Build Arm Cortex-M voice assistant with Google TensorFlow Lite

Non-Confidential

Issue 1.0

ARM062-948681440-3282

Build Arm Cortex-M voice assistant with Google TensorFlow Lite

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	23 July 2019	Non-Confidential	First release

Non-Confidential Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

1 Overview

As an IoT developer, you might think of machine learning as a server-side technology. In the traditional view, sensors on your device capture data and send it to the cloud, where Machine Learning (ML) models on hefty machines make sense of it. A network connection is obligatory, and you are going to expect some latency, not to mention hosting costs.

But more and more, developers want to deploy their ML models to the edge, on IoT devices themselves. If you bring ML closer to your sensors, you remove your reliance on a network connection, and you can achieve much lower latency without a round trip to the server.

This is especially exciting for IoT, because less network utilization means lower power consumption. Also, you can better guarantee the security and privacy of your users, since you do not need to send data back to the cloud unless you know for sure that it is relevant.

In the following guide, you will learn how you can perform machine learning inference on an Arm Cortex-M microcontroller with TensorFlow Lite for Microcontrollers.

1.1 About TensorFlow Lite

TensorFlow Lite is a set of tools for running machine learning models on-device. TensorFlow Lite powers billions of mobile app installs, including Google Photos, Gmail, and devices made by Nest and Google Home.

With the launch of TensorFlow Lite for Microcontrollers, developers can run machine learning inference on extremely low-powered devices, like the Cortex-M microcontroller series. Watch the video to learn more about the announcement:

1.1.1 TensorFlow Lite - video

TensorFlow Lite (TF Dev Summit '19)

2 Getting started

2.2 Before you begin

Here is what you will need to complete the guide:

- Computer that supports Mbed CLI
- STM32F7 discovery kit board
- Mini-USB cable
- Python 2.7. Using pyenv is recommended to manage python versions.
- Serial port terminal programs. On Windows, examples are install programs like CoolTerm or Putty.

2.3 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 **STM32F7 discovery kit**.

We will deploy a sample application that uses the microphone on the STM32F7 and a TensorFlow machine learning model to detect the words “yes” and “no”.

To do this, we will show you how to complete the following steps:

1. Download and build the sample application
2. Deploy the sample to your STM32F7
3. Make some code changes to utilize the LCD display on the board
4. Use new trained models to recognize different words

3 Download and build the sample application

3.1 Install Arm toolchain and Mbed CLI

1. Download Arm cross compilation. Select the correct toolchain for the OS that your computer is running.

2. To build and deploy the application, we will use the **Mbed CLI**. We recommend that you install Mbed CLI with our installer. If you need more customization, you can perform a manual install. Although this is not recommended.

   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.

   If you do not already have Mbed CLI installed, download the installer:

   Mac installer

   Windows installer

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

   For Linux/Mac:

   ```
   mbed config -G GCC_ARM_PATH /bin
   ```

   For Windows:

   ```
   mbed config -G GCC_ARM_PATH C:\Program Files (x86)\GNU Tools ARM Embedded\8
   20180-q4-major\bin
   ```

Important: We recommend running the following commands from inside the Mbed CLI terminal that gets launched with the Mbed CLI Application. This is because it will be much quicker to set up, because it resolves all your environment dependencies automatically.

3.2 Build and compile micro speech example

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

```
git clone https://github.com/tensorflow/tensorflow.git
```

While you wait for the project to download, let's explore the project files on GitHub and learn how this TensorFlow Lite for Microcontrollers example works.

The code samples audio from the microphone on the STM32F7. The audio is run through a Fast Fourier transform to create a spectrogram. The spectrogram is then fed into a pre-trained machine learning model. The model uses a convolutional neural network to identify whether the sample represents either the command "yes" or "no", silence, or an unknown input. We will explore how this works in more detail later in the guide.

Here are descriptions of some interesting source files:

- disco_f746ng/audio_provider.cc captures audio from the microphone on the device.
- micro_features/micro_features_generator.cc: uses a Fast Fourier transform to create a spectrogram from audio.
- micro_features/tiny_conv_micro_features_model_data.cc. This file is the machine learning model itself, represented by a large array of unsigned char values.
- command_responder.cc is called every time a potential command has been identified.
- main.cc. This file is the entry point for the Mbed program, which runs the machine learning model using TensorFlow Lite for Microcontrollers.

