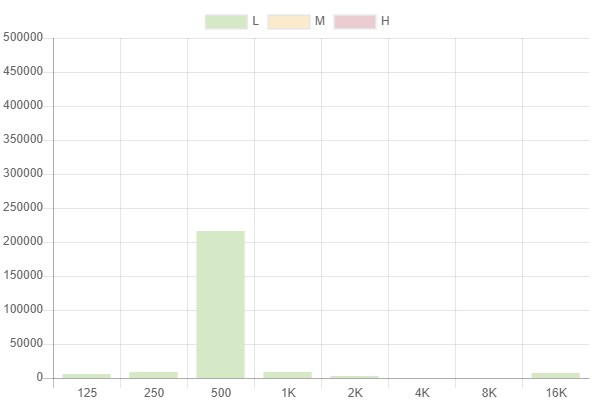
**TensorFlow Lite for MicroControllers 热词唤醒**

  TensorFlow Lite for MicroControllers 实现了人工智能在无操作系统的单片机上直接运行，这使得我们可以直接使用语音去控制单片机做出一些有意思的操作。不仅仅是人类的语言，我们甚至还能用它来识别鸟语，或者根据声音判断机器是否故障等等一切与声音相关的识别操作。

  本节课将学习TensorFlow Lite for MicroControllers的相关知识，尝试体验一整套微控制器上运行人工智能的开发流程，最终制作一个热词唤醒的声控灯。



**你需要准备：**

  microUSB 数据线一根

  TinyML开发板一块

  win10 系统笔记本电脑

  BlackWalnut Labs 访问账号

  特制的Arduino开发环境

**音频数据采集**

**Arduino的Hello World**

  我们打开Arduino，选择开发板为ESP32 Dev Module,并且选择端口号为实际开发板的端口号。

  配置完成后，我们可以发现它已经为我们预置了两个函数，一个是setup,另一个是loop。其中setup是在开发板上电后运行的第一个程序，运行且只运行一遍。loop是一个死循环，在上电之后会一遍又一遍反复运行其中的代码。



  接下去我们就来写一个Hello World，这也将会是Arduino为数不多的调试方法之一。我们在接下去的开发中都会使用这种方法来进行调试。

  我们会先在这两个函数之外定义一个全局变量count,接着在Setup里面设置串口波特率为115200，使用串口我们能够将开发板内部的数据向外部发送。

  在loop函数里面我们会使用Serial.print或者Serial.println将Hello World 和count 数据使用串口进行输出，并使用delay函数控制输出频率，使其每隔1000毫秒输出一次数据。

int conut=0;

void setup() {

Serial.begin(115200);

}

void loop() {

conut=conut+1;

Serial.print("Hello ");

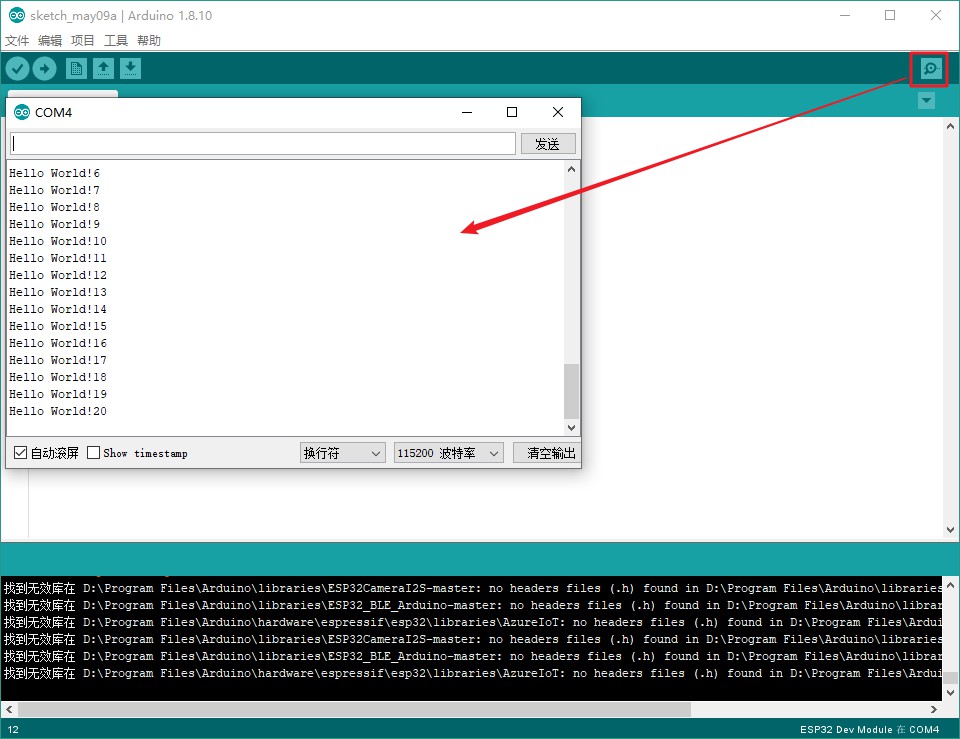
Serial.print("World!");

Serial.println(conut);

delay(1000);

}

  我们可以使用串口监视器进行查看实时输出的数据,来了解此时开发板的内存数据情况，但是有一点要注意，这种调试方法，很占CPU的资源。



**全向麦克风音频采样**

  在开发板上有一个全向麦克风，这个全向麦克风所使用的是I2S（Inter-IC Sound）协议，I2S协议是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专门用于音频设备之间的数据传输。

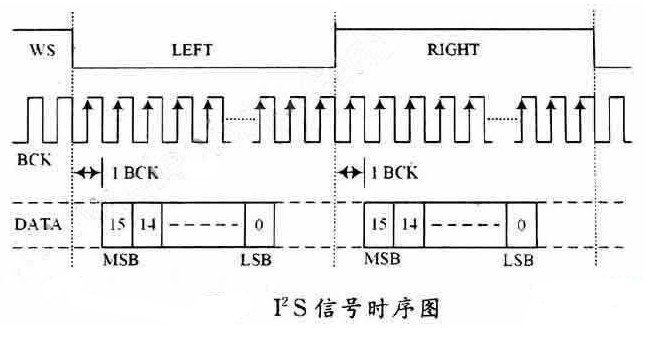


  I2S协议有3个主要信号，串行时钟（SCK），帧时钟（WS），串行数据（SD）。

  当音频数据每传递一位的时候，串行时钟（sck）都会对应发出一个脉冲。以包装这一位数据的真实有效。

  帧时钟用于区分左右声道，0表示正在传输的数据是左声道的，1表示正在传输的数据是右声道的数据。

  串行数据主要用于描述真实数据，它在传输过程中使用二进制补码表示的音频数据。



  它们在开发板上与引脚的连接关系是WS接18号引脚，SD接32号引脚，SCK接14号引脚。

  那么接下去我们就使用I2S协议读取音频信号的数据，首先导入I2S的库。

//添加↓

#include <driver/i2s.h>

//添加↑

void setup(){

Serial.begin(115200);

}

void loop(){

//清空之前代码

}

  然后我们设置好串行时钟、帧时钟、串行数据这些信号线的引脚号，接着配置I2S的端口号，ESP32（开发板内核）最多允许我们使用2路的I2S。我们这里使用的端口为I2S\_NUM\_1。

#define I2S\_WS 18

#define I2S\_SD 32

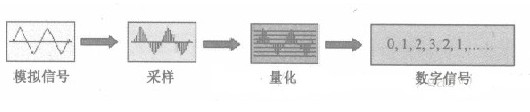
#define I2S\_SCK 14

#define I2S\_PORT I2S\_NUM\_1

  等这些调节都完善了以后，我们先不急着继续写代码，我们先来了解一下什么是采样。

  我们知道计算机内部是使用0或者1来离散地表示数据的，但是纯粹的模拟信号是无法完全不失真地用数据来描述的，我们必然需要经过各种处理，最终获得用于描述现象的理想数据。

  如下图所示，在最简单的获得模拟量数据的过程中我们至少需要用采样和量化两个过程来最终将外界模拟信号转化为数据。



  采样和量化两个过程中分别需要注意的要素是采样频率和量化位数。

  首先是采样频率,模拟信号不仅仅在每一个独立的信号表示上是连续的(比如灯的亮暗，是可以无限细分下去的)，而且还在时间轴上是连续的，灯从暗到亮的过程由于时间段可以被无限细分，因此我们也很难去用数据描述在某个无限不循环小数时间点上灯的具体亮度。

  采样频率就是将时间进行切片。不去采集所有的时间点上的所有数据，因为根本采集不完，而是隔一段时间采集一次。当然这也直接抛弃掉了两次采样时间间隔内所有其他的样本数据。但是没关系我们可以通过调节采样频率来获取相对合理的样本数据。

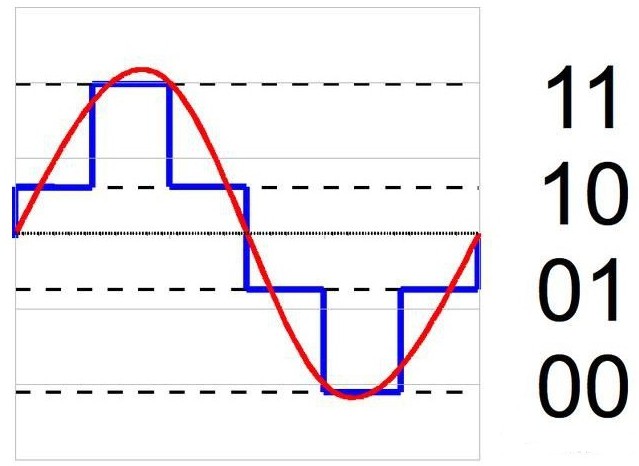
  比如我们对声音的采样,标准的CD唱片是使用44.1kHz采样频率进行采样的。因为我们人类耳朵能够感觉到的最高频率为20kHz，然后根据奈奎斯特定理，只要高于最高频率的两倍，就可以实现声音不失真的还原。

  因此要满足人耳的听觉要求，则需要至少每秒进行4万次采样，用40kHz表达，所以我们的CD才将频率定义为44.1kHZ。采样频率不用太高，只要合理，能满足需求即可。

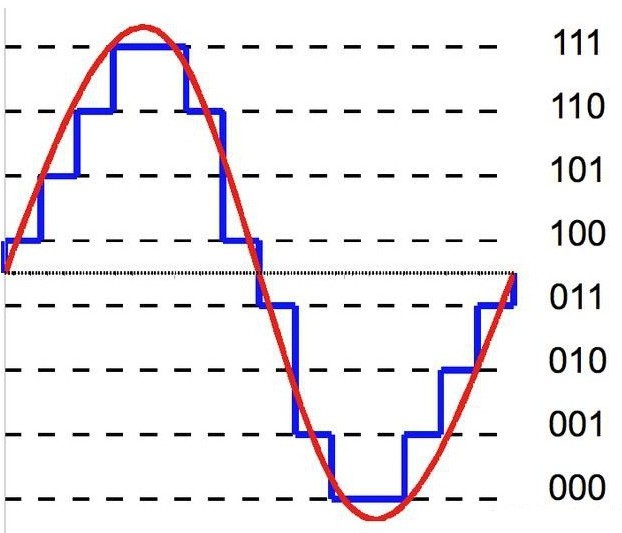


  模拟信号经过采样，还不能最终转化为数据的形式，还需要一个叫做量化的过程。

  比如下图就是一个2bit量化的过程，2bit的模数转换采样意味着我们只能取值4次，只能把红色模拟信号重建成蓝色线这个样子。那假如说我们通过2bit去量化世界上所有的声音，那么最终只能得到四种声音，并且与实际的模拟量声音效果有比较大的出入。会带来很明显的噪声，我们称之为量化噪声。



  然后我们可以将量化位数增加到3bit,8次取值看起来是不是比4次好多了呢？



  在了解了采样频率和量化位数两个概念后，我们继续编程，完成声音信号的采样。

  我们需要先对I2S总线进行初始化,我们编写i2s\_init函数对I2S函数进行初始化处理， 这里的参数我们主要关心sample\_rate、bits\_per\_sample两个值。这就分别对应着采样频率和量化范围

//保留之前代码

void setup(){

//保留之前代码

}

void loop(){

//保留之前代码

}

//添加↓

void i2s\_init(){

const i2s\_config\_t i2s\_config = {

.mode = i2s\_mode\_t(I2S\_MODE\_MASTER | I2S\_MODE\_RX),

.sample\_rate = 8000,

.bits\_per\_sample = i2s\_bits\_per\_sample\_t(16),

.channel\_format = I2S\_CHANNEL\_FMT\_ONLY\_LEFT,

.communication\_format = i2s\_comm\_format\_t(I2S\_COMM\_FORMAT\_I2S | I2S\_COMM\_FORMAT\_I2S\_MSB),

.intr\_alloc\_flags = 0, // default interrupt priority

.dma\_buf\_count = 8,

.dma\_buf\_len = 64,

.use\_apll = true

};

i2s\_driver\_install(I2S\_PORT, &i2s\_config, 0, NULL);

}

//添加↑

  我们先将sample\_rate设置为8000，8000Hz一般用于电话音频的采样上，节约数据空间的同时能够提供较为清晰的音频内容信息。

  然后bits\_per\_sample量化位数我们使用16bit量化。这样出来的数据就是-32767到32768之间，其中一位表示正负。

  对与其他参数的用法和说明，这里不做详细展开，我们都可以去[这份手册](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/i2s.html)中进行详细说明的查询。最后顺便一提，在其他参数中，有点意思的是这个dma\_buf。

  完成了I2S初始化之后，需要编写i2s\_setpin函数定义三个引脚，正式与硬件进行通信。

  这里data\_out\_num表示ESP32需要向外发送数据的引脚，由于在这个实验中占时不涉及到ESP32向麦克风发送数据，所以这里填写为-1。

//保留之前代码

void setup(){

//保留之前代码

}

void loop(){

//保留之前代码

}

void i2s\_init(){

//保留之前代码

}

//添加↓

void i2s\_setpin(){

const i2s\_pin\_config\_t pin\_config = {

.bck\_io\_num = I2S\_SCK,

.ws\_io\_num = I2S\_WS,

.data\_out\_num = -1,

.data\_in\_num = I2S\_SD

};

i2s\_set\_pin(I2S\_PORT, &pin\_config);

}

//添加↑

  然后我们在setup函数中对这些函数进行调用。

//保留之前代码

void setup() {

Serial.begin(115200);

