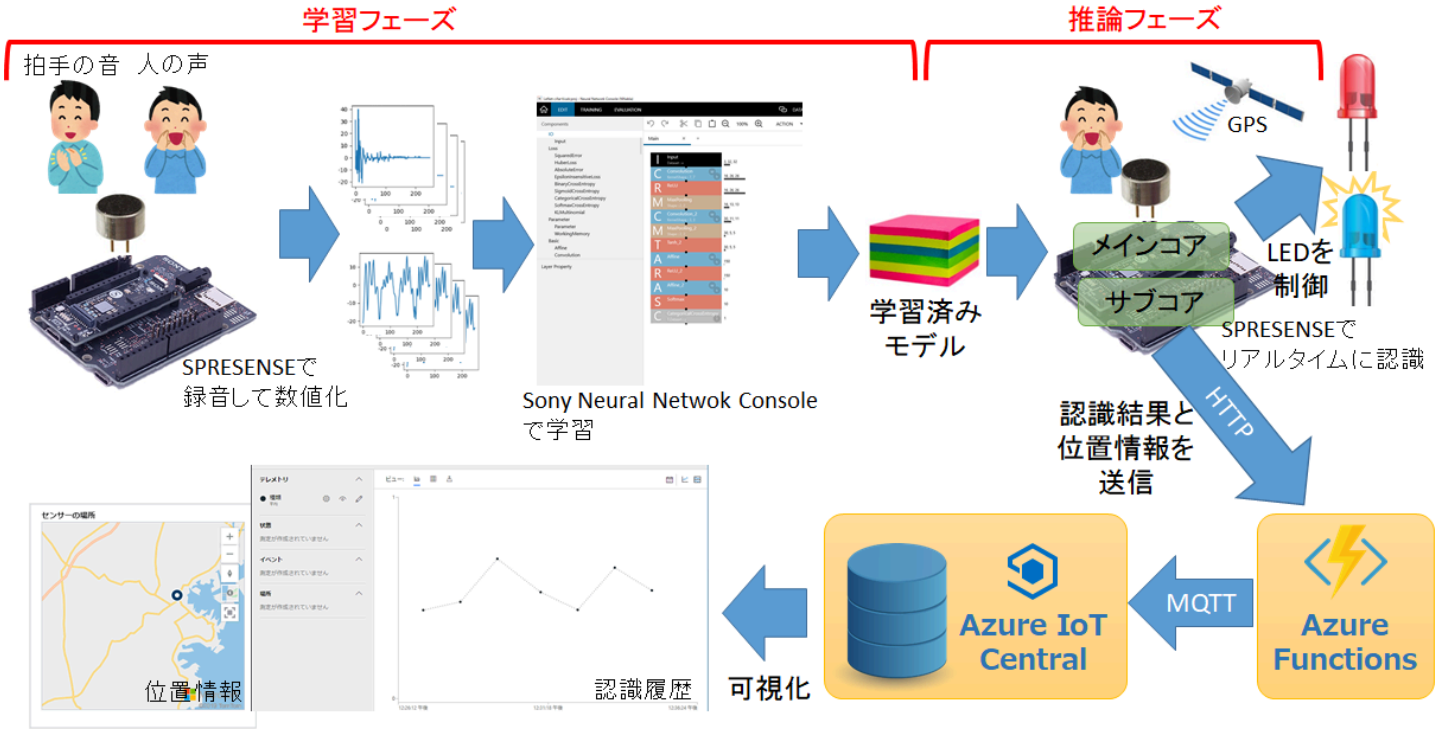
SPRESENSE＋ニューラルネットワークで音声推論！軽量AI+IoTハンズオン