After the project has downloaded, you can run the following commands to navigate into the project directory and build it:

For Linux/Mac:

```
cd tensorflow


make -f tensorflow/lite/experimental/micro/tools/make/Makefile TARGET=mbed
TAGS="disco_f746ng" generate_micro_speech_mbed_project
```

This will create a folder in `tensorflow/lite/experimental/micro/tools/make/gen/mbed_cortex-m4/prj/micro_speech/mbed` that contains the source and header files, Mbed driver files, and a README.

For Windows:

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

```
cd tensorflow/lite/experimental/micro/tools/make/gen/mbed_cortex-
m4/prj/micro_speech/mbed


mbed config root .


mbed deploy
```

TensorFlow requires C++ 11, so you will need to update your profiles to reflect this. Here is a short Python command that does that. Run it from the command line:

```
python -c 'import fileinput, glob;


for filename in glob.glob("mbed-os/tools/profiles/*.json"):


  for line in fileinput.input(filename, inplace=True):


    print line.replace("\"-std=gnu++98\"","\"-std=c++11\", \"-fpermissive\"")'
```

After that setting is updated, you can compile:

```
mbed compile -m DISCO_F746NG -t GCC_ARM
```

4 Project structure

While the project builds, we can look in more detail at how it works.

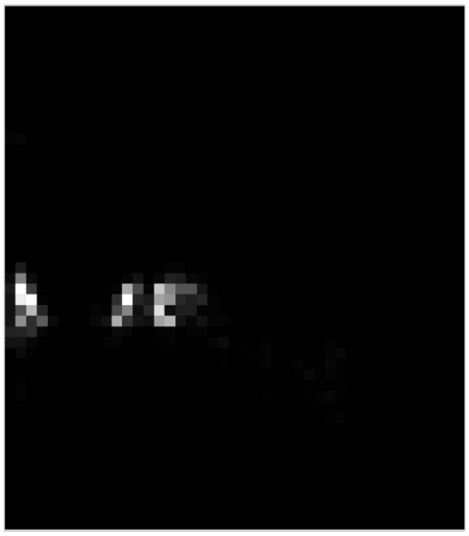
4.1 Convolutional neural networks

Convolutional networks are a type of deep neural network. These networks are designed to identify features in multidimensional vectors. The information in these vectors is contained in the relationships between groups of adjacent values.

These networks are usually used to analyze images. An image is a good example of the multidimensional vectors described above, in which a group of adjacent pixels might represent a shape, a pattern, or a texture. During training, a convolutional network can identify these features and learn what they represent. The network can learn how simple image features, like lines or edges, fit together into more complex features, like an eye, or an ear. The network can also learn, how those features are combined to form an input image, like a photo of a human face. This means that a convolutional network can learn to distinguish between different classes of input image, for example a photo of a person and a photo of a dog.

While they are often applied to images, which are 2D grids of pixels, a convolutional network can be used with any multidimensional vector input. In the example we are building in this guide, a convolutional network has been trained on a spectrogram that represents 1 second of audio bucketed into multiple frequencies.

The following image is a visual representation of the audio. The network in our sample has learned which features in this image come together to represent a "yes", and which come together to represent a "no".


Spectrogram for “yes”(**data**)

Spectrogram for “no” (**data**)

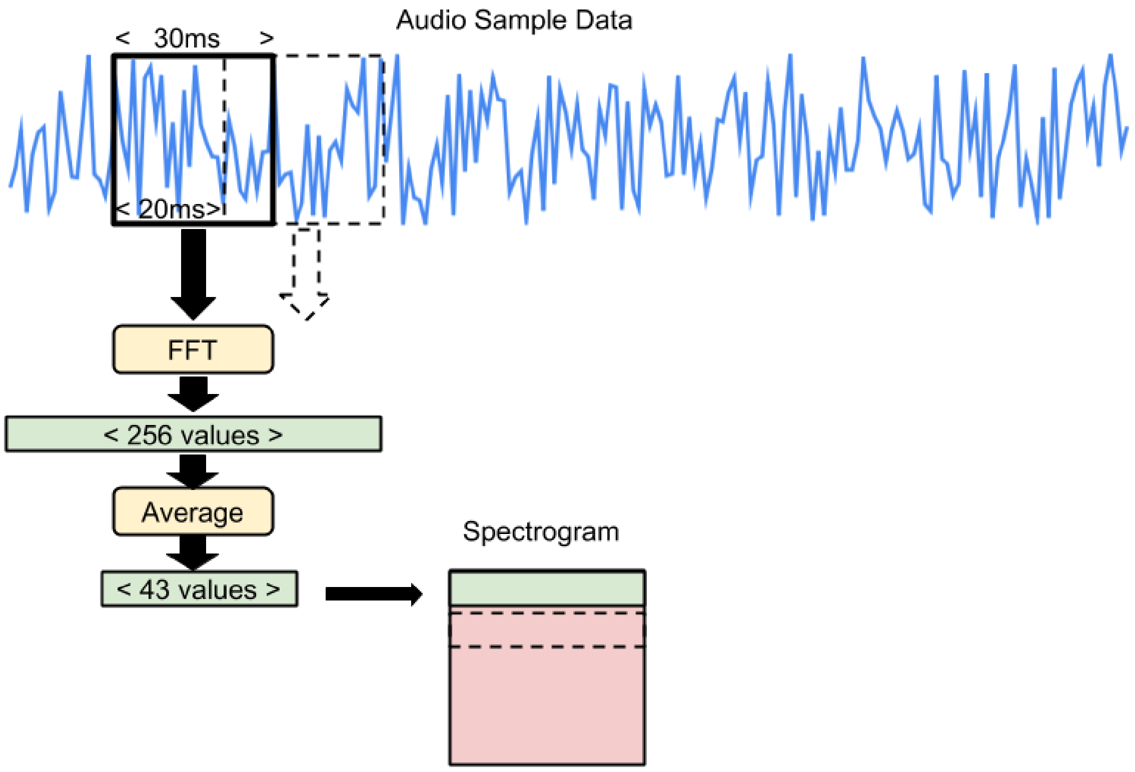
To generate this spectrogram, we use an interesting technique that is described in the next section.

