_reporter->Report(
      "Model provided is schema version %d not equal "
      "to supported version %d.\n",
      model->version(), TFLITE_SCHEMA_VERSION);
}
```

Instantiate operations resolver

An `AllOpsResolver` instance is declared. This will be used by the interpreter to access the operations that are used by the model:

```
tflite::ops::micro::AllOpsResolver resolver;
```

The `AllOpsResolver` loads all of the operations available in TensorFlow Lite for Microcontrollers, which uses a lot of memory. Since a given model will only use a subset of these operations, it's recommended that real world applications load only the operations that are needed.

Allocate memory

We need to pre-allocate a certain amount of memory for input, output, and intermediate arrays. This is provided as a `uint8_t` array of size `tensor_arena_size`:

```
const int tensor_arena_size = 63 * 1024;
uint8_t tensor_arena[tensor_arena_size];
```

The size required will depend on the model you are using, and may need to be determined by experimentation.

Instantiate interpreter

We create a `tflite::MicroInterpreter` instance, passing in the variables created earlier:

```
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,
                                     tensor_arena_size, error_reporter);
```

Allocate tensors

We tell the interpreter to allocate memory from the `tensor_arena` for the model's tensors:

```
interpreter.AllocateTensors();
```

Validate input shape

The `MicroInterpreter` instance can provide us with a pointer to the model's input tensor by calling `.input(0)`, where `0` represents the first (and only) input tensor:

```
// Obtain a pointer to the model's input tensor
TfLiteTensor* input = interpreter.input(0);
```

We then inspect this tensor to confirm that its shape and type are what we are expecting:

```
// Make sure the input has the properties we expect
TF_LITE_MICRO_EXPECT_NE(nullptr, input);
// The property "dims" tells us the tensor's shape. It has one element for
// each dimension.
TF_LITE_MICRO_EXPECT_EQ(4, input->dims->size);
// The value of each element gives the length of the corresponding tensor.
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(32, input->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(32, input->dims->data[2]);
TF_LITE_MICRO_EXPECT_EQ(3, input->dims->data[3]);
// The input is a 8 bit uint value
TF_LITE_MICRO_EXPECT_EQ(kTfLiteUInt8, input->type);
```

The enum value `kTfLiteUInt8` is a reference to one of the TensorFlow Lite data types, and is defined in `common.h`.

Run the model

To run the model, we can call `Invoke()` on our `tflite::MicroInterpreter` instance:

```
TfLiteStatus invoke_status = interpreter.Invoke();
if (invoke_status != kTfLiteOk) {
  error_reporter->Report("Invoke failed\n");
}
```

We can check the return value, a `TfLiteStatus`, to determine if the run was successful. The possible values of `TfLiteStatus` are `kTfLiteOk` and `kTfLiteError`.
The following code asserts that the value is `kTfLiteOk`, meaning inference was successfully run.

```
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);
```

Obtain the output
The model's output tensor can be obtained by calling `output(0)` on the `tflite::MicroIntepreter`, where `0` represents the first (and only) output tensor.

Deploy the example to your STM32H7

Now that the build has completed, we will look in this section of the guide at how to deploy the binary to the STM32H7 and test to see if it works.

First, plug in your STM32H7 board via USB. The board should show up on your machine as a USB mass storage device. Copy the binary file that we built earlier to the USB storage.

For Linux/Mac:
Use the following command:

```
$cp ./BUILD/DISCO_H747I/GCC_ARM/mbed.bin /Volumes/DIS_H747I/
```

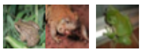
For Windows:
STM32H7 shows up as a new drive, for example, D. You can simply drag the generated binary at `.\BUILD\DISCO_H747I\GCC_ARM-RELEASE\mbed.bin` and drop it to the drive.

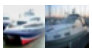
Depending on your platform, the exact copy command and paths may vary. When you have copied the file, the LEDs on the board should start flashing, and the board will eventually reboot with the sample program running.

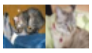
From your device display, you will be able to see the inference results of the following images:

Airplane

Automobile


Frog


Ship


Cat

Test from serial console

This step is only required when you debug your code. The program outputs debug info and recognition results to its serial port. To see the output of the program, we will need to establish a serial connection with the board at 9600 baud.

For Linux/Mac:

The board's USB UART shows up as `/dev/tty.usbmodemXXXXXXX.` We can use 'screen' to access the serial console, this is not installed on Linux by default, but you can use `apt-get install screen` to install the package.

Run the following command in a separate terminal:

```
$screen /dev/tty.usbmodemXXXXXX 9600
```

For Windows:
Launch CoolTerm or Putty or another serial console program of your choice.
Once you connect to the board, you will see any recognition results printed to the terminal.

Congratulations! You are now running a machine learning model that can do image classification on an Arm Cortex-M7 microcontroller, directly on your STM32H7.

Troubleshooting

We have found some common errors users face and listed them here to help you get started with your application as quickly as possible.

1. Mbed CLI issues or Error: **collect2: error: ld returned 1 exit status**

Purge the cache with the following command:

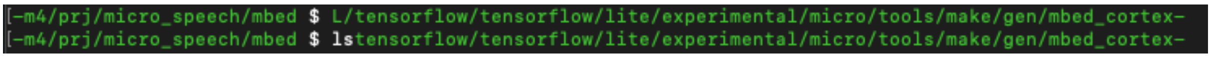
```
mbed cache purge
```

You probably also have a stale BUILD folder. Clean up your directory and try again:

```
rm -rf BUILD
```

2. Error: **Prompt wrapping around line**

If your terminal is wrapping text as shown here:



```
[-m4/prj/micro_speech/mbed $ L/tensorflow/tensorflow/lite/experimental/micro/tools/make/gen/mbed_cortex-
[-m4/prj/micro_speech/mbed $ lstensorflow/tensorflow/lite/experimental/micro/tools/make/gen/mbed_cortex-
```

In your terminal type:

```
export PS1='\u@\h: '
```

For a more minimalist type:

```
export PS1='> '
```

3. Error: **"Requires make version 3.82 or later (current is 3.81)"**

If you encounter this error on a Mac, install brew and make by typing the following commands:

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"

brew install make
```

Note: You may use **gmake** instead of make to run your commands.

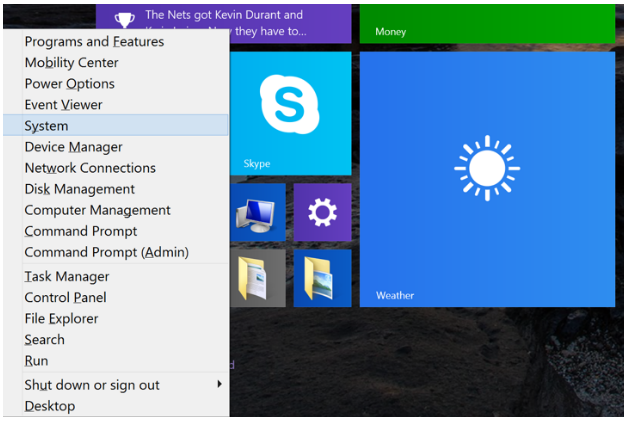
4. Error: -bash: mbed: command not found

If you encounter this error, try the following fixes.

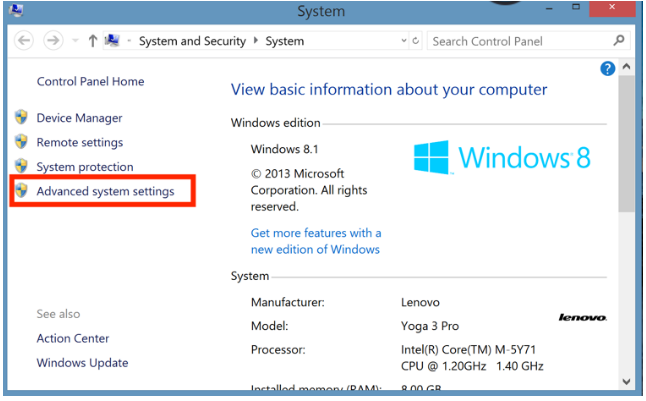
For Mac:
Include the location of mbed to your environment variable PATH

For Windows:
Make sure that you add Mbed CLI to Windows system PATH

1. Press Windows+X to open the Power Users menu, and then click **System** to launch System dialog.



On Windows 8, select **Advanced system settings**.



View basic information about your computer

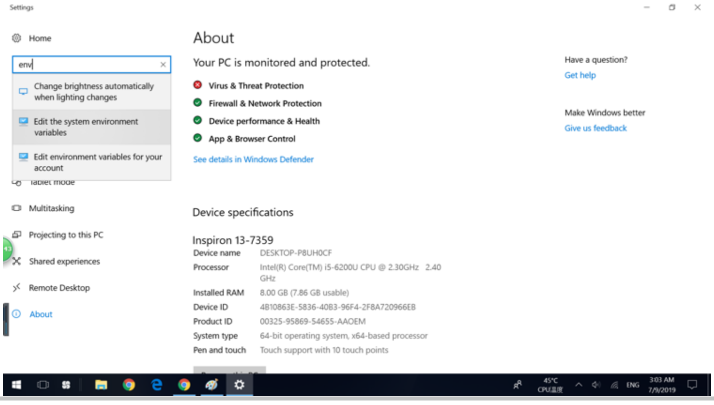
Windows edition

Windows 8.1
© 2013 Microsoft Corporation. All rights reserved.
Get more features with a new edition of Windows