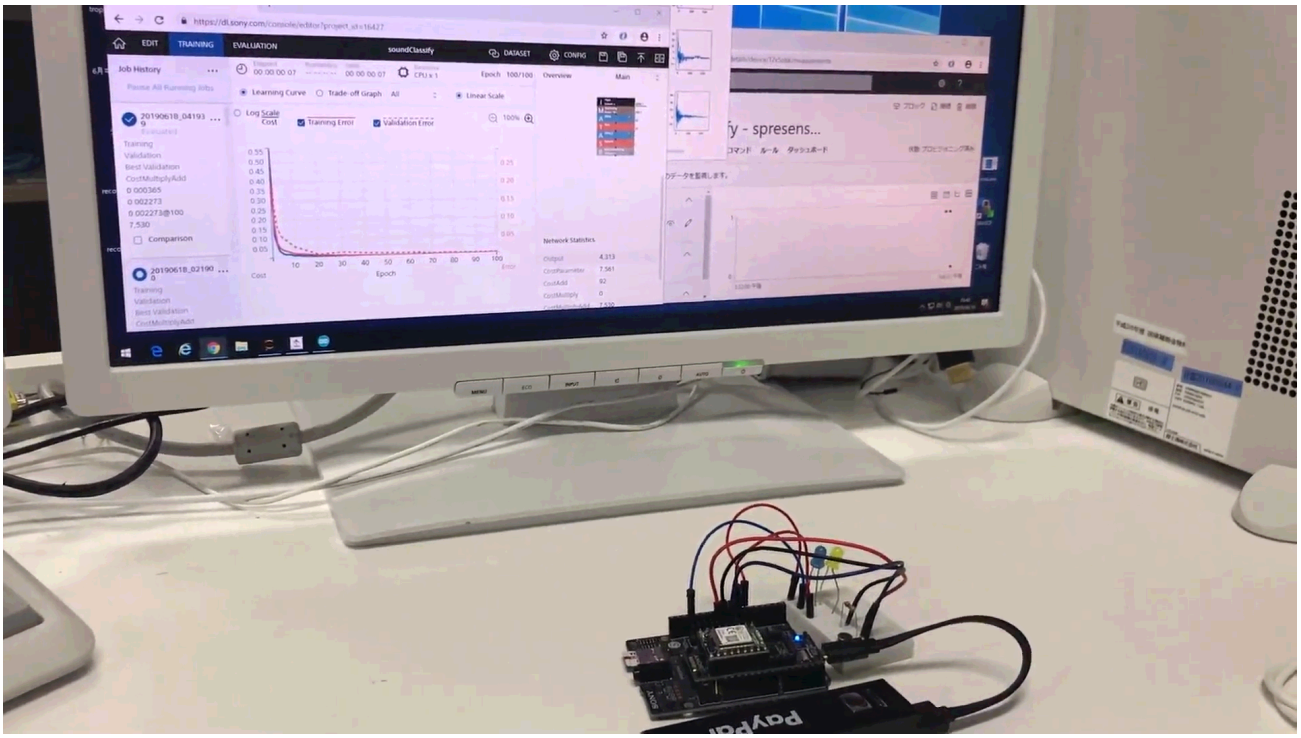
本資料の前提

- ●Aruduino IDE + SPRESENSE Arduinoライブラリ1.4.0以上 をインストールしたPCがあること
  (インストール手順は こちら を参照)
- ●Neural Network Console Cloud(https://dl.sony.com/console)のアカウントを取得していること
- ●Microsoft Azure(https://portal.azure.com/)のアカウントを取得していること

目次

1. 今日作るもの

このハンズオンでは、マイクから入力した音声をリアルタイムに認識し、拍手の音なら青(緑)のLED、人の声なら黄色のLEDを点灯させるIoTシステムを作成します。また、結果の履歴をグラフ化します。