4.2 Feature generation with Fast Fourier transform

In our code, each spectrogram is represented as a 2D array, with 43 columns and 49 rows. Each row represents a 30ms sample of audio that is split into 43 frequency buckets.

To create each row, we run a 30ms slice of audio input through a Fast Fourier transform. Fast Fourier transform analyzes the frequency distribution of audio in the sample and creates an array of 256 frequency buckets, each with a value from 0 to 255. These buckets are averaged together into groups of 6, leaving us with 43 buckets. The code in the file **micro_features/micro_features_generator.cc** performs this action.

To build the entire 2D array, we combine the results of running the Fast Fourier transform on 49 consecutive 30ms slices of audio, with each slice overlapping the last by 10ms. The following diagram should make this clearer:



You can see how the 30ms sample window is moved forward by 20ms each time until it has covered the full one-second sample. The resulting spectrogram is passed into the convolutional model.

4.3 Recognition and windowing

The process of capturing one second of audio and converting it into a spectrogram leaves us with something that our ML model can interpret. The model outputs a probability score for each category it understands (yes, no, unknown, and silence). The probability score indicates whether the audio is likely to belong to that category.

The model was trained on one-second samples of audio. In the training data, the word “yes” or “no” is spoken at the start of the sample, and the entire word is contained within that one-second. However, when this code is running, there is no guarantee that a user will begin speaking at the very beginning of our one-second sample.

If the user starts saying “yes” at the end of the sample instead of the beginning, the model might not be able to understand the word. This is because the model uses the position of the features within the sample to help predict which word was spoken.

To solve this problem, our code runs inference as often as it can, depending on the speed of the device, and averages all of the results within a rolling 1000ms window. The code in the file **recognize_commands.cc** performs this action. When the average for a given category in a set of predictions goes above the threshold, as defined in **recognize_commands.h**, we can assume a valid result.

4.3.1 Interpreting the results

The `RespondToCommand` method in **command_responder.cc** is called when a command has been recognized. Currently, this results in a line being printed to the serial port. Later in this guide, we will modify the code to display the result on the screen.

5 Deploy the sample to your STM32F7

In the previous section of this guide, we explained the build process for a keyword spotting example application.

Now that the build has completed, we will look in this section of the guide at how to deploy the binary to the STM32F7 and test to see if it works.

First, plug in your STM32F7 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.

Note: if you have skipped the previous steps, download the **binary file** to proceed.

For Linux/Mac:

Use the following command:

```
cp ./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin /Volumes/DIS_F746NG/
```

For Windows:

STM32F7 shows up as a new drive, for example, D. You can simply drag the generated binary at `.\BUILD\DISCO_F746NG\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.

5.1 Test keyword spotting

The program outputs 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. Although, 'screen' is not installed on Linux by default, 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.

Try saying the word "yes" several times. You should see some output like the following:

```
Heard yes (208) @116448ms


Heard unknown (241) @117984ms


Heard no (201) @124992ms
```

Congratulations! You are now running a machine learning model that can recognize keywords on an Arm Cortex-M7 microcontroller, directly on your STM32F7.