//添加↓

i2s\_init();

i2s\_setpin();

delay(1000);

//添加↑

}

//保留之前代码

  然后我们在loop函数中使用i2s\_pop\_sample函数对声音进行持续的采样。这里我们需要定义一个类型为int16\_t的变量sample来保存当前采样点的具体值。

//保留之前代码

void loop() {

//添加↓

int16\_t sample=0;

int bytes=i2s\_pop\_sample(I2S\_PORT,(void \*)&sample,portMAX\_DELAY);

if(bytes>0){

Serial.println(sample);

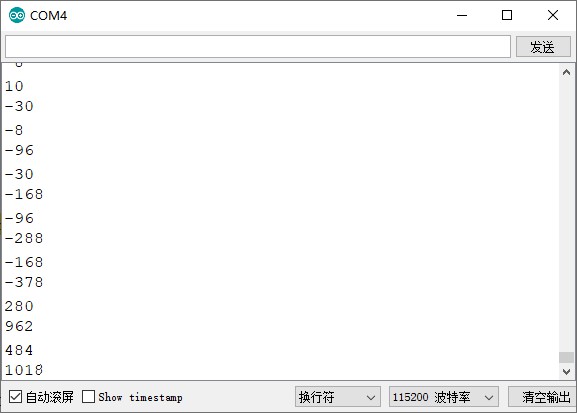
}

//添加↑

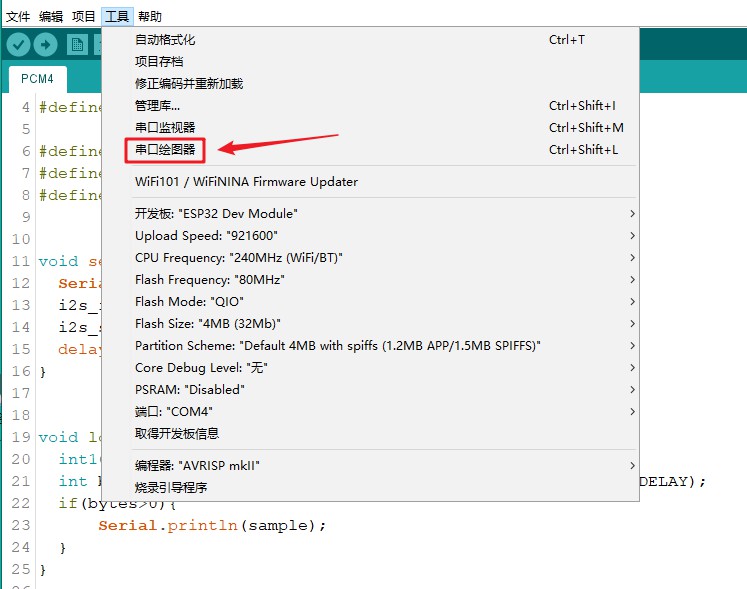
}

//保留之前代码

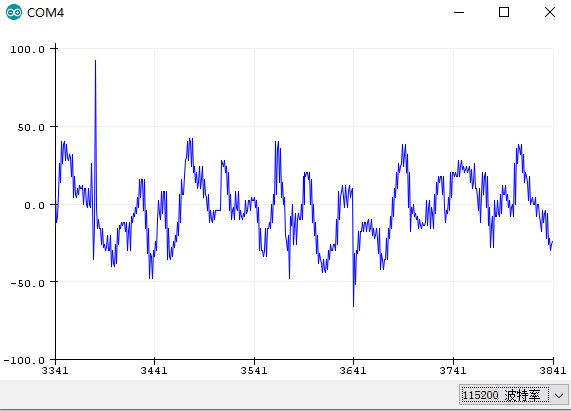
  我们可以使用串口来打印实时的数据。



  也可以使用串口绘图器来观察信号。



  这时我们可以对着麦克风说几句话，观察曲线的变化。



  到目前为止，我们已经可以将声音使用数字的方式记录下来，这种方式被称为PCM（Pulse Code Modulation），脉冲编码调制。

# 录音机

  我们经过采样采集到的PCM信号，要怎么样证明它是正确的呢？因为到目前为止，我们只是获取到了一堆我们认为有实际意义的数字，那么这堆数字是否就是我们所期望的数字呢？唯一能证明的方式就是这堆数字还原成音频，然后用耳朵辨别。

## 采集两秒的PCM信号

  我们目前的采样频率是8000Hz，在这个采样频率下为了采集到2秒的音频信号，我们总共要采集16000个数据。

  因为串口输出是需要占用一定的时间，会影响采样，所以我们就先设置一个长度为16000的int16\_t的数组，一旦数据被采集到，先不把它直接通过串口打印出来，而是存放到这个数组当中去。等到采集完毕，再将它们全部打印出来。

//保留之前代码

#define I2S\_SCK 14

//添加↓

const int BLOCK\_SIZE = 16000;

int16\_t samples[BLOCK\_SIZE];

//添加↑

void setup() {

//保留之前代码

}

//保留之前代码

  然后我们再定义一个samples\_count变量来统计当前已经记录的信号数据的个数。并且等到samples\_count为16000时停止采样，进行所有数据的打印。

//保留之前代码

int16\_t samples[BLOCK\_SIZE];

//添加↓

int samples\_count=0;

//添加↑

void setup() {

//保留之前代码

}

void loop() {

int16\_t sample=0;

int bytes=i2s\_pop\_sample(I2S\_PORT,(void \*)&sample,portMAX\_DELAY);

//删去这些代码↓

// if(bytes>0){

// Serial.println(sample);

// }

//删去这些代码↑

//添加↓

if(bytes>0&&samples\_count<BLOCK\_SIZE){

samples[samples\_count]=sample;

samples\_count=samples\_count+1;

}else if(samples\_count==BLOCK\_SIZE){

for(int i=0;i<BLOCK\_SIZE;i++){

Serial.print(samples[i]);

Serial.print(", ");

}

samples\_count=samples\_count+1;

}

//添加↑

}

  然后我们现在就可以获取到16000个采样点的数据，为了确保数据的正确性，我们还可以做一个操作。理论上采集16000个点需要的时间是2秒，那么实际上是否是2秒呢？

  我们需要添加一个start\_time来记录当前第一个点的时刻，再记录采集完16000个点的时刻，两个数相减就是采集的总耗时数。

//保留之前代码

int samples\_count=0;

int start\_time=0;

void setup() {

//保留之前代码

}

//保留之前代码

void loop() {

//保留之前代码

int bytes=i2s\_pop\_sample(I2S\_PORT,(void \*)&sample,portMAX\_DELAY);

if(bytes>0&&samples\_count<BLOCK\_SIZE){

//添加↓

if(samples\_count==0){

start\_time=millis();

}

//添加↑

samples[samples\_count]=sample;

//保留之前代码

}else if(samples\_count==BLOCK\_SIZE){

//添加↓

int fin\_time=millis()-start\_time;

Serial.println(fin\_time);

//添加↑

for(int i=0;i<BLOCK\_SIZE;i++){

//保留之前代码;

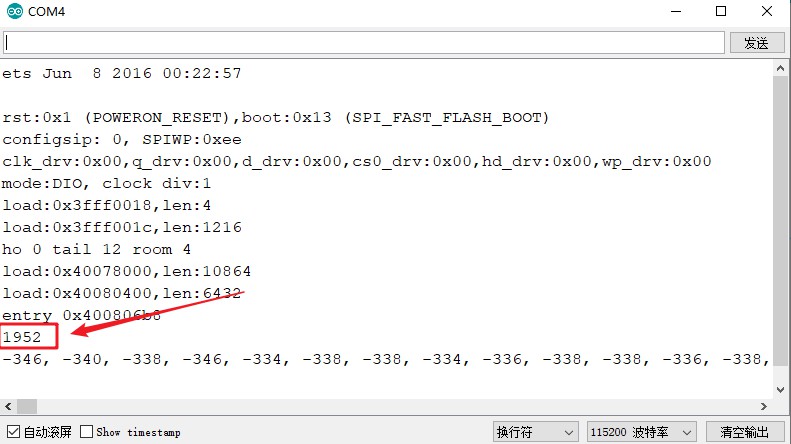
}

//保留之前代码

}

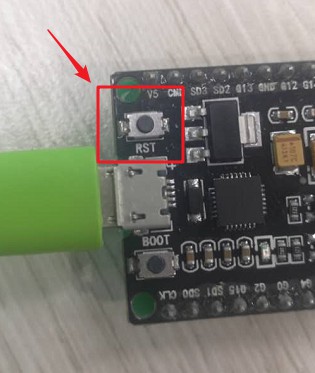
}

  最终我们获得到的总耗时数为1952毫秒,这个结果已经能够很好地说明问题了,也就是说我们采集16000个点花了1952毫秒的时间，在偏差允许的范围内约等于2秒,恰好是我们所期望得到的值。



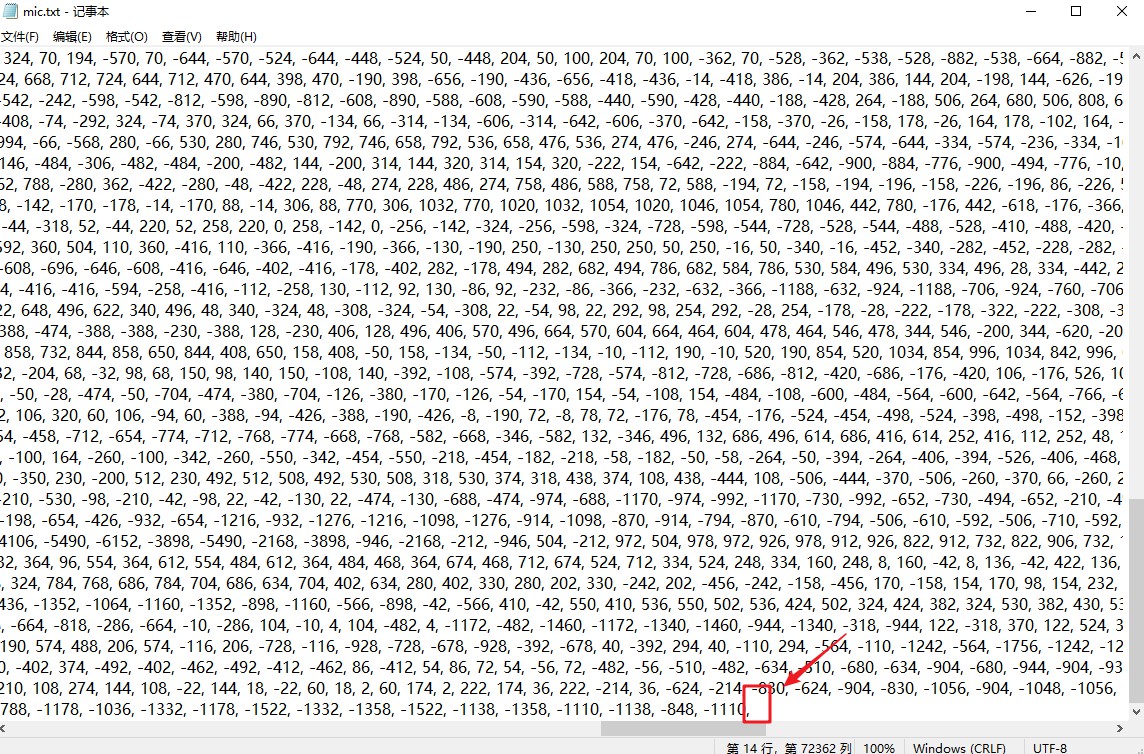
  如果我们将采样频率从8000设置为44100，我们就会观察到这个数据变成了353。采样频率提高之后，采集相同样本所需要的时间就短了。

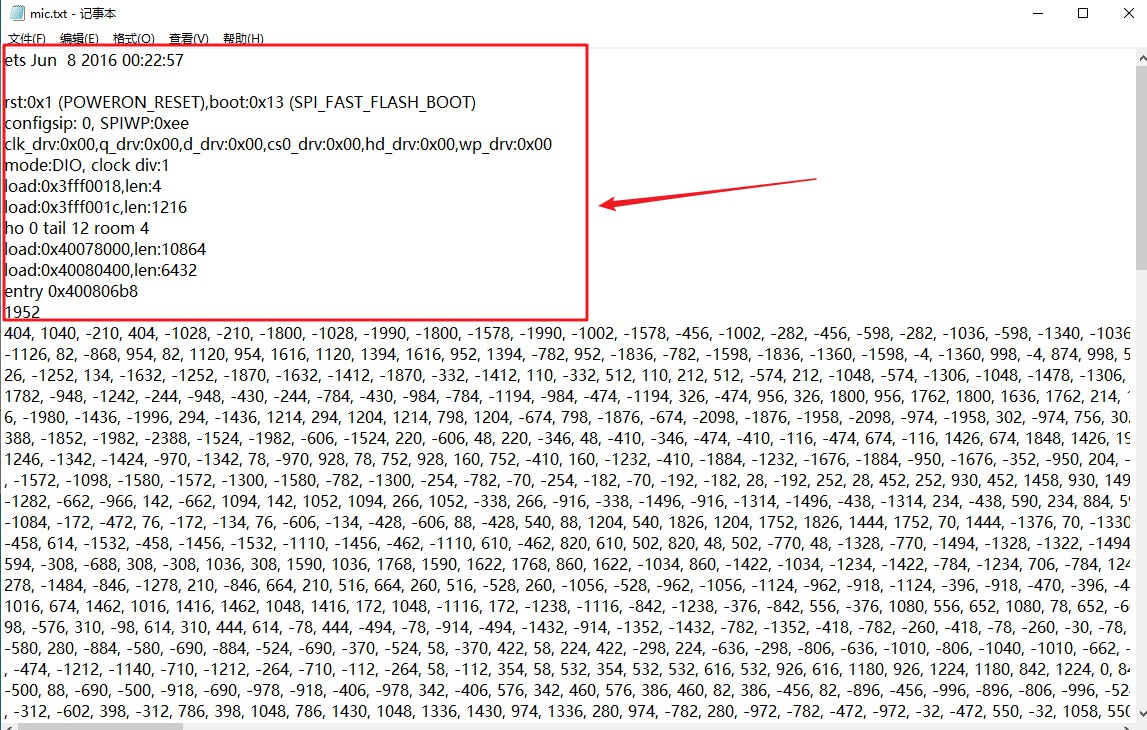
  接下去我们继续进行声音的还原，除了使得开发板断电后上电进行重启，我们也可以按板载的RST按钮进行重启。