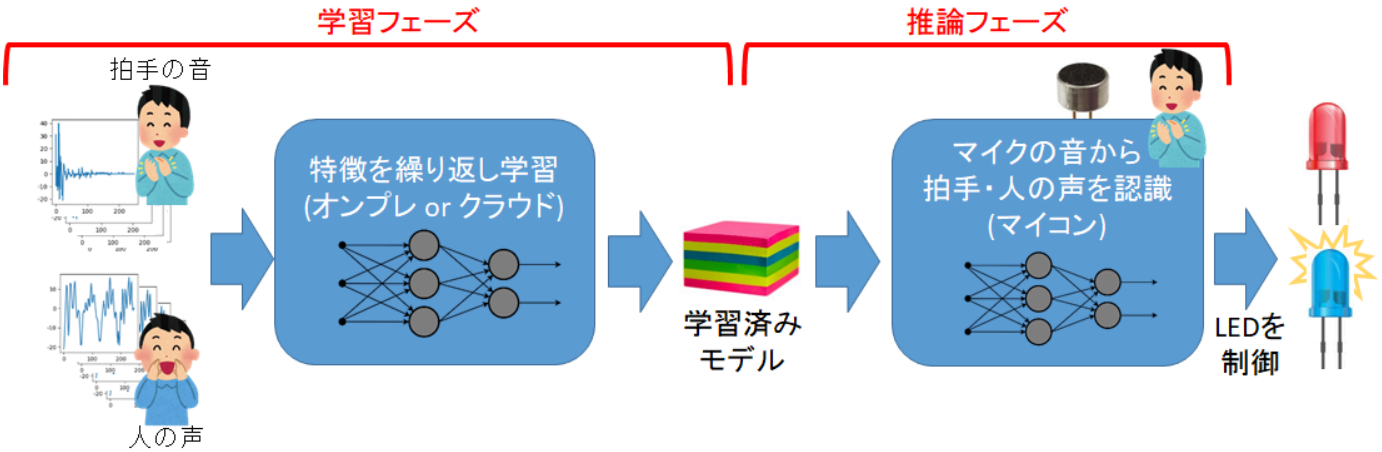


(↑ 画像をクリックすると動画が再生されます ↑)

→今回のハンズオンはArduino IDEで開発しますので、応用で様々な電子パーツ(サーボモータやディスプレイ)も簡単に利用できます。例えば拍手の音なら合いの手を打つといった仕組みも簡単に作れます。
→今回のハンズオンでは実際の音を録音して学習する部分も含めて実習しますので、工場の異常音検知システムなどにもすぐに応用できます。

2. システムの主要な部分の全体像

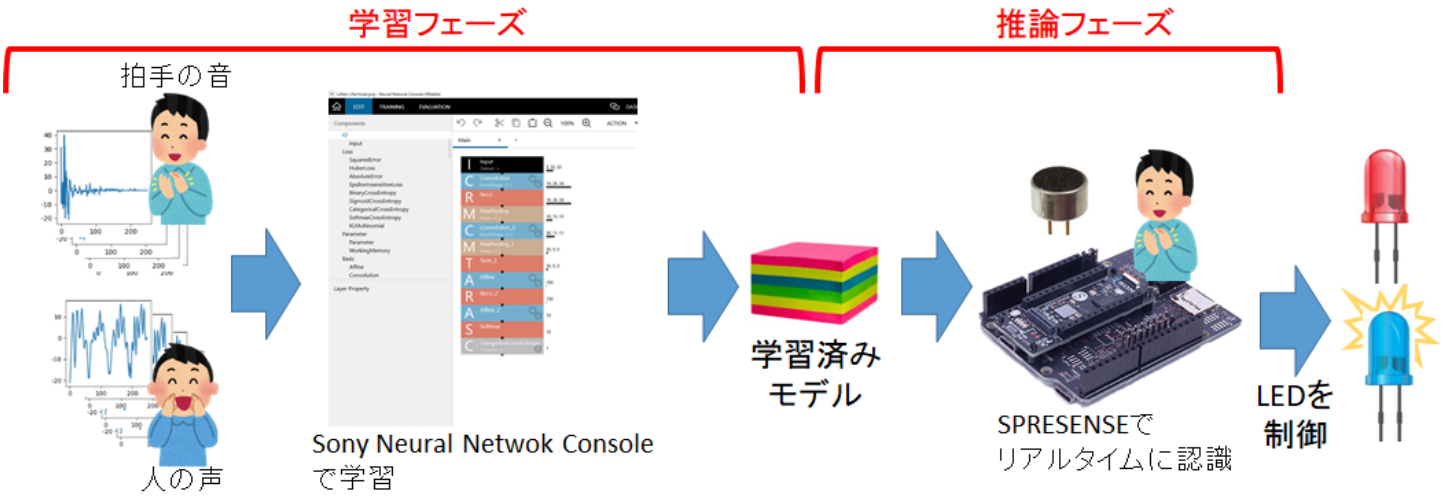
このような音声認識を実現する一つの方法としてディープラーニングがあります。ディープラーニングを利用した実現の流れは下図の通りです。