6 Extend the program

Part of what makes Mbed exciting is how easy it is to pull in dependencies to use in your program. In the next section of this guide, we will show how you can extend the program to make use of the LCD display on the STM32F7.

First, if the LCD driver library has not already been added, we will need to run a command to add the library to our project.

```
mbed add http://os.mbed.com/teams/ST/code/LCD_DISCO_F746NG/
```

Once the library has downloaded, we will work with the code and start making changes.

As mentioned, the file **command_responder.cc** is a convenient integration point. From here, we can extend the program to behave differently depending on which command is detected.

In this case, we will be extending the program to display different information on the LCD display for each command.

In `tensorflow/lite/experimental/micro/examples/micro_speech/` open the following command in your text editor:

```
command_responder.cc
```

After the license comments, you should see the following code:

```
#include
"tensorflow/lite/experimental/micro/examples/micro_speech/command_responder.h"



// The default implementation writes out the name of the recognized command

// to the error console. Real applications will want to take some custom

// action instead, and should implement their own versions of this function.

void RespondToCommand(tflite::ErrorReporter* error_reporter,

                      int32_t current_time, const char* found_command,

                      uint8_t score, bool is_new_command) {

  if (is_new_command) {
    error_reporter->Report("Heard %s (%d) @%dms", found_command, score,

                           current_time);

  }

}
```

In the preceding code, you can see how the argument `is_new_command` indicates whether a new command has been detected. Currently, when a command is detected, the program just writes a debug line to the serial connection. Let's make things more interesting.

First, we will add a couple of lines to include and instantiate the LCD driver. Next, we will add some logic that checks the `found_command` pointer to determine which command was detected. In each case, we call `lcd.Clear()`. This call, clears the LCD and sets the background color to an appropriate value. We then draw a line of text to the LCD.

```
#include
"tensorflow/lite/experimental/micro/examples/micro_speech/command_responder.h"

#include "LCD_DISCO_F746NG.h"

LCD_DISCO_F746NG lcd;

// The default implementation writes out the name of the recognized command

// to the error console. Real applications will want to take some custom

// action instead, and should implement their own versions of this function.

void RespondToCommand(tflite::ErrorReporter* error_reporter,

                      int32_t current_time, const char* found_command,

                      uint8_t score, bool is_new_command) {

  if (is_new_command) {

    error_reporter->Report("Heard %s (%d) @%dms", found_command, score,

                           current_time);

    if(*found_command == 'y') {

      lcd.Clear(0xFF0F9D58);
      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard yes!", CENTER_MODE);

    } else if(*found_command == 'n') {

      lcd.Clear(0xFFDB4437);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard no :(", CENTER_MODE);

    } else if(*found_command == 'u') {

      lcd.Clear(0xFFF4B400);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard unknown", CENTER_MODE);

    } else {

      lcd.Clear(0xFF4285F4);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard silence", CENTER_MODE);

    }

  }

}
```

Once you have updated the file, save it and then go back in the tensorflow directory

```
cd tensorflow/lite/experimental/micro/tools/make/gen/mbed_cortex-
m4/prj/micro_speech/mbed
```

and run the following command to rebuild the project:

```
mbed compile -m DISCO_F746NG -t GCC_ARM
```

Next, copy the binary to the USB storage on your device, using the same method that you used earlier.

Once the board finishes flashing, you should be able to say “yes” and “no” and see the LCD display "Heard yes!", as you can see in the following image:

Extra: The arguments passed to `RespondToCommand` contain interesting information about the recognition result. For example, `score` contains a number that indicates the probability that the result is correct. You might want to modify the code so that the display varies based on this information.

It is easy to change the behavior of our program by writing code, but is it difficult to modify the machine learning model itself? The answer is no, and the next section of this guide, **Retrain the machine learning model**, will show you how.

7 Retrain the machine learning model

The model that we are using for speech recognition was trained on a dataset of one-second spoken commands called the **Speech Commands Dataset**. The dataset includes examples of the following ten different words:

yes, no, up, down, left, right, on, off, stop, go

While the model in this sample was originally trained to recognize "yes" and "no", the TensorFlow Lite for Microcontrollers source contains scripts that make it easy to retrain the model to classify any other combination of these words.

We are going to use another pre-trained model to recognize "on" and "off", instead. If you are interested in the full flow including the training of the model refer to the **Supplementary information: model training** section of this guide.