  重启后立刻对麦克风说点什么，让声音被记录下来。等到串口完成数据的输出之后，我们新建一个mic.txt文件，将串口输出的内容Ctrl+A,Ctrl+C复制黏贴到其中。

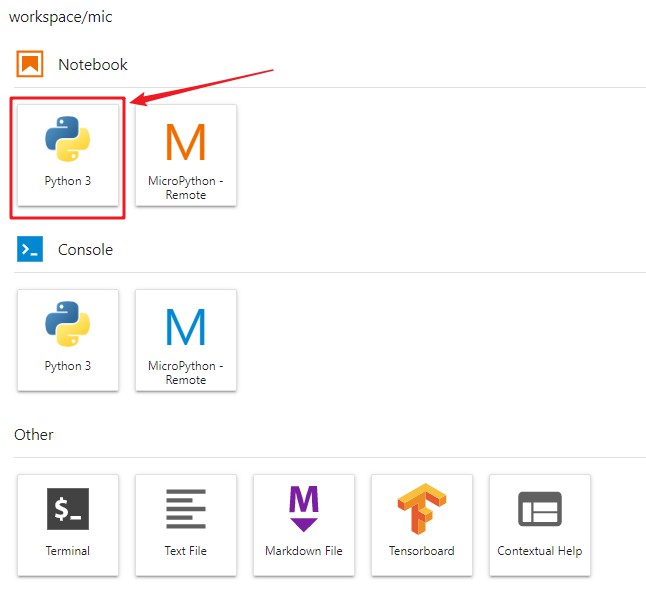
  我们去除最后一个,然后再去除头部的无关信息最后获得一串CSV格式的数据。





  我们保存mic.txt文件，并将其重命名为mic.csv文件，然后上传到云平台上进行处理。

  我们新建一个Python3文件作为编程环境。



  我们首先引入一些python常用的库。

import wave

import numpy as np

import array

import struct

import pandas as pd

  接着我们需要将mic.csv文件中的数据加载到python3运行的内存中去，我们使用pandas进行数据的读取。

mic = pd.read\_csv('mic.csv', header = None)

data = mic.values

  可以使用print(data)的方法打印data数据,我们发现data目前是一个2维数组，那么我们为了避免后续步骤出错，需要将其转化为一维数组。

mic = pd.read\_csv('mic.csv', header = None)

data = mic.values

data=data[0]

  接下去我们需要提取出每个采样点的数据和生成一个采样频率。

rate=8000

pcmData16Bit=data[0:16000]

print(pcmData16Bit)

  然后为了最终合成为WAV音频格式文件，我们需要将其转换为浮点类型的数据。

pcmData16Bit\_f=pcmData16Bit.astype(np.float32)

  一切数据准备都做好了之后我们就准备开始合成这段音频。我们首先定义要生成的文件的名称。当然他最终会以相对路径的方式进行解析，最终将音频导出到与本python3文件相同文件夹下。

name="mic.wav"

  接着我们定义一个waveSetOut函数，这个函数主要用于合成音频。我们全程都会使用wave的库进行操作。首先在内存中创建一个waveF文件，然后设置其声道数为1、设置其量化宽度(量化宽度)为2（16bit）、设置采样率为真实的采样率、这个采样率最终会配合采样的点数来决定这段音频的时间长度。

  然后我们需要将PCM的每一个采样点通过struct库进行序列化操作，转化为一个个的字符序列，方便存储和传输。

  最后将音频数据写入磁盘后关闭。

def waveSetOut(outPath,pcmArray,pcmRate):

waveF=wave.open(outPath,"wb")

waveF.setnchannels(1)

waveF.setsampwidth(2)

waveF.setframerate(pcmRate)

for val in pcmArray:

val=round(float(val))

dataStruct =struct.pack("<h",val)

waveF.writeframesraw(dataStruct)

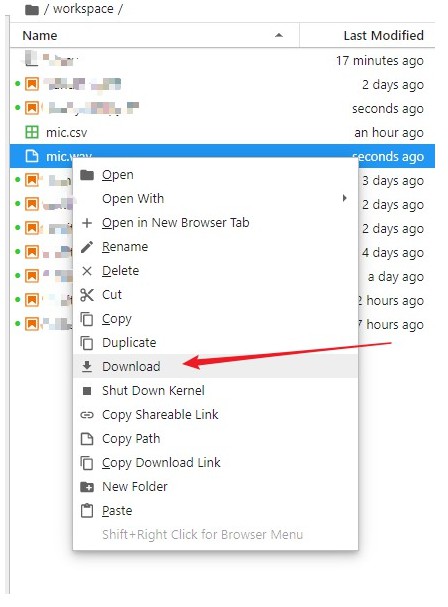
waveF.writeframes(b"")

waveF.close()

  我们可以实际运行一下这段代码，观察是否产生了我们希望得到的那个音频文件。

waveSetOut(name,pcmData16Bit\_f,rate)

  我们可以将音频下载到本地进行播放。



## 录更长时间的音频

  我们需要录制超过2秒钟的音频会比较困难，因为首先是由于ESP32的内存有限，我们无法开辟很大的缓冲区去暂存音频PCM数据。

  其次，我们可能会想到使用一边采用串口输出的方式及时释放内存空间，一边继续采集数据持续对音频进行采样。

  那么对于后面这种方式其实也不是非常的理想，因为我们可以实践一下，会发现串口输出释放内存的速度不如采样的速度来得快。

  那么对此有什么办法呢？以及我们为什么要录更长时间的音频呢？

  我们最后在识别的时候，肯定是用每隔两秒这样不联系的声音样本去识别，必须是一个流畅完整持续的声音样本，从中进行识别。

  解决的方案是，首先我们不用串口，我们采用websocket的方式将数据传到浏览器端进行PCM信号的收集，这种方法会比串口输出来得快，以确保缓冲区内数据不会溢出。

  其次我们不使用i2s\_pop\_sample的方法，我们使用i2s\_read\_bytes的方法来读取数据，这种方法直接排除其他干扰将数据填入samples数组中。

  以下是所有的代码，由于我们需要将PCM数据发送给浏览器，所以我们需要使得开发板连入路由器，使之和浏览器处在同一个局域网内。最简单的做法就是让板子和运行浏览器的电脑接入同一个路由器。所以在以下代码中我们需要配置ssid和password，让板子连接路由器。当然这里值得注意的是开发板只能接入2.4G的路由器，由于硬件的原因不能接入5G路由器。

#include <driver/i2s.h>

#include <WiFi.h>

#include <WebServer.h>

#include <WebSocketsServer.h>

#include <ArduinoJson.h>

#define I2S\_PORT I2S\_NUM\_1

#define I2S\_WS 18

#define I2S\_SD 32

#define I2S\_SCK 14

const int BLOCK\_SIZE = 512;

int16\_t samples[BLOCK\_SIZE];

const char \* ssid = "\*\*\*\*\*\*";

const char \* password = "\*\*\*\*\*\*";

char webpage[] PROGMEM = R"=====(

<html>

<head>

<script src='https://cdn.staticfile.org/jquery/1.10.2/jquery.min.js'></script>

</head>

<body onload="javascript:init()">

<p></p>

<script>

var webSocket

var data

var count=0

var voice=new Array()

var show=""

function init() {

webSocket = new WebSocket('ws://' + window.location.hostname + ':81/');

webSocket.onmessage = function(event) {

data = JSON.parse(event.data);

voice=voice.concat(data.data)

show=JSON.stringify(voice)

$("p").html(show);

}

}

</script>

</body>

</html>

)=====";

WebServer server(80);

WebSocketsServer webSocket = WebSocketsServer(81);

void setup() {

Serial.begin(115200);

i2s\_init();

i2s\_setpin();

WiFi.begin(ssid, password);

while(WiFi.status()!=WL\_CONNECTED) {

Serial.print(".");

delay(500);

}

Serial.println("");

Serial.print("IP Address: ");

Serial.println(WiFi.localIP());

delay(1000);

server.on("/",[](){

server.send\_P(200, "text/html", webpage);

});

server.begin();

webSocket.begin();

}

void loop() {

webSocket.loop();

server.handleClient();

int num\_bytes\_read = i2s\_read\_bytes(I2S\_PORT,

(void \*)samples,

BLOCK\_SIZE\*2,

portMAX\_DELAY); // no timeout

String json="";

DynamicJsonBuffer jsonBuffer;

JsonObject& jsonobj = jsonBuffer.createObject();

JsonArray& data = jsonobj.createNestedArray("data");

for (int i = 0; i < BLOCK\_SIZE; i++) {

data.add(samples[i]);

}

jsonobj.printTo(json);

webSocket.broadcastTXT(json.c\_str(), json.length());

}

void i2s\_init(){

const i2s\_config\_t i2s\_config = {

.mode = i2s\_mode\_t(I2S\_MODE\_MASTER | I2S\_MODE\_RX),

.sample\_rate = 8000,

.bits\_per\_sample = i2s\_bits\_per\_sample\_t(16),

.channel\_format = I2S\_CHANNEL\_FMT\_ONLY\_LEFT,

.communication\_format = i2s\_comm\_format\_t(I2S\_COMM\_FORMAT\_I2S | I2S\_COMM\_FORMAT\_I2S\_MSB),

.intr\_alloc\_flags = 0, // default interrupt priority

.dma\_buf\_count = 8,

.dma\_buf\_len = 64,

.use\_apll = true

};

i2s\_driver\_install(I2S\_PORT, &i2s\_config, 0, NULL);

}

void i2s\_setpin(){

const i2s\_pin\_config\_t pin\_config = {

.bck\_io\_num = I2S\_SCK,

.ws\_io\_num = I2S\_WS,

.data\_out\_num = -1,

.data\_in\_num = I2S\_SD

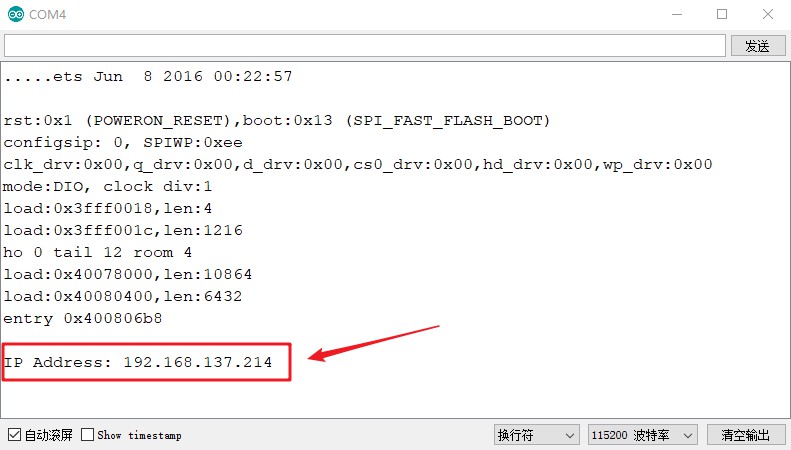
};

i2s\_set\_pin(I2S\_PORT, &pin\_config);

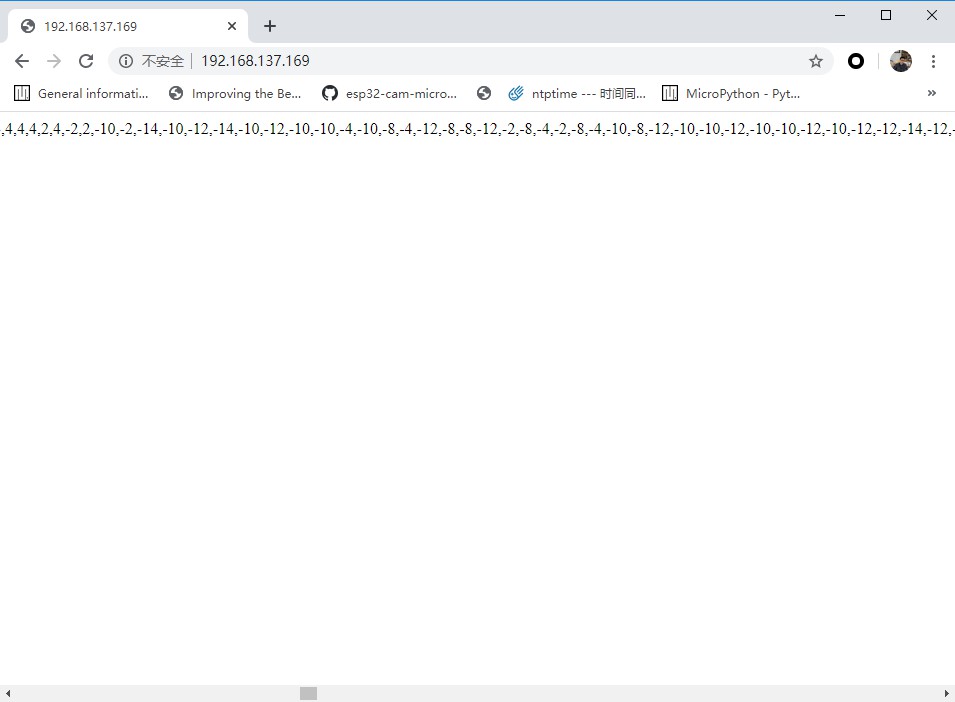
}

  关于这段代码最主要的是i2s\_read\_bytes,在这个函数中，我们传入一个int16\_t的数组指针和它长度的两倍。因为i2s\_read\_bytes是一个个bytes进行读取的，而一个int16\_t所占的是两个bytes,所以在长度上要乘2。

  这段代码的运行效果是这样的，连接成功后在串口监视器中会返回一个IP地址，我们可以之间通过浏览器访问这个IP。当然我们每个人在实际网络环境下的IP都不一样。



  最后打开浏览器，我们会发现那串我们要的PCM码,当然它会一直采集下去直到我们将开发板断电为止。这时我们就可以使用与刚才一样的方法将PCM码丢到云上转成音频。此时的音频应该是一段连续不断的音频。