認識したい音(今回は拍手と人の声)を大量に用意し、これらから音の特徴を見つける「学習フェーズ」と、学習した特徴データ(モデル)を使ってマイクから入力された音が何なのかを推測する「推論フェーズ」に分かれます。
ディープラーニングの仕組み(参考)上、学習も推論も膨大な計算量が必要ですが、推論に関しては近年、軽量・低消費電力で実現できる製品が多数登場しています。SPRESENSEもその一つです。

推論に比べて計算量が桁違いに多い学習フェーズは、多くの場合PCやワークステーションのハイエンドGPUを利用します。ただし、高価な割に使用頻度が低く(学習フェーズでしか使わない)、また世代交代も激しいため、クラウドサービスを利用する事も多々あります。

そこで今回は学習フェーズにはSonyが提供するNeural Network Console(以降NNC)のクラウド版サービスを利用し、推論フェーズにSPRESENSEを利用します。



3. 音声データの用意

学習を行うためには、まず大量の音声データが必要です。音声のデータセットには公開されているものがいくつかあり、たとえばESC-50は環境音を50種類、2,000ファイル集めたデータセットです。

これらを利用してもよいのですが、認識の精度を上げたい場合、推論を行う環境と極力同じ環境で音声を集めた方が良い結果になります。

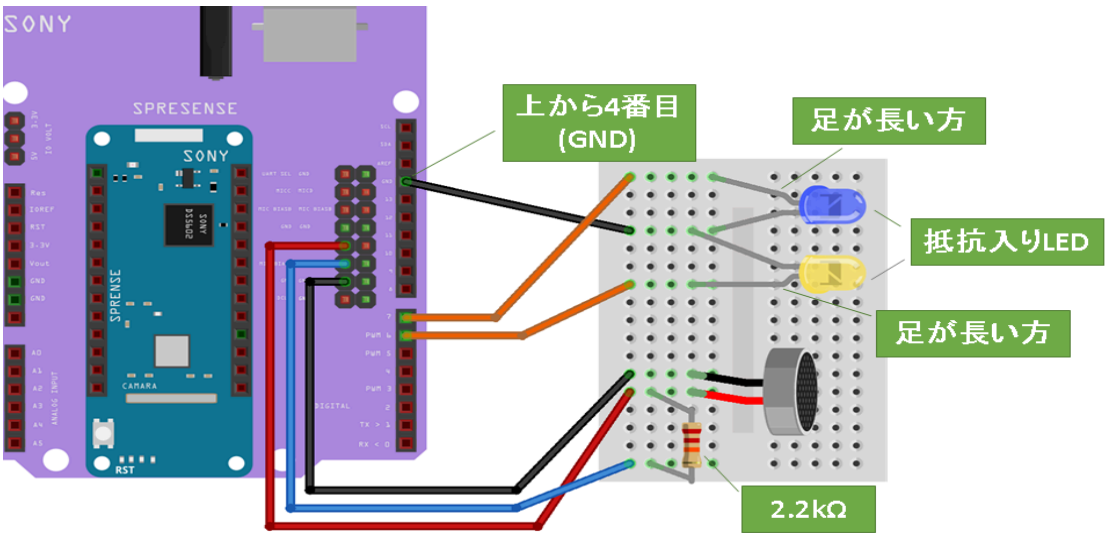
そこで今回はSPRESENSE上で録音プログラムを作成し、実際に拍手・人の声を録音してファイル化します。データ数は多いほうがよいですが、今回は時間の都合上それぞれ20個とします。