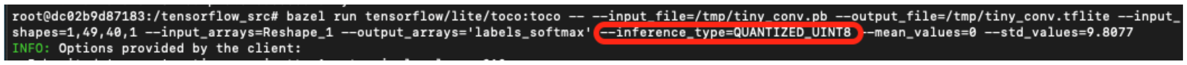
To build our new ML application we will now follow these steps:

1. Download a pretrained model that has been trained and frozen using TensorFlow.
2. Look at how the TensorFlow model gets converted to the TensorFlow Lite format.
3. Convert the TensorFlow Lite model into a C source file.
4. Modify the code and deploy to the device.

Note: Building TensorFlow and training the model will each take a couple of hours on an average computer. We will not perform this at this stage. For a full guide on how to do this, refer to the **Supplementary information: model training** section in this guide.

7.1 Convert the model

Starting from the trained model to obtain a converted model that can run on the controller itself, we need to run a conversion script: the **TensorFlow Lite converter**. This tool uses clever tricks to make our model as small and efficient as possible, and to convert it to a TensorFlow Lite FlatBuffer. To reduce the size of the model, we used 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, as you can see in the following command:



```
root@dc02b9d87183:/tensorflow_src# bazel run tensorflow/lite/toco:toco -- --input_file=/tmp/tiny_conv.pb --output_file=/tmp/tiny_conv.tflite --input_
shapes=1,49,40,1 --input_arrays=Reshape_1 --output_arrays='labels_softmax' --inference_type=QUANTIZED_UINT8 --mean_values=0 --std_values=9.8077
INFO: Options provided by the client:
```

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

To save time, we will skip this step and instead download the **tiny_conv.tflite** file from the guide page.

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 tiny_conv.tflite > tiny_conv_micro_features_model_data.cc
```

Next, we need to update `tiny_conv_micro_features_model_data.cc` so that it is compatible with our code. First, open the file. The top two lines should look similar to the following code, although the exact variable name and hex values may be different:

```
unsigned char tiny_conv_tflite[] = {

  0x18, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x0e, 0x00,
```

You need to add the `include` from the following snippet and change the variable declaration without changing the hex values:

```
#include
"tensorflow/lite/experimental/micro/examples/micro_speech/micro_features/tiny_con
v_micro_features_model_data.h"

const unsigned char g_tiny_conv_micro_features_model_data[] = {

  0x18, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x0e, 0x00,
```

Next, go to the very bottom of the file and find the unsigned int variable.

```
unsigned int tiny_conv_tflite_len = 18216;
```

Change the declaration to the following code, but do not change the number assigned to it, even if your number is different from the one in this guide.

```
const int g_tiny_conv_micro_features_model_data_len = 18216;
```

Finally, save the file, then copy the `tiny_conv_micro_features_model_data.cc` file into the `tensorflow/tensorflow/lite/experimental/micro/tools/make/gen/mbed_cortex-m4/prj/micro_speech/mbed/tensorflow/lite/experimental/micro/examples/micro_speech/micro_features`directory.

7.2 Modify the device code

If you build and run your code now, your device should respond to the words "stop" and "go". However, the code was written to assume that the words are "yes" and "no". Let's update the references and the user interface so that the appropriate words are printed.

First, go to the following directory:

```
tensorflow/lite/experimental/micro/examples/micro_speech/
```

and open the file:

```
micro_features/micro_model_settings.cc
```

You will see the following category labels:

```
const char* kCategoryLabels[kCategoryCount] = {

    "silence",

    "unknown",

    "yes",

    "no",

 };
```

The code uses this array to map the output of the model to the correct value. Because we specified our `wanted_words` as "on, off"in the training script, we should update this array to reflect these words in the same order. Edit the code so it appears as follows:

```
const char* kCategoryLabels[kCategoryCount] = {


    "silence",


    "unknown",


    "on",


    "off",


 };
```

Next, we will update the code in `command_responder.cc` to reflect these new labels, modifying the if statements and the `DisplayStringAt` call:

```
void RespondToCommand(tflite::ErrorReporter* error_reporter,

                      int32_t current_time, const char* found_command,

                      uint8_t score, bool is_new_command) {

  if (is_new_command) {

    error_reporter->Report("Heard %s (%d) @%dms", found_command, score,

                           current_time);

    if(strcmp(found_command, "on") == 0) {

      lcd.Clear(0xFF0F9D58);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard on", CENTER_MODE);

    } else if(strcmp(found_command, "off") == 0) {

      lcd.Clear(0xFFDB4437);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard off", CENTER_MODE);

    } else if(strcmp(found_command, "unknown") == 0) {

      lcd.Clear(0xFFF4B400);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard unknown", CENTER_MODE);

    } else {

      lcd.Clear(0xFF4285F4);

      lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard silence", CENTER_MODE);

    }

  }

 }
```

Now that we have updated the code, go back to the `mbed` directory:

```
cd
<path_to_tensorflow>/tensorflow/lite/experimental/micro/tools/make/gen/mbed_corte
x-m4/prj/micro_speech/mbed
```

and run the following command to rebuild the project:

```
mbed compile -m DISCO_F746NG -t GCC_ARM
```

Finally, copy the binary to the USB storage of the device, using the same method that you used earlier. You should now be able to say “stop” and “go” to update the display.

8 Troubleshooting

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

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

Purge the cache with the following command:

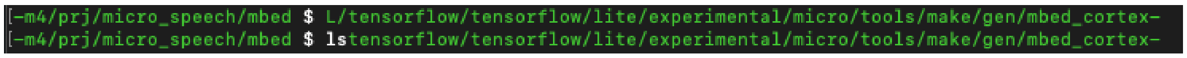
```
mbed cache purge
```

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

```
rm -rf BUILD
```

8.2 Error: Prompt wrapping around line

If your terminal is wrapping your text as show 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='> '
```

8.3 Error: "Requires make version 3.82 or later (current is 3.81)"

If you encounter this error, install the brew and make by typing the following code:

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

brew install make
```

Note: On a Mac, you might have to use gmake instead of make, to run your commands.

8.4 Error: -bash: mbed: command not found

If you encounter this error, try the following fixes.

For Mac:

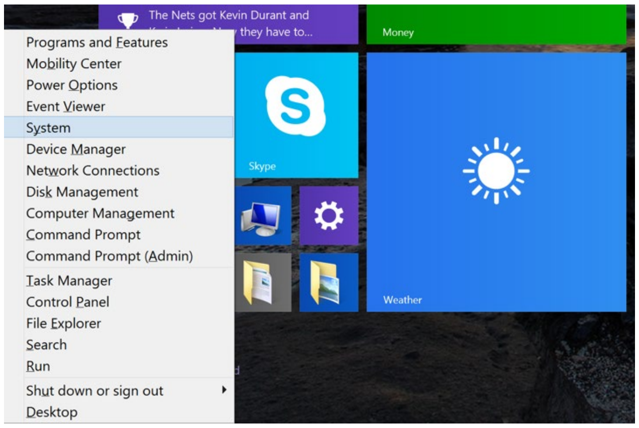
We recommend using the **installer** and running the downloaded Mbed CLI App. This app will automatically launch a shell with all the dependencies solved for you.

If installed manually, make sure to follow these **instructions**.

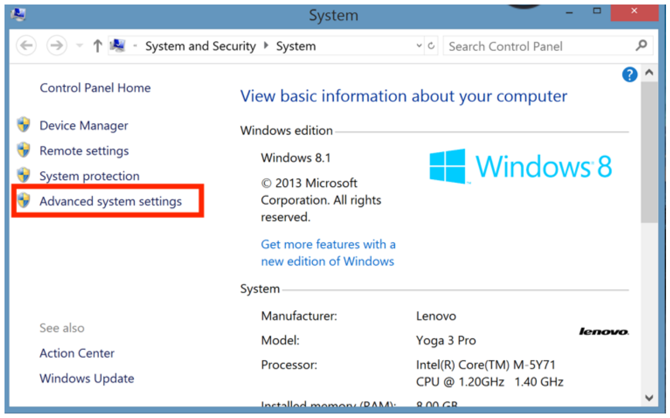
For Windows:

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

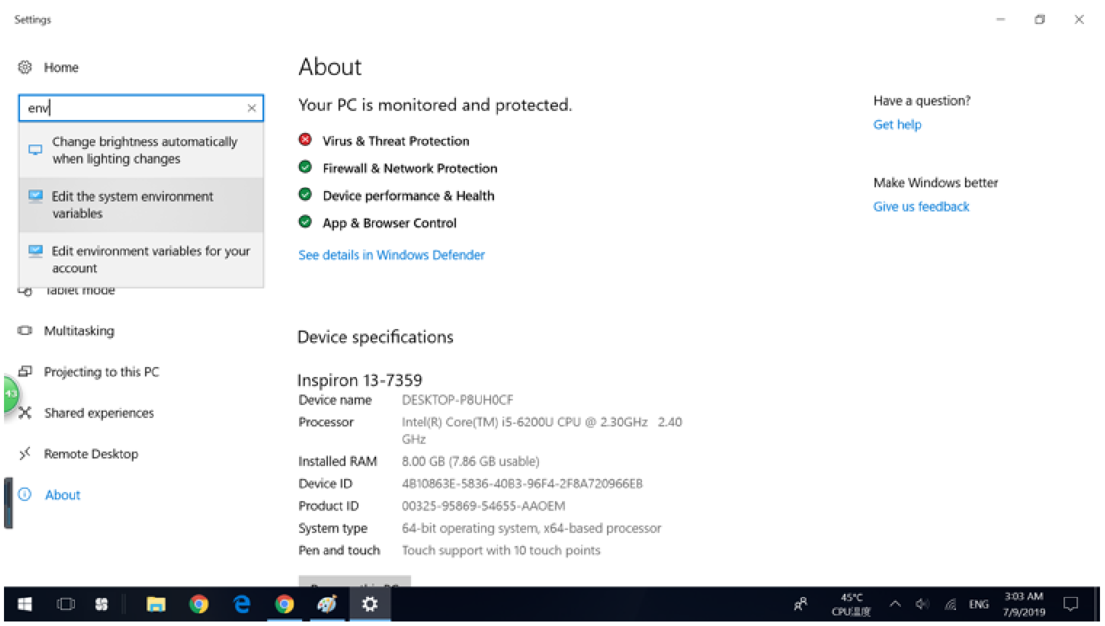
© 2013 Microsoft Corporation. All rights reserved.

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