# FFT频谱数据可视化

## PCM可视化

  我们继续对音频进行分析，最直观地使用图像来观察目前的PCM音频是怎么样的。

  我们可以继续在mic.wav文件所在目录下新建一个python3文件，在这个代码中我们使用以下代码就可以看到整个PCM信号在时间上的分布。

import wave

import numpy as np

import pylab as pl

waveAbsPath="mic.wav"

f=wave.open(waveAbsPath,"rb")

params=f.getparams()

nchannels, sampwidth, framerate, nframes = params[:4]

str\_data = f.readframes(nframes)

wave\_data = np.frombuffer(str\_data, "int16")

wave\_data.shape = -1, 2

#将其转置得到：

wave\_data = wave\_data.T

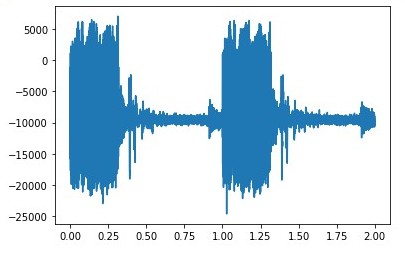
#通过取样点数和取样频率计算出每个取样的时间：

time = np.arange(0, nframes)/framerate

pl.plot(time, wave\_data.reshape((wave\_data.shape[0] \* wave\_data.shape[1])))

pl.show()

  最终我们可以观察到这样一幅图像，这副图横坐标表示时间，纵坐标表示此刻的声波能量大小。

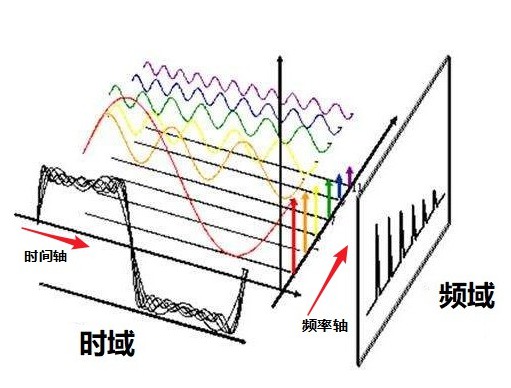


  的确，使用这幅图，我们已经能够完整地描述一整段音频信息，至少我们能够把它反过来还原出声音。

  但是这幅图并不是声音的全部信息。一般在目前比较流行的人工智能语音识别等项目中我们需要使用的往往是这副图的另一面。

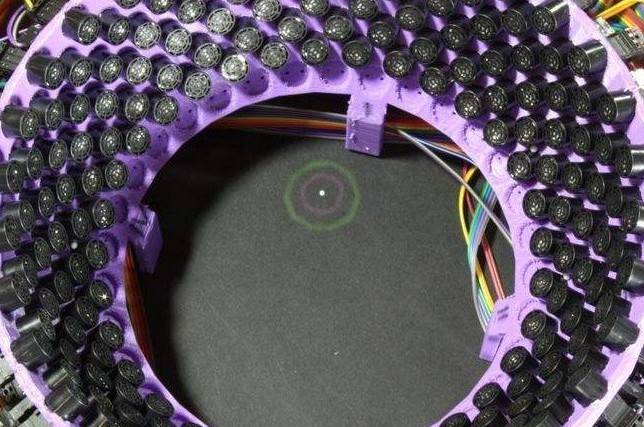
## 频谱图

  我们可以看下面这副图，这副图将声音分为时域和频域两个方面。其中声音在时域上的图像就是直接用PCM信号画出来的图。



  那么频域上的图像又是讨论怎么，样一件事情呢？假如我们在某个时刻，给正在播放的音乐切上这么一刀，假如当前播放音乐的蜂鸣器能够发出非常纯频率的声音，那么很显然，一个PCM采样点的数字是无法用于表示当前时刻的声音频率和声音响度的。

  况且，万一这段音乐还同时是由多个不同频率的蜂鸣器一起和弦演奏的，那就更不能从这一个采样点中获取到这所有的蜂鸣器的演奏情况,更何况只要条件允许，蜂鸣器可以没有上限地添加。



  那么其实对于每一个PCM的采样点而言，都只能采集到这一个时刻的瞬间所有声音频率和响度的总合情况。

  我们再回来解释频域上的这幅频谱图。频谱图的X轴是用来描述递增的不同的频率。我们可以简单将它理解成有这么多个发出不同频率声音信号的蜂鸣器。然后频谱图的纵坐标就是这些蜂鸣器在此刻发出声音的响度。

  那么我们要如何将PCM图转化为频谱图呢？我们是否可以通过某种方法来将这每一个PCM采样点背后的声音的各种频率与振幅（响度）情况分析出来呢? 其实这种方法就是傅里叶变换。如果在计算机内做声音的频谱分析，我们可以尝试快速傅里叶变换（FFT），因为快速傅里叶变换的时间复杂度仅仅是O(nlog2n)。

## 快速傅里叶变换

  我们需要在底层实现快速傅里叶变换的功能，因为我们最终的推理需要在底层完成。

  我们使用上一小节最后的录更长时间的音频代码基础上进行迭代。我们首先引入arduinoFFT库。

//保留之前代码

#include <ArduinoJson.h>

//添加↓

#include <arduinoFFT.h>

//添加↑

#define I2S\_PORT I2S\_NUM\_1

//保留之前代码

  然后我们添加一些后续可能会用到的初始化变量。

//保留之前代码

int16\_t samples[BLOCK\_SIZE];

//添加↓

arduinoFFT FFT = arduinoFFT();

double vReal[BLOCK\_SIZE];

double vImag[BLOCK\_SIZE];

String labels[] = {"125", "250", "500", "1K", "2K", "4K", "8K", "16K"};

int bands[8] = {0, 0, 0, 0, 0, 0, 0, 0};

const uint8\_t amplitude = 150;

//添加↑

const char \* ssid = "tinyml";

//保留之前代码

  接着我们需要编写一个FFT\_Operation函数来处理PCM信号。我们构建一个空的函数然后在loop函数中完成采样后调用FFT\_Operation函数进行快速傅里叶变换。

void loop() {

webSocket.loop();

server.handleClient();

int num\_bytes\_read = i2s\_read\_bytes(I2S\_PORT,

(void \*)samples,

BLOCK\_SIZE\*2,

portMAX\_DELAY);

//删除↓

// String json="";

// DynamicJsonBuffer jsonBuffer;

// JsonObject& jsonobj = jsonBuffer.createObject();

// JsonArray& data = jsonobj.createNestedArray("data");

// for (int i = 0; i < BLOCK\_SIZE; i++) {

// data.add(samples[i]);

// }

// jsonobj.printTo(json);

// webSocket.broadcastTXT(json.c\_str(), json.length());

//删除↑

//添加↓

FFT\_Operation();

//添加↑

}

//添加↓

void FFT\_Operation(){

}

//添加↑

  我们继续完善这个快速傅里叶变换函数，在这个函数中，我们只要关心函数的传出结果即可。在经过fft运算之后，会返回N个点的复数数组，vReal和vImag,它们分别表示一个数的实部和虚部。

  为了计算的快速进行，我们可以先抛弃掉虚部vImag,对实部进行操作分析和处理。因为实部已经可以用于反映不同频率的振幅情况。

void FFT\_Operation(){

for (uint16\_t i = 0; i < BLOCK\_SIZE; i++) {

vReal[i] = samples[i] << 8;

vImag[i] = 0.0;

}

FFT.Windowing(vReal, BLOCK\_SIZE, FFT\_WIN\_TYP\_HAMMING, FFT\_FORWARD);

FFT.Compute(vReal, vImag, BLOCK\_SIZE, FFT\_FORWARD);

FFT.ComplexToMagnitude(vReal, vImag, BLOCK\_SIZE);

for (int i = 0; i < 8; i++) {

bands[i] = 0;

}

//0赫兹不存在，另外由于FFT结果的对称性，通常我们只使用前半部分的结果，即小于采样频率一半的结果。

for (int i = 1; i < (BLOCK\_SIZE/2); i++){

if (vReal[i] > 2000) {

if (i<2 ) bands[0] = max(bands[0], (int)(vReal[i]/amplitude)); // 125Hz

if (i >=2 && i<=4 ) bands[1] = max(bands[1], (int)(vReal[i]/amplitude)); // 250Hz

if (i >4 && i<=8 ) bands[2] = max(bands[2], (int)(vReal[i]/amplitude)); // 500Hz

if (i >8 && i<=15 ) bands[3] = max(bands[3], (int)(vReal[i]/amplitude)); // 1000Hz

if (i >15 && i<=35 ) bands[4] = max(bands[4], (int)(vReal[i]/amplitude)); // 2000Hz

if (i >35 && i<=58 ) bands[5] = max(bands[5], (int)(vReal[i]/amplitude)); // 4000Hz

if (i >58 && i<=128 ) bands[6] = max(bands[6], (int)(vReal[i]/amplitude)); // 8000Hz

if (i >128 ) bands[7] = max(bands[7], (int)(vReal[i]/amplitude)); // 16000Hz

}

}

}

  那么接下去还有个问题，我们如何从vReal中分离出不同频率点的振幅状况？

  这其实和我们的采样频率以及输入到傅里叶变换中的样本容量有关。如果仅仅是为了傅里叶变换的准确性，那么这两个值都是越大越好。

  在解释这个问题之前，我们先将原本8000的采样频率调整到44100。

void i2s\_init(){

const i2s\_config\_t i2s\_config = {

.mode = i2s\_mode\_t(I2S\_MODE\_MASTER | I2S\_MODE\_RX),

//修改↓

.sample\_rate = 44100,

//修改↑

.bits\_per\_sample = i2s\_bits\_per\_sample\_t(16),

.channel\_format = I2S\_CHANNEL\_FMT\_ONLY\_LEFT,

.communication\_format = i2s\_comm\_format\_t(I2S\_COMM\_FORMAT\_I2S | I2S\_COMM\_FORMAT\_I2S\_MSB),

.intr\_alloc\_flags = 0, // default interrupt priority

.dma\_buf\_count = 8,

.dma\_buf\_len = 64,

.use\_apll = true

};

i2s\_driver\_install(I2S\_PORT, &i2s\_config, 0, NULL);

}

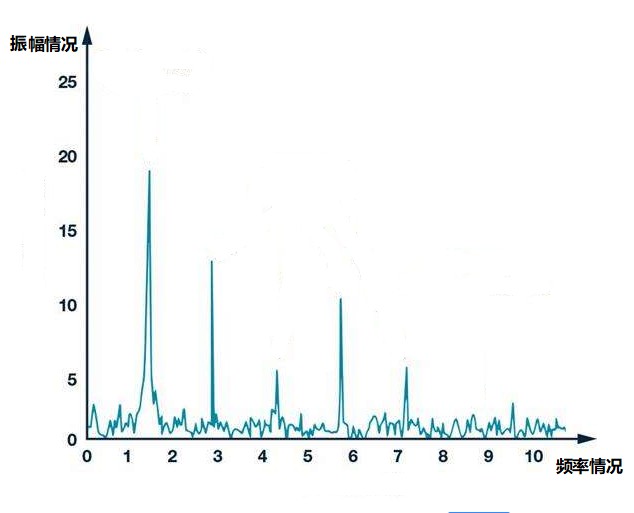
  那么到目前为止，我们的采样频率为44100Hz,样本容量（代码中的BLOCK\_SIZE）为512。这512个点经过了FFT之后会将我们侧重研究的结果保存在vReal中。

  在vReal数组中，它将声音的频率从0Hz到44100Hz进行切分，切分成vReal的数组长度等分，也就是512份,然后依次将这512个频率下的振幅情况的值保存在vReal数组中。

  举个例子来说明，在vReal下标为5的位置，vReal[5]所描述的振幅情况是(5+1)\*(44100/512)=516.80Hz频率下的振幅情况。 另外由于FFT结果的对称性，通常我们只使用前半部分的结果，即小于采样频率一半的结果，也就是在本案例中我们最多只能测到22050Hz。

  最后还有几个现象：采样频率越高，我们可以测量的频谱范围越大，这也是我们将采样频率从8000Hz提高到44100Hz的原因。

  然后样本容量越大，可以切分的频率精度就越细，当然样本容量和采样时间是有关系的，我们尽可能将CPU的时间平均分配开来做各种事情，因为后面还有进行人工智能推理运算，所以选取512个点作为样本容量是一个比较合理的做法。



## 显示频谱图

  最后我们需要将结果进行可视化处理，在网页上显示实时的频谱图。我们目前得到的数据是8个特征频率的振幅情况，这些数据存放在bands数组中。

  我们先使用Send\_Msg函数对数据进行前置处理,将处理完的数据使用webSocket发送出去。

//保留之前代码

void loop() {

webSocket.loop();

server.handleClient();

int num\_bytes\_read = i2s\_read\_bytes(I2S\_PORT,

(void \*)samples,

BLOCK\_SIZE\*2,

portMAX\_DELAY);

FFT\_Operation();

//添加↓

Send\_Msg();

//添加↑

}

//保留之前代码

void FFT\_Operation(){

//保留之前代码

}

//添加↓

void Send\_Msg() {

String json = "[";

for (int i = 0; i < 8; i++) {

if (i > 0) {

json +=", ";

}

json += "{\"bin\":";

json += "\"" + labels[i] + "\"";

json += ", \"value\":";

json += String(bands[i]);

json += "}";

}

json += "]";

webSocket.broadcastTXT(json.c\_str(), json.length());

}

//添加↑

//保留之前代码

  然后我们修改webpage,使得浏览器能够接到开发板的实时频谱数据并渲染出当前的频谱图。

//删除↓

//char webpage[] PROGMEM = R"=====(

//<html>

//<head>

// <script src='https://cdn.staticfile.org/jquery/1.10.2/jquery.min.js'></script>

//</head>

//<body onload="javascript:init()">

//

//<p></p>

//

//

//<script>

// var webSocket

// var data

// var count=0

// var voice=new Array()

// var show=""

// function init() {

// webSocket = new WebSocket('ws://' + window.location.hostname + ':81/');

// webSocket.onmessage = function(event) {

// data = JSON.parse(event.data);

// voice=voice.concat(data.data)

// show=JSON.stringify(voice)

// $("p").html(show);

// }

// }

//

//</script>

//</body>

//</html>

//)=====";

//删除↑

//添加↓

char webpage[] PROGMEM = R"=====(

<html>

<!-- Adding a data chart using Chart.js -->

<head>

<script src='https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.5.0/Chart.min.js'></script>

</head>

<body onload="javascript:init()">

<div>

<canvas id="chart" width="600" height="400"></canvas>

</div>

<!-- Adding a websocket to the client (webpage) -->

<script>

var webSocket, dataPlot;

var maxDataPoints = 20;

const maxValue = 500000;

const maxLow = maxValue \* 0.5;

const maxMedium = maxValue \* 0.2;

const maxHigh = maxValue \* 0.3;

function init() {

webSocket = new WebSocket('ws://' + window.location.hostname + ':81/');

dataPlot = new Chart(document.getElementById("chart"), {

type: 'bar',

data: {

labels: [],

datasets: [{

data: [],

label: "L",

backgroundColor: "#D6E9C6"