SPRESENSEには最大4個のアナログマイクまたは最大8個のデジタルマイクを接続できますが、今回はアナログマイク1個のみ接続します。ついでに、録音状態を知らせるため(かつ、推論時には推論結果を示すため)のLEDも2個接続しておきます。

アナログマイクには通常のコンデンサマイクを利用できますが、バイアス電圧を2.2kΩの抵抗を挟んでつなげる必要があります。LEDは拡張ボードのピンがArduino互換となっているため簡単に接続できます。

★作業1★ コンデンサマイクとLEDの接続

●ブレッドボードを利用して以下のような回路を作成して下さい



※マイク接続ピン配列についてはこちらを、Arduino互換ピン配列についてはこちらを参照願います

では、まずは開発環境の確認を兼ねてLチカ(LEDチカチカ)のプログラムを作ってみましょう。

★作業2★ Lチカプログラムの作成と実行

●SPRESENSEの本体側のMicroUSB端子とPCのUSB端子をUSBケーブルで接続して下さい

●Arduino IDEを起動し、以下のプログラムを入力して下さい(ファイル名は LChika など)

```
#define PIN1 7     //青LED接続ピンの定義
#define PIN2 6     //黄LED接続ピンの定義

//初期化処理
void setup()
{
  //LEDを初期化(消灯)
  pinMode(PIN1, OUTPUT);     //PIN1を出力用に設定
  pinMode(PIN2, OUTPUT);     //PIN2も出力用に設定
  digitalWrite(PIN1, LOW);   //PIN1をOFF
  digitalWrite(PIN2, LOW);   //PIN2もOFF
}

//メインループ
void loop()
{
  digitalWrite(PIN1, HIGH);   //PIN1をON
  digitalWrite(PIN2, LOW);    //PIN2をOFF
  usleep(100000);             //0.1秒(100,000μ秒)待つ
  digitalWrite(PIN1, LOW);    //PIN1をOFF
  digitalWrite(PIN2, HIGH);   //PIN2をON
  usleep(100000);             //0.1秒(100,000μ秒)待つ
}
```

●「ツール」→「ボード」で「Spresense」を選び、「ツール」→「Core」で「MainCore」を選び、「ツール」→「シリアルポート」で「COMx (Spresense)」を選びます(Macの場合は ～USBtoUARTのような名前です)

※シリアルポートに選択肢が出てこない場合は、シリアルポートドライバが入っていないか、利用してるUSBケーブルが充電専用タイプの可能性があります。シリアルポートドライバが入っていない場合はこちらを参考にUSB シリアルドライバをインストールしてください。

●「ツール」→「書込装置」→Spresense Firmware Updaterを選択し、「ツール」→「ブートローダを書き込む」を選択してブートローダーを書き込みます(一度だけ)