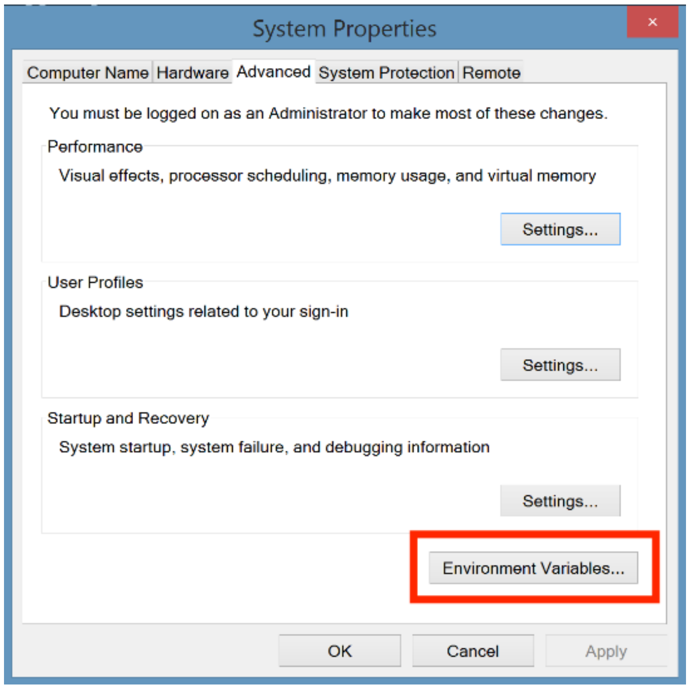
See details in Windows Defender

Device specifications

Inspiron 13-7359

Device name	DESKTOP-P8UH0CF
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.



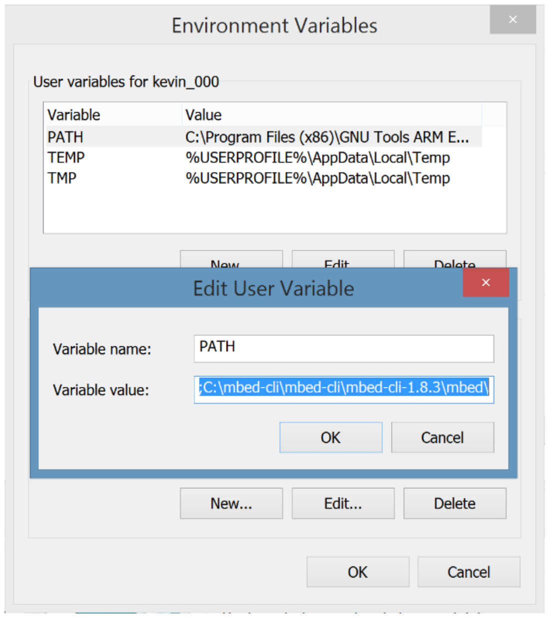
You must be logged on as an Administrator to make most of these changes.

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\`.


Environment Variables

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

Edit User Variable

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.

9 Next steps

Until recently, AI on tiny microcontrollers was deemed impossible. Now, thanks to tools like Mbed and TensorFlow Lite for Microcontrollers, it is not only possible, it is easy and within the reach of every open source software developer, maker, and start-up.

Soon, 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**

Learn how to use the **OpenMV camera for ML** applications

Explore **Arm NN** for ML on other Arm processors and GPUs

Share your projects using the hashtag #TinyML, and tag @Arm.

10 Supplementary information: model training

10.1 Prepare to build TensorFlow

The code that we used to train our voice model currently depends on some experimental operations that are only available when building TensorFlow from source. We will have to build this.

The easiest way to build TensorFlow from source is to use Docker. Docker is a tool that enables you to run tasks on a virtual machine that is isolated from the rest of your computer, which makes dependency management easier. TensorFlow provides a custom docker image that can be used to build the toolchain from source.

The first step is to follow the instructions to **install Docker**.

Once Docker is installed, run the following command to test that it works:

```
docker run hello-world
```

You should see a message starting with “Hello from Docker!”.

Once you have installed Docker, use the following command to install the latest TensorFlow development Docker image. This contains the TensorFlow source:

```
docker pull tensorflow/tensorflow:devel
```

Visit **TensorFlow’s Docker images** for more information.

Now, run the following command to connect to your Docker instance and open a shell:

```
docker run -it -w /tensorflow_src -v $PWD:/mnt tensorflow/tensorflow:devel bash
```

You should now be on the command line of the TensorFlow Docker image, in the directory that contains the TensorFlow source code. You should issue the following commands to fetch the very latest code and install some required Python dependencies:

```
git fetch

git rebase origin master

pip install -U --user pip six numpy wheel setuptools mock tensorflow_estimator

pip install -U --user keras_applications==1.0.6 --no-deps

pip install -U --user keras_preprocessing==1.0.5 --no-deps
```

We now need to configure the build. Running the following command from the root of the TensorFlow repo will start configuration. You will be asked a series of questions. Just hit **return** at every prompt to accept the default option.

```
./configure
```

Once configuration is done, we are ready to go.

10.2 Train the model

The following command will build TensorFlow from source and then start training.

Note: The build will take several hours. To save time you can download the **tiny_conv.pb** and skip to the following section on this page: Converting the trained model section below.

```
bazel run -c opt --copt=-mavx2 --copt=-mfma
tensorflow/examples/speech_commands:train -- --model_architecture=tiny_conv --
window_stride=20 --preprocess=micro --wanted_words="on,off" --
silence_percentage=25 --unknown_percentage=25 --quantize=1
```

Notice how the `wanted_words` argument contains the words "on" and "off". You can add any words that you like from the available ten to this field, separated by commas.

On older CPUs, you can leave out the `--copt` arguments. These arguments are there to accelerate training on chips that support the extensions.

The process will take a couple of hours. While you wait, you can take a look at a more detailed overview of the **speech model** that we are training.

10.3 Freeze the model

We need to perform a few extra steps to be able to run the model directly on our microcontroller. With your trained model, you should run the following command to create a single "frozen graph" file that represents the trained model.

Note: we need to provide our `wanted_words` argument again:

```
bazel run tensorflow/examples/speech_commands:freeze -- --
model_qqqcarchitecture=tiny_conv --window_stride=20 --preprocess=micro --
wanted_words="on,off" --quantize=1 --output_file=/tmp/tiny_conv.pb --
start_checkpoint=/tmp/speech_commands_train/tiny_conv.ckpt-18000
```

You now have a file, `/tmp/tiny_conv.pb`, that represents the model. This is great, but since we are deploying the model on a tiny device, we need to do everything that we can to make it as small and simple as possible.

10.4 Converting the model to the TensorFlow Lite format

To obtain a converted model that can run on the microcontroller itself, we need to run a conversion script, **TensorFlow Lite converter**. This tool uses clever tricks to make our model as small and efficient as possible and to convert it to a TensorFlow Lite FlatBuffer. To reduce the size of the model we used 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 will not only reduce the size of the network, but also avoid floating-point computations, that are more computationally expensive.

Run the following command to perform the conversion:

```
bazel run tensorflow/lite/toco:toco -- --input_file=/tmp/tiny_conv.pb --
output_file=/tmp/tiny_conv.tflite --input_shapes=1,49,40,1 --
input_arrays=Reshape_1 --output_arrays='labels_softmax' --
inference_type=QUANTIZED_UINT8 --mean_values=0 --std_values=9.8077
```

Instead you should now have a `/tmp/tiny_conv.tflite` file. We now have to copy this file from our Docker instance to the host machine. To do this, run the following command:

```
cp /tmp/tiny_conv.tflite /mnt
```

This will place the file in the directory that you were in when you first ran the command to connect to Docker. For example, if you ran the command from `~/Desktop`, the file will be at `~/Desktop/tiny_conv.tflite`.

To leave the Docker instance and get back to your regular command line, type the following:

```
exit
```