},

{

data: [],

label: "M",

backgroundColor: "#FAEBCC"

},

{

data: [],

label: "H",

backgroundColor: "#EBCCD1"

},

]

},

options: {

responsive: false,

animation: false,

scales: {

xAxes: [{ stacked: true }],

yAxes: [{

display: true,

stacked: true,

ticks: {

beginAtZero: true,

steps: 1000,

stepValue: 500,

max: maxValue

}

}]

}

}

});

webSocket.onmessage = function(event) {

var data = JSON.parse(event.data);

dataPlot.data.labels = [];

dataPlot.data.datasets[0].data = [];

dataPlot.data.datasets[1].data = [];

dataPlot.data.datasets[2].data = [];

data.forEach(function(element) {

dataPlot.data.labels.push(element.bin);

var lowValue = Math.min(maxLow, element.value);

dataPlot.data.datasets[0].data.push(lowValue);

var mediumValue = Math.min(Math.max(0, element.value - lowValue), maxMedium);

dataPlot.data.datasets[1].data.push(mediumValue);

var highValue = Math.max(0, element.value - lowValue - mediumValue);

dataPlot.data.datasets[2].data.push(highValue);

});

dataPlot.update();

}

}

</script>

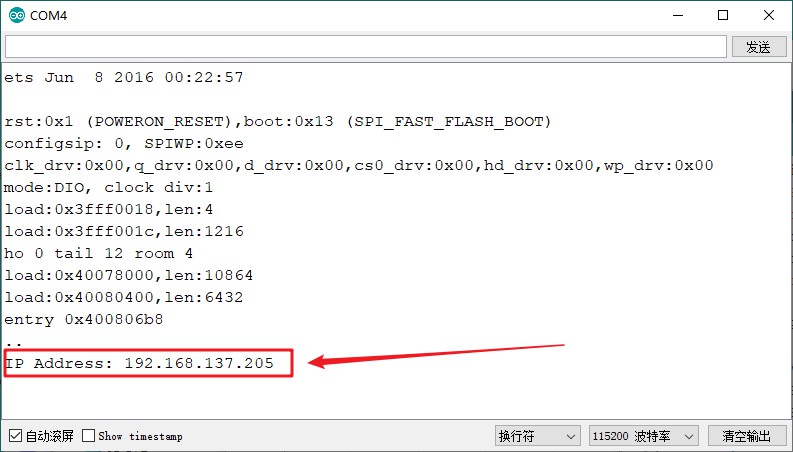
</body>

</html>

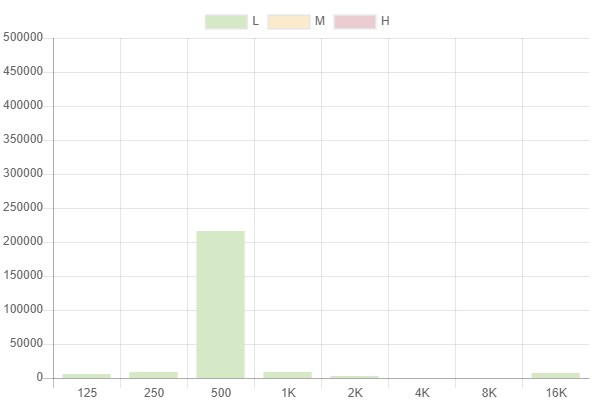
)=====";

//添加↑

  最后我们与之前一样，使用串口获取到IP地址，并通过这个IP地址使用浏览器进行访问。



  进去以后我们可以访问到这张频谱图。我们可以尝试在网上找一些频率比较纯的声音进行分析，比如我们对它播放一段500Hz的声音。就会出现以下的效果。



  其实到这里我们已经可以知道对于声音的识别是怎么进行的了，因为目前至少我们已经有能力将纯的频率的声音区分出来了。

  我们对麦克风说的话，无非就是不同频率的纯频声音的叠加而已。

# 数据集收集

## 量化振幅范围

  我们之前没有对振幅进行统一的量化，振幅情况随着声音的响度的上升可能会一直上升最终超出频谱表的最大范围。那么我们就需要确定一个有效范围去描述振幅，并且最终将振幅情况映射到0-128的区间内，用以表达。

  我们首先修改webpage网页变量,将频谱表的纵坐标范围定义在0-128之间。

//保留之前代码

char webpage[] PROGMEM = R"=====(

//保留之前代码

var maxDataPoints = 20;

//修改值为128↓

const maxValue = 128;

//修改值为128↑

const maxLow = maxValue \* 0.5;

//保留之前代码

)=====";

//保留之前代码

  然后我们添加一个constrain128函数将bands进行量化。

//保留之前代码

void loop() {

//保留之前代码

FFT\_Operation();

//添加↓

constrain128();

//添加↑

Send\_Msg();

}

//添加↓

void constrain128(){

for(int j=0;j<8;j++){

bands[j]=int(bands[j]\*(128.0/100000));

bands[j]=constrain(bands[j],0,128);

}

}

//添加↑

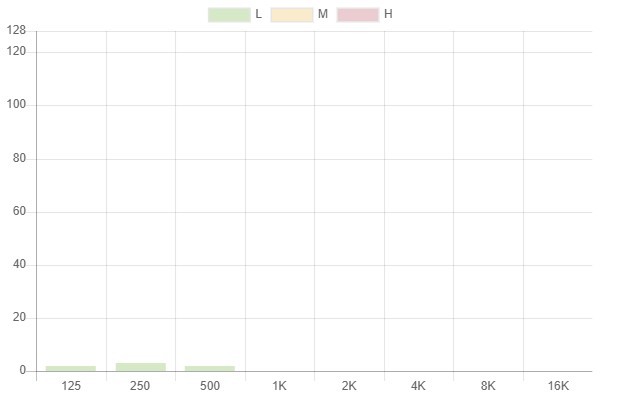
void Send\_Msg() {

//保留之前代码

}

//保留之前代码

  在这个函数中，那个100000的数值可以通过多次调整和观察得到。如果我们正常说话，发现频谱表数据变化如下图一样不明显，那么就将这个值调低，反之则调高。这个调节的是响度范围，我们暂时用这种方式手工调整一个合适的响度范围，并不讨论响度和识别的关系。（当然我们可以自行研究）。由于每个人实验的环境和发出声音的响度不同，这个值可能会略有不相同。



## 数据集收集

  收集数据集的过程是一个繁琐的体力劳动，而设计一套收集数据集的流程却又十分复杂。尤其是这些边缘人工智能设备，它们对于人的交互没有像鼠标键盘这么成熟，甚至连什么时候开始记录，什么时候结束记录都呈很大的问题。

  那么对于本项目的数据集收集，我们需要越过以下几个瓶颈问题:

  1.选取合理的参数用于对声音进行描述。

  2.相对准确地捕捉每个声音从开始到结束的数据。

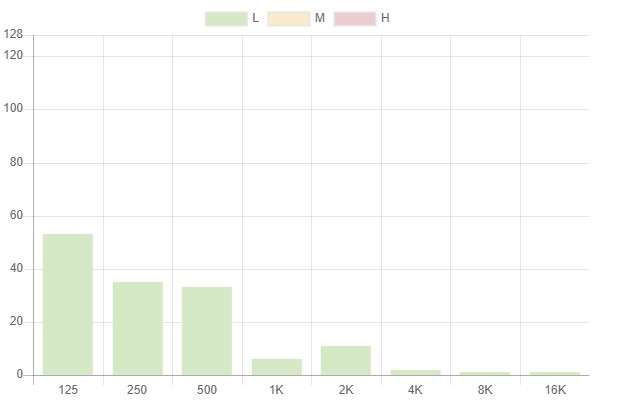
  3.每个声音可以由多少条数据来描述（样本容量是多少）。

  4.如何采集样本容量合适且相等的声音样本数据。

  5.数据集如何有效地记录，在这份记录中每个声音的采样次数是多少相对合适。

### 问题1的方案

  针对这一点，我们会采用的是声音在频域上的特征数据，使用频域数据的好处是能够直接展现出声音在不同频率下的情况，是可以作为区分不同声音的一个指标的。另一个好处是频域上的特征无法还原出原始的音频，它会严重失真，这样也可以保护个人隐私。所以我们采集的数据集是声音在频域上的特征。我们也不会选取所有频谱点，我们会平均选取其中的几个特征点进行分析。



### 问题2的方案

  如果是采集图像数据，那很简单，按一下快门就完事了。但是我们要如何去捕捉一段声音从开始到结束的数据呢？

  我们继续在之前的代码基础上进行迭代，为程序添加Check\_Start函数。

  在这个Check\_Start函数中我们设置两个阈值，一个阈值是声音响度的上限(Threshold\_HIGH)，另一个阈值是声音响度的下限(Threshold\_LOW)。我们可以实际测量和调整出我们说话和没说话时的声音响度，然后记录下来形成阈值。

  有了这些响度阈值之后，我们就可以在Check\_Start函数中写三个判断，判断开始统计的时刻，判断声音是否应该被记录，判断话音刚落的时刻。

//保留之前代码

void loop(){

//保留之前代码

constrain128();

//添加↓

Check\_Start();

//添加↑

Send\_Msg();

}

//添加↓

int bsum=0;

int Threshold\_HIGH=40;

int Threshold\_LOW=20;

int smooth\_count=0;

int record\_count=0;

void Check\_Start(){

bsum=0;

for(int j=0;j<8;j++){

bsum=bsum+bands[j];

}

if(bsum>=Threshold\_HIGH&&record\_count==0){ //开始统计

Serial.println("开始");

smooth\_count=0;

}

if(smooth\_count<=10){ //声音统计中

if(bsum>Threshold\_LOW) //清除疑似没人说话的记录

{

smooth\_count=0;

}

record\_count++;

}

if(bsum <= Threshold\_LOW)//检测到疑似没人说话

{

if(smooth\_count>=10&&record\_count>0) //话音刚落结束统计

{

Serial.println("检测完成");

Serial.print("收集到的元组数为：");

Serial.println(record\_count-10); //最后10条数据没有实际效果

record\_count=0;

}

smooth\_count++;

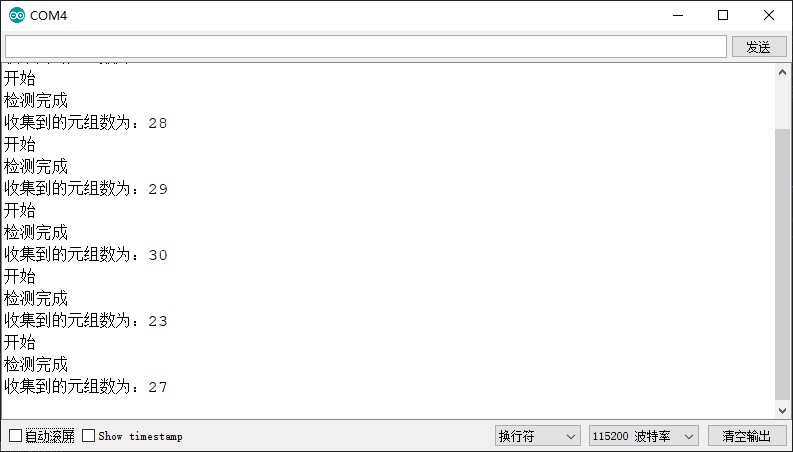
}

}

//添加↑

### 问题3的方案

  然后我们找一个安静的地方，开始检测说话产生的数据条数。我们以念开和关这两个语音为例。可以发现开和关这样一个字大约会产生20-30条左右的数据。可以多尝试几次，我们最终就可以将需要采集多少条数据确定下来。在本案例中，根据实际测量结果，不妨使用30条频谱数据来描述一个语音指令。



### 问题4的方案

  根据上面的步骤，我们已经定义好了用来描述每个声音指令的数据条数，那么我们就按照这个标准来收集数据集，每个声音都需要收集30个频谱数据。

  于是我们继续修改代码，首先我们定义record\_num来存放要每个声音要收集的数据条数。我们会先将数据存放到record\_125至record\_16K这8个数组中，等完成收集后再将数据通过串口打印出来。因为串口打印是需要花费大量的时间的，这会降低采样的分辨率，所以先将采集到的数据存放到一个数组中，然后再将这个数组打印出来会是一个比较好的方法。

//保留之前代码

const uint8\_t amplitude = 150;

//添加↓

const int record\_num=30;

float record\_125[record\_num];

float record\_250[record\_num];

float record\_500[record\_num];

float record\_1k[record\_num];

float record\_2k[record\_num];

float record\_4k[record\_num];

float record\_8k[record\_num];

float record\_16k[record\_num];

//添加↑

const char \* ssid = "tinyml";

//保留之前代码

  之后我们对Check\_Start函数中的三个判断进行修改，这三个判断完成的逻辑是检测到声音发出后将数据通过串口打印出来。

//修改↓

int record\_count=-1;

//修改↑

void Check\_Start(){

bsum=0;

for(int j=0;j<8;j++){

bsum=bsum+bands[j];

}

//删去↓

// if(bsum>=Threshold\_HIGH&&record\_count==0){ //开始统计

// Serial.println("开始");

// smooth\_count=0;

// }

//

// if(smooth\_count<=10){ //声音统计中

// if(bsum>Threshold\_LOW) //清除疑似没人说话的记录

// {

// smooth\_count=0;

// }

// record\_count++;

// }

//

// if(bsum <= Threshold\_LOW)//检测到疑似没人说话

// {

// if(smooth\_count>=10&&record\_count>0) //话音刚落结束统计

// {

// Serial.println("检测完成");

// Serial.print("收集到的元组数为：");

// Serial.println(record\_count-10); //最后10条数据没有实际效果

// record\_count=0;

// }

// smooth\_count++;

// }

//删去↑

//添加↓

if(bsum>=Threshold\_HIGH&&record\_count==-1){

Serial.println("开始");

record\_count=0;

}

if(record\_count<record\_num&&record\_count!=-1)//收集30条数据

{

record\_125[record\_count]= bands[0];

record\_250[record\_count]= bands[1];

record\_500[record\_count]= bands[2];

record\_1k[record\_count]= bands[3];

record\_2k[record\_count]= bands[4];

record\_4k[record\_count]= bands[5];

record\_8k[record\_count]= bands[6];

record\_16k[record\_count]= bands[7];

record\_count++;