System

Manufacturer: Lenovo
Model: Yoga 3 Pro
Processor: Intel(R) Core(TM) M-5Y71 CPU @ 1.20GHz 1.40 GHz
Installed memory (RAM): 8.00 GB

On Windows 10, type **env** in the search box, and select **Edit the system environment variables** option from the dropdown list.



About

Your PC is monitored and protected.

- Virus & Threat Protection
- Firewall & Network Protection
- Device performance & Health
- App & Browser Control

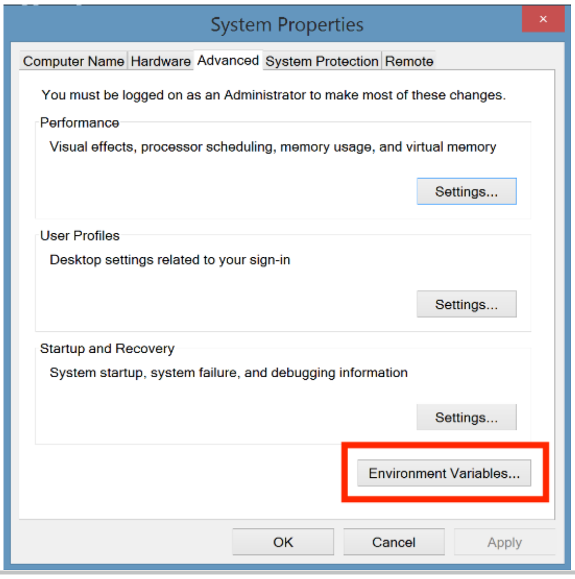
See details in Windows Defender

Device specifications

Inspiron 13-7359

Device name	DESKTOP-P8UHOCF
Processor	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz
Installed RAM	8.00 GB (7.86 GB usable)
Device ID	4B10863E-5836-40B3-96F4-2F8A720966EB
Product ID	00325-95869-54655-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	Touch support with 10 touch points

2. Click the **Environment Variable** button.



You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

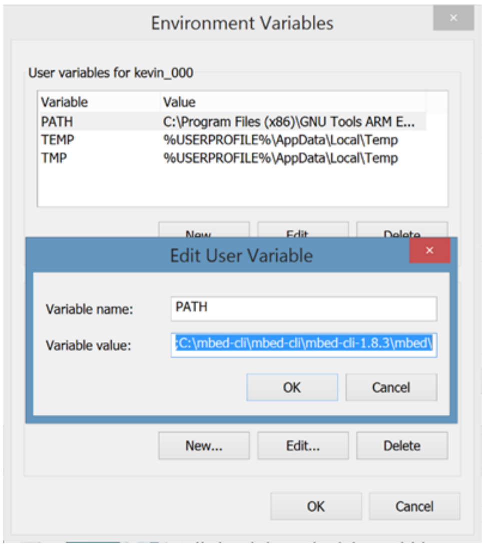
User Profiles

Desktop settings related to your sign-in

Startup and Recovery

System startup, system failure, and debugging information

3. Click the PATH variable and append mbed-cli to the PATH. By default, use `C:\mbed-cli\mbed-cli\mbed-cli-1.8.3\mbed\`.



User variables for kevin_000

Variable	Value
PATH	C:\Program Files (x86)\GNU Tools ARM E...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable name: PATH

Variable value: ;C:\mbed-cli\mbed-cli\mbed-cli-1.8.3\mbed\

Now, if you type **mbed** from a command prompt, you should be able to see mbed help commands.

Next steps

AI on tiny microcontrollers is possible, thanks to tools like TensorFlow Lite for Microcontrollers.

More optimized low-level kernels will be available as part of the CMSIS-NN open source project. This will allow developers to leverage Single Instruction Multiple Data (SIMD) instructions and receive an uplift in performance. SIMD instructions are available in Arm Cortex-M4, Cortex-M7, Cortex-M33 and Cortex-M35P processors.

Now that you have implemented your first machine learning application on a microcontroller, it is time to get creative.

To keep learning about ML with Arm and TensorFlow, here are some additional resources:

- View another in depth end-to-end TensorFlow from training to deployment guide
- Read a white paper on CMSIS-NN: Machine learning on Arm Cortex-M Microcontrollers