}

if(record\_count==record\_num)//收集完成一次声音的30个元组数据

{

for(int k=0;k<record\_count;k++){

Serial.print(record\_125[k]);

Serial.print(",");

Serial.print(record\_250[k]);

Serial.print(",");

Serial.print(record\_500[k]);

Serial.print(",");

Serial.print(record\_1k[k]);

Serial.print(",");

Serial.print(record\_2k[k]);

Serial.print(",");

Serial.print(record\_4k[k]);

Serial.print(",");

Serial.print(record\_8k[k]);

Serial.print(",");

Serial.print(record\_16k[k]);

Serial.println("");

}

Serial.println("检测完成");

Serial.print("收集到的元组数为：");

Serial.println(record\_count);

record\_count=-1;

}

//添加↑

}

### 问题5的方案

  我们最终需要拿着几份csv格式的数据集丢到云上去训练，当然如何训练是下一步的任务。问题5要解决的是怎么样去获得这份csv文件。

  到目前为止，arduino的数据输出方式有两种，一种是串口输出，另一种是websocket网络数据传输。我们能做的最快的获取数据集的方式就是直接从串口监视器里将数据复制出来。

  为了将数据一次性全都复制出来，我们需要修改代码，将一些其他的调试用数据输出关闭。

void Check\_Start(){

//保留之前代码

if(bsum>=Threshold\_HIGH&&record\_count==-1){

Serial.println("开始");

//保留之前代码

}

//保留之前代码

if(record\_count==record\_num)//收集完成一次声音的30个元组数据

{

for(int k=0;k<record\_count;k++){

//保留之前代码

}

//删除↓

// Serial.println("检测完成");

// Serial.print("收集到的元组数为：");

// Serial.println(record\_count);

//删除↑

record\_count=-1;

}

}

  到目前为止，虽然我们已经可以开始收集数据集了，但是我们需要面对开发板重复收集很多次同一个声音的数据，比方说开字的100次的数据。那么这里就会有个问题，假如在认真喊话的时候数错已经录了几次，就又得重新开始。

  那么我们就可以继续改写程序逻辑。当到达规定次数时，红灯亮起，若超过规定次数蓝灯亮起，这样就可以提醒我们数据集何时收集完毕。

  代码可以这样编写，我们需要引入用来控制RGBLED灯的库，并生成一个RGBLED灯的对象。

  我们在这个对象中定义12号引脚为控制灯的引脚，因为在实际电路中LED灯连接着ESP32的12号引脚，然后我们定义好灯的个数为1，因为这个灯采用一种I2S通信协议，能支持多个灯使用I2S总线进行串接并同时控制。

//保留之前代码

#include <arduinoFFT.h>

//添加↓

#include <Adafruit\_NeoPixel.h>

#define PIN 12

#define NUM 1 //灯个数

Adafruit\_NeoPixel pixels = Adafruit\_NeoPixel(NUM, PIN, NEO\_GRB + NEO\_KHZ800);

//添加↑

#define I2S\_PORT I2S\_NUM\_1

//保留之前代码

  接着我们定义一个data\_count变量来记录当前的声音次数，然后在声音触发时判断data\_count的值是否等于或者大于我们要采集的次数，从而改变灯的颜色。

  在声音次数方面我们不妨先以10次为例来展示如何收集数据集，并最终生成csv文件。我们会在Check\_Start函数中加入灯光的控制代码，以起到计数的作用，当然，我们也可以写一个简单的可视化进度条来打发数据集收集时的无聊。

//保留之前代码

float record\_16k[record\_num];

//添加↓

int data\_count=0;

int n=10; //声音的次数

//添加↑

const char \* ssid = "tinyml";

//保留之前代码

void setup() {

//保留之前代码

Serial.println(WiFi.localIP());

//添加↓

pixels.begin();

pixels.clear(); //清除颜色

pixels.show(); //打印

pixels.setPixelColor(0, pixels.Color(0,0,0)); //调颜色

pixels.show(); //打印

//添加↑

delay(1000);

//保留之前代码

}

void Check\_Start(){

//保留之前代码

if(bsum>=Threshold\_HIGH&&record\_count==-1){

//添加↓

data\_count++;

if(data\_count==n){

pixels.setPixelColor(0, pixels.Color(255,0,0)); //调颜色

pixels.show(); //打印

}else if(data\_count>n){

pixels.setPixelColor(0, pixels.Color(0,0,255)); //调颜色

pixels.show(); //打印

}

//添加↑

record\_count=0;

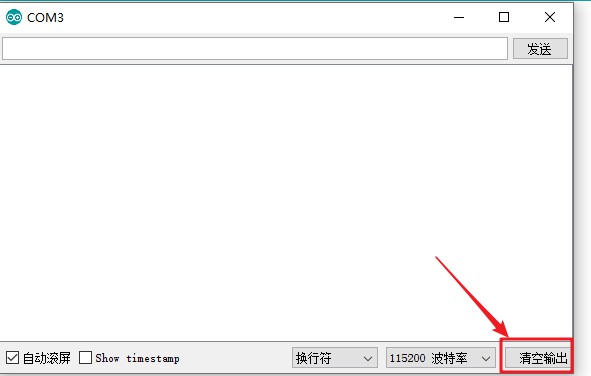
}

}

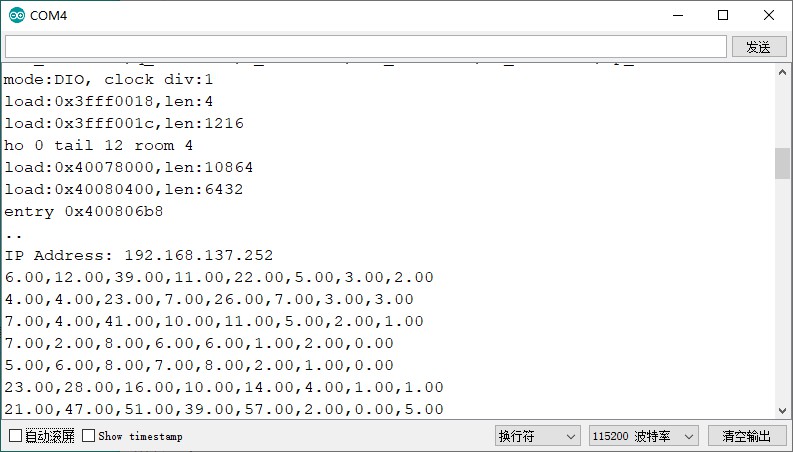
  将程序上传完成后，我们打开串口监视器，然后按下开发板上的RST按钮，重启开发板，以重置内部的变量。接着将板子轻拿轻放，并千万注意不要误触发数据的收集。



  如果一不小心误触发了，那么我们需要先清空串口监视器的数据输出,再按RST按钮。



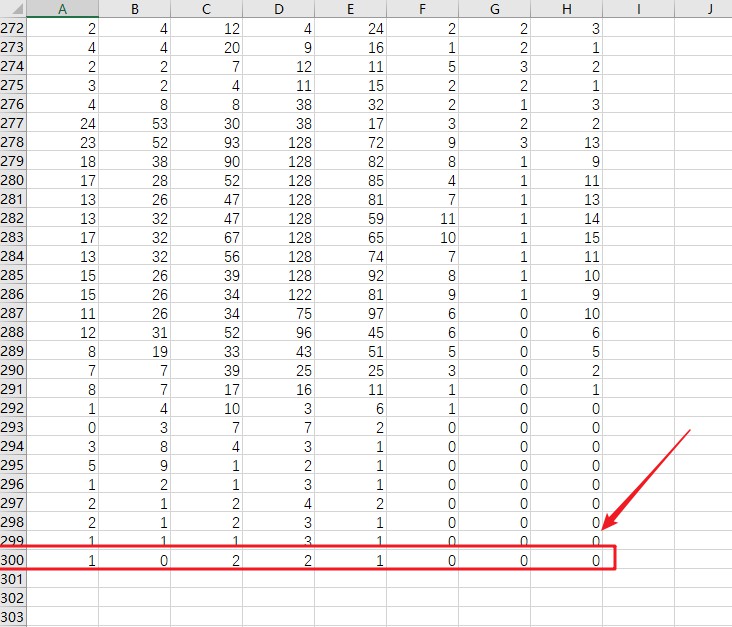
  一切准备就绪之后，我们就可以开始进行数据集的收集了，并且注意，我们此时只能先喊10次开作为开字的数据集。



  喊完100次的时候，红灯亮起，然后我们将串口监视器中的数据使用"Ctrl+A"和"Ctrl+C"的方法复制出来。并且在其他地方新建一个open.txt文件，将数据黏贴到其中。然后到文件内容的头部，将一些开机时的乱码信息直接删去，不留空行。



  保存完毕以后，我们将open.txt的文件名改为open.csv，然后使用Excel打开。理论上我们收集了10次声音数据，每个声音包含30个元组数据，所以在这张表格中应该会存在300条记录，并且每条记录都是由8个频率数据组成。



  这仅仅是收集了10次开的数据集。我们还需要用同样方法收集其他声音的数据集。

### 收集环境噪音

  除了收集到开和关数据以外，我们还需要收集环境噪音。因为我们最终要分类的其实是环境噪音、开的声音和关的声音这三种声音。在绝大多数时候，模型识别出来的是环境噪音。收集环境噪音十分简单，我们只要将高阈值调到0就可以了，等到红灯亮起，我们就能将这些数据用上面同样的方式做成.csv格式的数据集。

//保留之前代码

int bsum=0;

//添加↓

int Threshold\_HIGH=0;

//添加↑

int Threshold\_LOW=20;

//保留之前代码

  那么到目前为止，我们每个声音（开、关、噪音）都收集了10次，但是为了更加准确地识别这些声音，仅仅10次数据是不够的。我们对于每种声音都需要收集至少100次的数据。形成最终的数据集。

# 模型训练

  我们现在已经收集了3种声音，每种各100次的数据集，当然我们也可以直接在Tools/data文件夹下找到这两份数据集来继续完成下一步的实验，不过此份数据集的声音样本并不适用于所有人的声音，仅仅起到示例作用，如果需要效果好，就需要自己收集真实的数据集。

## 数据集预处理

  我们打开云平台，然后新建一个名为Codelabs的文件夹。并在该文件夹下新建一个app.ipynb的Python3 文件，并且在这个文件中进行编程。

  我们首先新建一个代码块引入相关的库,主要是tensorflow、keras这些人工智能计算库以及numpy、pandas这些科学计算库。然后tqdm 是进度条库。

import tensorflow as tf

import tensorflow.keras as keras

import tensorflow.keras.layers as layers

import numpy as np

import pandas as pd

from tqdm import tqdm

  我们继续新建一个代码块，并且在Codelabs文件夹中新建名为data的目录，并将Tools/data下的三个CSV文件移动到新建的data文件夹中。然后使用pandas将在data下的三个csv数据集文件加载到内存中。之后的步骤全都是围绕silence、open和close这三个数据展开的数据处理操作，并最终生成能直接放到模型里训练的数据。

openD = pd.read\_csv('data/open.csv', header = None)

closeD = pd.read\_csv('data/close.csv', header = None)

silenceD = pd.read\_csv('data/silence.csv', header = None)

print(openD)

print(closeD)

print(silenceD)

  根据上一小节的测量结果，我们每个声音需要用30个元组来描述。所以我们新建一个代码块，用一个SAMPLES\_PER\_VOICE变量定义每个声音的元组数量。

  接着我们使用FEATURE\_NUM来表示描述每个声音的频率数量，这里我们之前定义为8。

SAMPLES\_PER\_VOICE = 30

FEATURE\_NUM = 8

  然后我们定义一个processData函数专门用来处理每个声音的数据集，这个函数主要的目的有三个。

  第一是以每个声音的30条数据进行切割，最终生成一个二维矩阵，矩阵的每一行表示一个声音的30条数据。

  第二个是将这些数据进行正规化操作，使得每个数据的范围都控制在[0,1]之间。这样做可以提高训练和模型推理预测的速率。

  第三个是为这些数据打上一个数字的标签。最终生成一个一维标签矩阵。

  我们新建代码块，定义好processData的输入和输出，我们需要向processData中输入数据集本身d（之前使用pandas加载的原始数据集），和他的标签号v。然后processData会为我们输出两个参数，dataX是存放格式化后的数据集，是一个二维数组；dataY是存放标签号的一维数组。

def processData(d, v):

dataX = np.empty([0,SAMPLES\_PER\_VOICE\*FEATURE\_NUM])

dataY = np.empty([0])

return dataX, dataY

  然后我们继续迭代这个函数，正式对传入的数据进行加工。

  首先定义两个变量,data存放收到的所有具体数据的值，dataNum变量用来存放声音的次数，比如我们采集了100次的开数据，那么这个值就是100.

def processData(d, v):

dataX = np.empty([0,SAMPLES\_PER\_VOICE\*FEATURE\_NUM])

dataY = np.empty([0])

data = d.values

dataNum = data.shape[0] // SAMPLES\_PER\_VOICE

return dataX, dataY

  接着我们使用两层的循环来处理数据，内层循环能将每个声音的30个数据进行正规化处理，并且在每次循环完成后将结果存放到一个一维数据tmp中去。

  关于正规化，我们直接将每个数据都除以128.0，使之保持在[0,1]的范围内，这样可以加快后续模型的训练与推理。

  内层循环的正规化操作完成后，在外循环中我们将每次内循环的结果一条条添加到dataX矩阵中去。

  我们使用np.array将tmp这个普通的数组转化为numpy数组参与运算。经过转换后的每个tmp仅仅只描述了一个声音，最终需要添加到dataX矩阵当中去的，所以我们需要先对tmp进行一次np.expand\_dims操作扩充他的维度，为了方便后面的添加。然后我们使用np.concatenate将tmp添加到dataX中去。

  完成了dataX数据的操作后，我们还需要同步生成一个dataY矩阵用于存放标签号。

def processData(d, v):

dataX = np.empty([0,SAMPLES\_PER\_VOICE\*FEATURE\_NUM])

dataY = np.empty([0])

data = d.values

dataNum = data.shape[0] // SAMPLES\_PER\_VOICE

for i in tqdm(range(dataNum)):

tmp = []

for j in range(SAMPLES\_PER\_VOICE):

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][0] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][1] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][2] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][3] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][4] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][5] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][6] / 128.0]

tmp += [data[i \* SAMPLES\_PER\_VOICE + j][7] / 128.0]

tmp = np.array(tmp)

tmp = np.expand\_dims(tmp, axis = 0)

dataX = np.concatenate((dataX, tmp), axis = 0)

dataY = np.append(dataY, v)

return dataX, dataY

  继续新建一个代码块，在这个代码块中我们调用processData生成silenceX、silenceY;openX,openY;closeX,closeY三对独立的数据。

  在这里我们把环境噪声的标签号定义为0，开声音定义为1，关声音定义为2。这些标签可以人为随意定义但是要保证必须从0开始依次递增。

silenceX, silenceY = processData(silenceD, 0)

openX, openY = processData(openD, 1)

closeX, closeY = processData(closeD, 2)

  然后我们新建一个代码块，再使用numpy将它们进行拼接混合，生成完成预处理的数据集dataX，dataY。

dataX = np.concatenate((silenceX, openX), axis = 0)

dataY = np.append(silenceY, openY)

dataX = np.concatenate((dataX, closeX), axis = 0)

dataY = np.append(dataY, closeY)

print(dataX)

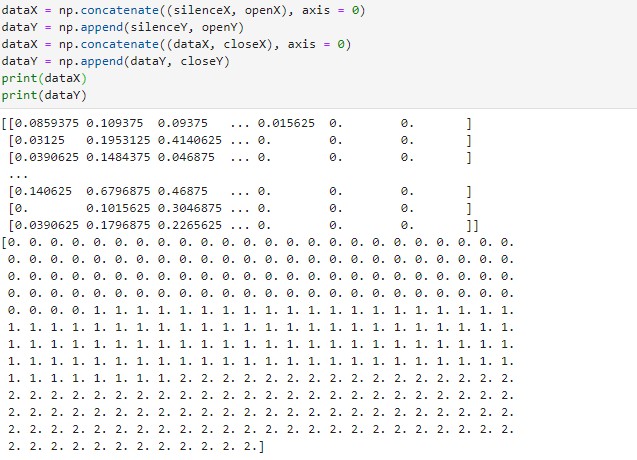
print(dataY)

## 评估数据集和训练数据集

  在模型的正式训练中我们需要准备评估数据集和训练数据集两种数据集。训练用数据集是正常拿来用作数据训练的，而评估用数据集是为了在每一步的训练结束后模拟当前模型的真实使用的数据场景来向模型训练者实时反馈模型的优良情况的，评估用数据集不会直接影响模型的参数。

  我们接下去要做的就是要将目前我们手头上掌握的经过预处理的数据集一分为二，生成一份评估用数据集和一份训练用数据集。那么我们就先来观察目前我们手上的这份已经完成预处理的数据集。

  此时的dataX和dataY有两个特点，首先任意选择dataX的一个下标n的值，取出来的这个值所描述的声音的标签号我们都可以在dataY的下标n中找到。比如dataX的第17个值是用于描述开声音的数据，那么dataY的第17个值就是开声音的标签号。其次，它们目前还有个特点，dataX的前三分之一的数据都在描述环境噪音，中间部分描述开声音，其余部分的数据都在描述关声音，dataY的前三分之一的标签号也都是环境噪音的标签号，其余部分是开声音的标签号,其余部分是关声音的标签号。

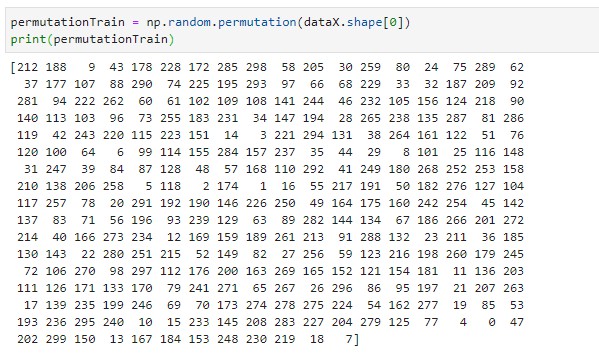


  针对数据集的第二个特点，我们需要先把它进行一次洗牌操作，将数据集的次序打乱，因为我们希望最终的评估用数据集和训练用数据集中都同时等概率含有开、关、环境噪音的数据。

  对于洗牌操作，关键在于随机，我们新建一个代码块将从0到300（所有的声音的总次数）的数随机存放到一个一维数组中。

permutationTrain = np.random.permutation(dataX.shape[0])

print(permutationTrain)

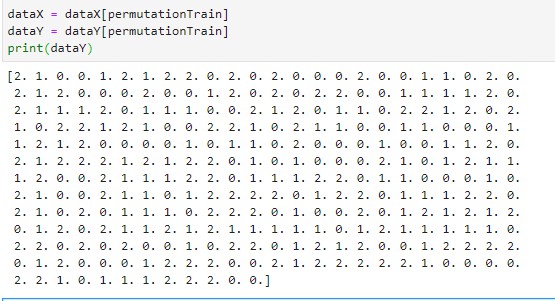


  接着继续新建一个代码块将数据集的标号部分和数据部分同步根据这个生成的一维数组进行洗牌。

dataX = dataX[permutationTrain]

dataY = dataY[permutationTrain]

print(dataY)



  获得到声音随机分布的数据集后，我们新建一个代码块利用八二法则将数据集拆分成评估数据集和训练数据集。其中评估用数据集为2，训练用数据集为8。

vfoldSize = int(dataX.shape[0]/100\*20)

xTest = dataX[0:vfoldSize]

yTest = dataY[0:vfoldSize]

xTrain = dataX[vfoldSize:dataX.shape[0]]

yTrain = dataY[vfoldSize:dataY.shape[0]]

## keras模型搭建和模型训练

  准备完数据集后，我们就可以用keras搭建一个合适的模型进行训练了。

  新建一个代码块，然后用keras生成一个模型。我们需要使用的是Sequential模型，Sequential模型形象地解释就是一个漏斗，数据只能从上层往下层流动，我们可以在这个漏斗中堆叠各种计算的层，以丰富这个模型的结构。

model = keras.Sequential()

  我们继续新建一个代码块，使用keras.layers.Dense定义模型的每一层，在定义模型时，我们一般需要注意这几个问题：

  1.每一层的输入数据的尺寸形态。

  2.每一层的输出数据的尺寸。

  3.每一层采用的激活函数的品种。

  我们先用以下代码直接搭好模型再来解释。

model.add(keras.layers.Dense(32, input\_shape =(FEATURE\_NUM\*SAMPLES\_PER\_VOICE,), activation='relu'))

model.add(keras.layers.Dense(16, activation='relu'))

model.add(keras.layers.Dense(3, activation='softmax'))

  我们先看第一层，首先这个模型一次只能输入一条声音数据，所以输入数据是一个一维的矩阵，内部的数据有8\*30=240条。然后我们使用的激活函数为relu函数。最后的输出的节点尺寸是32。

  然后是第二层，由于是Sequential模型，所以上一层的输出就是下一层的输入。第二层的激活函数我们依然用relu。然后它的输出是16，是上一层输出的一半。

  由于我们最终只有开、关和环境噪音三种声音，所以我们定义最后一层的输出节点尺寸为3.由于这个是一个多分类问题，所以，我们最后使用的激活函数是softmax函数。

  定义好模型结构之后我们还要定义优化器、损失函数、性能评估函数，来编译模型。

  首先是优化器，我们会有一个目标矩阵也就是我们标签定义的那个我们认为正确的矩阵，然后在模型训练的过程中每一步都会出现一个机器预测出来的矩阵。这两个矩阵之间会存在一定的偏差，优化器的作用就是使用某种算法使得机器预测出来的矩阵尽可能通过训练接近于目标矩阵。

  我们使用Adam优化器，这个优化器是一个常用的优化器，主要特点是速度比较快，能够使得模型在较短的迭代周期中产生较好的效果。为了让训练效果更好，我们需要调整优化器的参数，降低学习速率为0.00001。一般学习速率越低，训练的时长越长，但是学习效果越好。

adam = keras.optimizers.Adam(0.00001)

  然后我们还要定义损失函数，损失函数就是用于量化机器预测出来的矩阵和目标矩阵之间的偏差情况。

  在编译函数过程中，我们使用的损失函数是sparse\_categorical\_crossentropy。很显然，这个函数有三个特征分别是sparse、categorical、crossentropy。sparse表示它擅长处理稀疏矩阵，如果声音种类很多，那么一个理想的目标结果必然某一个声音的值概率为1其余都为0的矩阵。categorical表示这个损失函数能处理分类问题，crossentropy表示它用的是交叉熵的方式得出结果和实际值之间的量化差距的。

  最后还要有一个评估函数，评估函数最主要的功能就是调用评估用数据集去计算出模型当前的准确率，给训练者进行汇报。

  我们使用的性能评估函数是sparse\_categorical\_accuracy，最后一个accuracy表示这个性能评估函数能在训练过程中得出准确率数值。

  最后我们新建代码块，使用如下代码进行模型的编译。

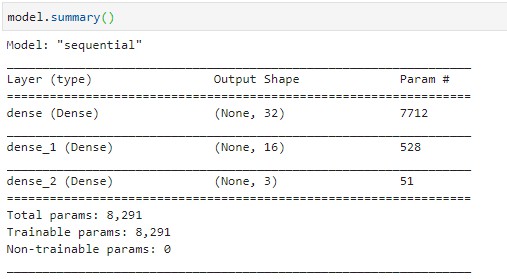
model.compile(loss='sparse\_categorical\_crossentropy',

optimizer=adam,

metrics=['sparse\_categorical\_accuracy'])

  完成编译后，我们就能通过model.summary()输出模型各层的参数状况。

model.summary()



  接下去是模型的训练，我们新建一个代码块，使用以下代码进行模型的训练。将评估数据集和训练数据集填入模型，设置epoch为1000，也就是训练1000步后结束训练。当然这个步数需要根据实际情况进行调整。然后由于数据量比较小，不会占用太多的处理器内存，所以batch\_size设置为1，这些数据会一次性加载到处理器中进行训练。verbose=1表示在训练过程中会有进度条记录输出。

history = model.fit(xTrain, yTrain, batch\_size=1, validation\_data=(xTest, yTest), epochs=1000, verbose=1)

  完成训练后，我们需要将模型冻结导出到磁盘。我们需要先将模型放入到模型转换器中，然后再使用模型转换器将模型转换为TFLite格式。最后将转换的结果从内存中捞出保存到磁盘中。

converter = tf.lite.TFLiteConverter.from\_keras\_model(model)

tflite\_model = converter.convert()

open("model", "wb").write(tflite\_model)

  这个时候我们就可以在文件夹中找到这个模型，当然此时这个模型还不能直接放到开发板上去运行，我们需要进一步将它进行转换，变成能被C语言加载的静态数组形式。

  我们使用linu命令xxd -i来将mode的二进制文件内容存储在C代码静态数组内。当然如果我们需要直接在python3文件夹中写，可以在命令前加入一个!。这里需要注意的是，这个操作不要多次执行，多次运行后model.h文件不会被覆盖，内容只会在一个model.h文件中进行堆叠。

!xxd -i model >> model.h

  最终得到的这个model.h是直接可以移植到开发板上去的模型。

**模型推理**

**加载库文件**

  我们继续打开之前步骤4中收集数据集的代码，一般数据集如何收集的，我们就如何用它来推理，我们不妨先这样来尝试一下。

  首先我们在Arduino中引入一些头文件。

  all\_ops\_resolver.h提供给解释器用于运行模型的操作。micro\_error\_reporter.h用于输出调试信息。micro\_interpreter.h为解释器，它包含处理和运行模型的代码。schema\_generated.h包含TensorFlow Lite FlatBuffer 模型文件格式的加载操作，最后一个version.h提供 Tensorflow Lite 架构的版本信息。

//保留之前代码

#include <Adafruit\_NeoPixel.h>

//添加↓

#include <TensorFlowLite.h>

#include <tensorflow/lite/experimental/micro/kernels/all\_ops\_resolver.h>

#include <tensorflow/lite/experimental/micro/micro\_error\_reporter.h>

#include <tensorflow/lite/experimental/micro/micro\_interpreter.h>

#include <tensorflow/lite/schema/schema\_generated.h>

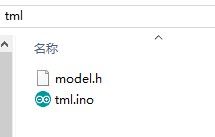
#include <tensorflow/lite/version.h>

//添加↑

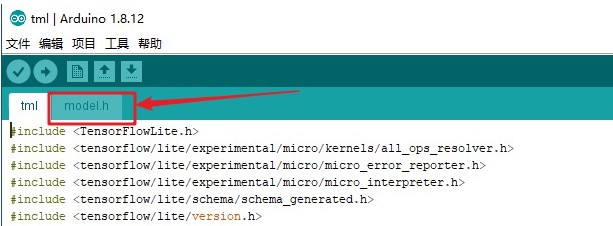
#define PIN 12

//保留之前代码

  然后我们将上一步完成训练的模型从云上下载到本地，并且将它放到和arduino代码在同一个文件夹下，然后重启arduino IDE。



  这时我们会发现Arduino IDE边上多了一个model.h文件。



  然后我们将模型也当作一个库导入其中。

//保留之前代码

#include <tensorflow/lite/version.h>

//添加↓

#include "model.h"

//添加↑

#define PIN 12

//保留之前代码

**创建操作实例**

  导入完所有的数据之后，我们就可以正式在这种单片机板子上编写TFlite的程序了。

  首先我们定义一个tflErrorReporter实例用于设置日志记录，让TFlite解释器记录各种日志。

//保留之前代码

int n=100; //声音的次数

//添加↓

tflite::MicroErrorReporter tflErrorReporter;

//添加↑

const char \* ssid = "\*\*\*\*\*\*";

//保留之前代码

  然后解释器需要一个操作解析器实例对象来访问Tensorflow操作。所以我们要实例化一个操作解析器tflOpsResolver。

//保留之前代码

tflite::MicroErrorReporter tflErrorReporter;

//添加↓

tflite::ops::micro::AllOpsResolver tflOpsResolver;

//添加↑

const char \* ssid = "\*\*\*\*\*\*";

//保留之前代码

  然后我们定义一个模型的指针，一个解释器的指针，一个指向输入数组的指针，以及一个指向输出数组的指针。

//保留之前代码

tflite::ops::micro::AllOpsResolver tflOpsResolver;

//添加↓

const tflite::Model\* tflModel = nullptr;

tflite::MicroInterpreter\* tflInterpreter = nullptr;

TfLiteTensor\* tflInputTensor = nullptr;

TfLiteTensor\* tflOutputTensor = nullptr;

//添加↑

const char \* ssid = "\*\*\*\*\*\*";

//保留之前代码

  然后我们需要预先为输入、输出以及中间数组分配一定的内存。这里我们用常量表达式进行定义，在编译的时候就将TF的运算空间分配好。该预分配的内存是一个大小为 tensorArenaSize 的 byte 数组。这个数组的大小是根据模型的大小来定义的，一般为1024的倍数。在这里我们不妨先定义一个8192长度的空间。从这一步我们也可以感受到这些人工智能模型跑在边缘设备上的一个瓶颈在于边缘设备的内存大小不够。但是一般计算机在设计硬件时会从各个方面中和考虑，内存大小和CPU的性能会存在一定的关系。所以我们很难买到那种内存空间很大，CPU性能较弱的很便宜的那种运行人工智能理想的单片机开发板。

//保留之前代码

TfLiteTensor\* tflOutputTensor = nullptr;

//添加↓

constexpr int tensorArenaSize = 8 \* 1024;

byte tensorArena[tensorArenaSize];

//添加↑

const char \* ssid = "\*\*\*\*\*\*";

//保留之前代码

  接着我们定义一个结果数组，在这个结果数组中每个声音名称的存放次序是有依据的。

//保留之前代码

byte tensorArena[tensorArenaSize];

//添加↓

const char\* VOICES[] = {

"silence",

"open",

"close"

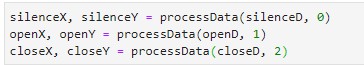
};

//添加↑

const char \* ssid = "\*\*\*\*\*\*";

//保留之前代码

  我们在训练的时候将每个声音赋予了一个标签号。其实这个标签号就可以直接作为现在这个结果数组的下标。



  然后我们为了方便后续的运算，使用NUM\_VOICES来保存声音的种类。

//保留之前代码

const char\* VOICES[] = {

"silence",

"open",

"close"

};

//添加↓

#define NUM\_VOICES (sizeof(VOICES) / sizeof(VOICES[0]))

//添加↑

const char \* ssid = "\*\*\*\*\*\*";

//保留之前代码

  定义好这些初始变量之后，我们来到setup函数中正式加载模型并初始化解释器。

  使用GetModel加载模型，并且检查模型的版本是不是符合要求。

setup(){

//保留之前代码

pixels.show(); //打印

//添加↓

tflModel = tflite::GetModel(model);

if (tflModel->version() != TFLITE\_SCHEMA\_VERSION) {

Serial.println("Model schema mismatch!");

while (1);

}

//添加↑

delay(1000);

}

  然后实例化解释器，我们需要将模型，操作解释器，模型的运行内存空间指针和空间大小，以及日志记录器都传入到解释器中。

setup(){

//保留之前代码

if (tflModel->version() != TFLITE\_SCHEMA\_VERSION) {

//保留之前代码

}

//添加↓

tflInterpreter = new tflite::MicroInterpreter(tflModel, tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);

//添加↑

delay(1000);

}

  解释器实例化完成后，我们需要为模型的输入和输出张量分配内存。

setup(){

//保留之前代码

tflInterpreter = new tflite::MicroInterpreter(tflModel, tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);

//添加↓

tflInterpreter->AllocateTensors();

tflInputTensor = tflInterpreter->input(0);

tflOutputTensor = tflInterpreter->output(0);

//添加↑

delay(1000);

}

**使用模型进行推理**

  我们在之前收集数据集的过程是先判断声音的响度是否超过阈值，如果超过，就开始记录数据，直到记录30条数据后进行数据的输出。

  那么我们就可以将原本要通过串口输出的数据直接输入到模型中去进行推理。

  找到原本输出数据的代码区域进行迭代，我们使用tflInputTensor将一组70条每条6种的数据输入到模型中。当然这些数据在输入之前需要先经过与模型训练时一样的正规化操作。

if(record\_count==record\_num)

{

for(int k=0;k<record\_count-10;k++){

//添加↓

tflInputTensor->data.f[k \* 8 + 0] = (record\_125[k]) / 128.0;

tflInputTensor->data.f[k \* 8 + 1] = (record\_250[k]) / 128.0;

tflInputTensor->data.f[k \* 8 + 2] = (record\_500[k]) / 128.0;

tflInputTensor->data.f[k \* 8 + 3] = (record\_1k[k] ) / 128.0;

tflInputTensor->data.f[k \* 8 + 4] = (record\_2k[k] ) / 128.0;

tflInputTensor->data.f[k \* 8 + 5] = (record\_4k[k] ) / 128.0;

tflInputTensor->data.f[k \* 8 + 4] = (record\_8k[k] ) / 128.0;

tflInputTensor->data.f[k \* 8 + 5] = (record\_16k[k]) / 128.0;

//添加↑

//删除↓

// Serial.print(record\_125[k]);

// Serial.print(",");

// Serial.print(record\_250[k]);

// Serial.print(",");

// Serial.print(record\_500[k]);

// Serial.print(",");

// Serial.print(record\_1k[k]);

// Serial.print(",");

// Serial.print(record\_2k[k]);

// Serial.print(",");

// Serial.print(record\_4k[k]);

// Serial.print(",");

// Serial.print(record\_8k[k]);

// Serial.print(",");

// Serial.print(record\_16k[k]);

// Serial.println("");

//删除↑

}

record\_count=-1;

}

  一次声音的数据输入完毕后我们就使用Invoke进行推理。

if(record\_count==record\_num)

{

for(int k=0;k<record\_count-10;k++){

//保留之前代码

}

//添加↓

TfLiteStatus invokeStatus = tflInterpreter->Invoke();

if (invokeStatus != kTfLiteOk) {

Serial.println("Invoke failed!");

while (1);

return;

}

//添加↑

record\_count=-1;

}

  完成推理后，我们将推理结果进行输出,我们用tflOutputTensor来取出这三个声音的准确率。

if(record\_count==record\_num)

{

//保留之前代码

if (invokeStatus != kTfLiteOk) {

//保留之前代码

}

//添加↓

for (int i = 0; i < NUM\_VOICES; i++) {

Serial.print(VOICES[i]);

Serial.print(": ");

Serial.println(tflOutputTensor->data.f[i], 6);

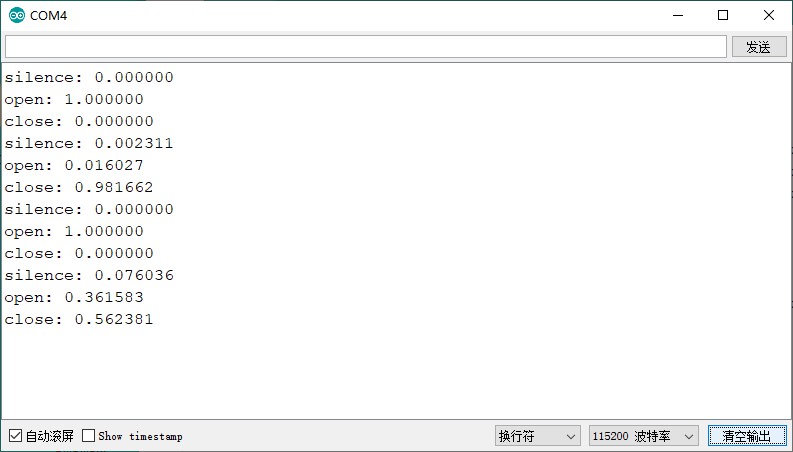
}

//添加↑

record\_count=-1;

}

  最终我们可以在串口监视器内观看到声音识别的结果，是不是很神奇呢。



**接上LED灯**

  我们可以使用LED灯来真实反映语音识别的结果，我们首先删去之前收集数据集时的灯光效果。

//保留之前代码

void Check\_Start(){

bsum=0;

for(int j=0;j<8;j++){

bsum=bsum+bands[j];

}

if(bsum>=Threshold\_HIGH&&record\_count==-1){

data\_count++;

//删去↓

// if(data\_count==n){

// pixels.setPixelColor(0, pixels.Color(255,0,0)); //调颜色

// pixels.show(); //打印

// }else if(data\_count>n){

// pixels.setPixelColor(0, pixels.Color(0,0,255)); //调颜色

// pixels.show(); //打印

// }

//删去↑

record\_count=0;

}

//保留之前代码

}

//保留之前代码

  然后我们在推理结束后对推理结果进行判断，从而控制LED的开关。

void Check\_Start(){

if(record\_count==record\_num)//收集完成一次声音的30个元组数据

{

//保留之前代码

for (int i = 0; i < NUM\_VOICES; i++) {

//保留之前代码

}

//添加↓

float silence=tflOutputTensor->data.f[0];

float open=tflOutputTensor->data.f[1];

float close=tflOutputTensor->data.f[2];

if(silence==1){

}else{

if(open>close&&open>0.6){

pixels.clear(); //调颜色

pixels.show(); //打印

pixels.setPixelColor(0, pixels.Color(255,255,255)); //调颜色

pixels.show(); //打印

}else if(close>open&&close>0.6){

pixels.clear(); //调颜色

pixels.show(); //打印

pixels.setPixelColor(0, pixels.Color(0,0,0)); //调颜色

pixels.show(); //打印

}

}

//添加↑

record\_count=-1;

}

//保留之前代码

}

**热词唤醒**

  到目前为止，我们确实可以将单个字的声音分得很准确。

  但是我们仔细思考整个识别的过程，我们会发现一个问题。我们是使用声音的响度来判断是否说话的，一但说了话以后立即采集且仅采集30条声音数据去做推理。所以这套程序只能采集并识别话音的起始那一个字。

  而我们平时说话很显然并不是使用单个字的发音进行沟通交流的。一句话可以由好几个字所构成。机器认识的那个字，并不一定是这句话的第一个字，就比如芝麻开门这句话，开是位于中间位置的。那么对于这种情况应该怎么处理呢？

  其实上述程序的缺陷在于一定要凑齐最新的30帧才能去推理，并且需要用响度超过阈值去触发单个字的识别。那我们其实可以更新这套程序，使之持续不停地进行识别，不用阈值去触发，并且也不用凑齐最新的30帧，使用之前参与过推理的旧的几帧数据外加最新的5、6帧数据凑齐30帧进行推理。

  这样有一个好处，我们至少会有一次30帧数据将开字的声音在一句联系的话中框起来，然后推理得到一个开字的高概率结果，从而在一句话内识别到这个字。



  那么我们接下去对代码进行修改。我们删去Check\_Start和Send\_Msg,然后加入move\_Inputindex函数用于不停移动这30帧数据去框这些声音，等到move\_Inputindex移动一定的步长的时候会将sign设置为1，通知识别程序可以进行识别了。识别程序写在detection函数中,完成识别后需要使用这个识别程序进行应用层面的操作，所以所有开灯关灯的程序都会在application函数中。

void loop() {

//保留之前代码

constrain128();

//添加↓

move\_Inputindex();

if(sign==1){

detection();

application();

sign=0;

}

//添加↑

//删除↓

// Check\_Start();

// Send\_Msg();

//删除↑

}

  然后我们继续将代码进行完善。

//保留之前代码

void constrain128(){

//保留之前代码

}

//保留之前代码

//添加↓

void move\_Inputindex(){

if(Inputindex<SAMPLE\_NUMS){

for(int bands\_index=0;bands\_index<8;bands\_index++){

Inputbands[Inputindex][bands\_index]=(int)bands[bands\_index];

}

Inputindex=Inputindex+1;

if(Inputindex==SAMPLE\_NUMS){

sign=1;

}

}else{

steps=steps+1;

if(steps==MAX\_STEP+1){

for(int mo=0;mo<SAMPLE\_NUMS-MAX\_STEP;mo++){

for(int bands\_index=0;bands\_index<8;bands\_index++){

Inputbands[mo][bands\_index]=Inputbands[mo+MAX\_STEP][bands\_index];

}

}

steps=0;

}

else{

for(int bands\_index=0;bands\_index<8;bands\_index++){

Inputbands[SAMPLE\_NUMS-1-MAX\_STEP+steps][bands\_index]=(int)bands[bands\_index];

}

if(steps==MAX\_STEP){

sign=1;

}

}

}

}

void detection(){

for(int i=0;i<SAMPLE\_NUMS;i++){

for(int j=0;j<8;j++){

tflInputTensor->data.f[i \* 8 + j] = Inputbands[i][j]/ 128.0;

}

}

TfLiteStatus invokeStatus = tflInterpreter->Invoke();

if (invokeStatus != kTfLiteOk) {

Serial.println("Invoke failed!");

while (1);

return;

}

}

void application(){

float silence=tflOutputTensor->data.f[0];

float open=tflOutputTensor->data.f[1];

float close=tflOutputTensor->data.f[2];

if(silence==1){

}else{

for (int i = 0; i < NUM\_VOICES; i++) {

Serial.print(VOICES[i]);

Serial.print(": ");

Serial.println(tflOutputTensor->data.f[i], 6);

}

if(open>close&&open>0.6){

Serial.print("open");

Serial.println(open);

Serial.println(close);

pixels.clear(); //调颜色

pixels.show(); //打印

pixels.setPixelColor(0, pixels.Color(255,255,255)); //调颜色

pixels.show(); //打印

}else if(close>open&&close>0.6){

Serial.print("close");

Serial.println(close);

Serial.println(open);

pixels.clear(); //调颜色

pixels.show(); //打印

pixels.setPixelColor(0, pixels.Color(0,0,0)); //调颜色

pixels.show(); //打印

}

}

}

//添加↑

  然后我们加入一些必要的变量。其中SAMPLE\_NUMS表示我们定义的一个识别声音的框需要多少条数据，MAX\_STEP表示这个框框每次向后移动的步长，这个值如果大了，可能会影响效果，如果小了，就会导致识别过于频繁从而影响采样。

//保留之前代码

#include "model.h"

//添加↓

#define MAX\_STEP 5

#define SAMPLE\_NUMS 30

//添加↑

#define PIN 12

//保留之前代码

byte tensorArena[tensorArenaSize];

//添加↓

int sign=0;

int Inputindex=0;

int Inputbands[SAMPLE\_NUMS][8]={0};

int steps=MAX\_STEP;

//添加↑

const char\* VOICES[] = {

"silence",

"open",

"close"

};

//保留之前代码

  最后，我们可以尝试对其进行喊出诸如芝麻开灯之类的话，就能做到热词唤醒的功能。

**尾声**

  我们只是收集了开和关的数据集，其实在中文语音中，所以的字都可以分解为有限几种的拼音音素，我们只要收集这些拼音音素的数据集，就能囊库所有的字的发音，这个是我们可以不断探索和尝试的。