●[マイコンボードに書き込む]ボタンをクリックし、プログラムの書き込みと実行を確認します

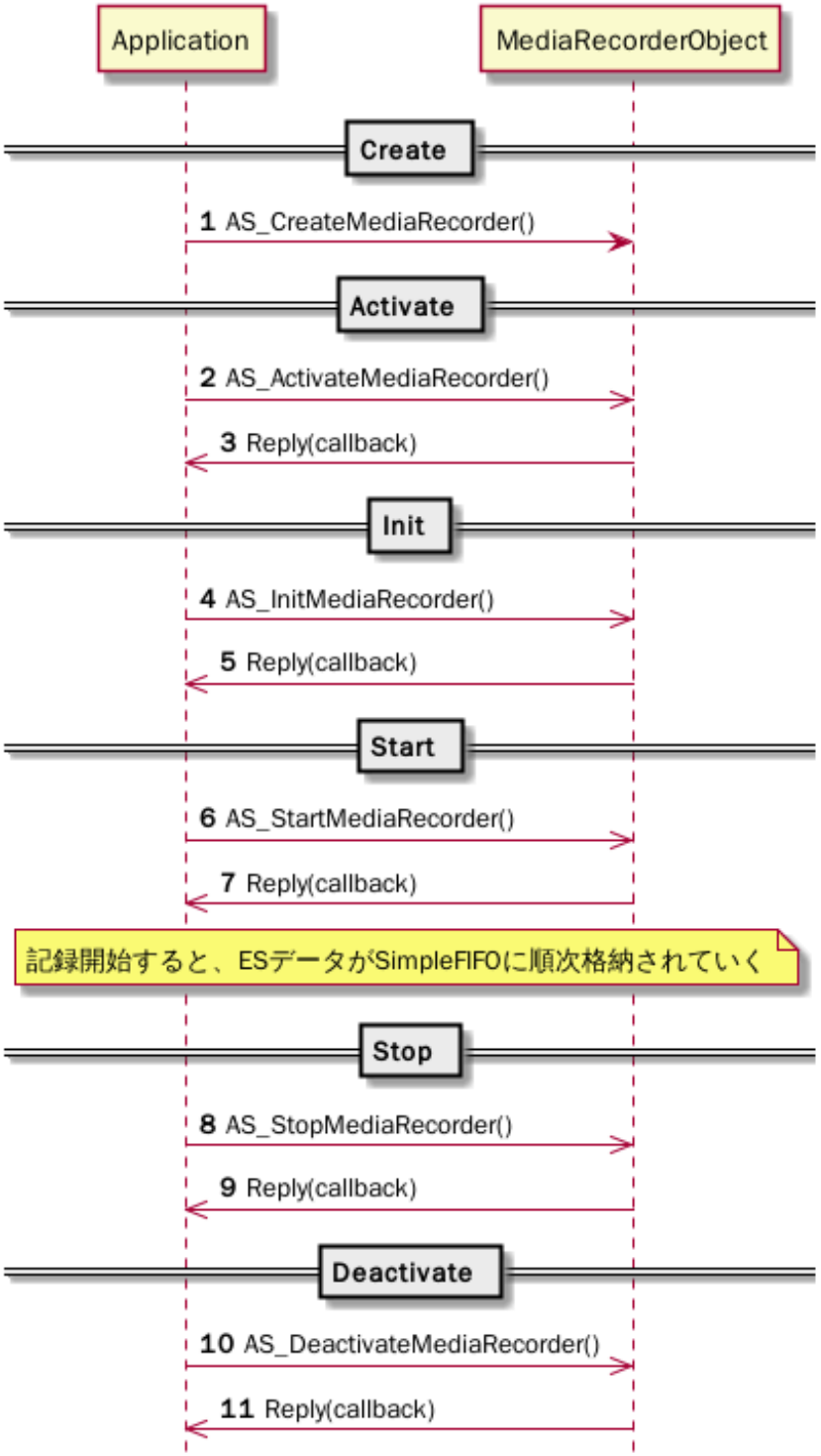
※回路・プログラムが正しければ2つのLEDが交互に点滅します。うまくいかなければ見直しましょう

次に、マイクを使った録音プログラムを作成します。

今回の処理は単純な録音ではなく、録音データをリアルタイムに処理し、一定以上の振幅(音量)があった場合に、そこから0.5秒だけ切り出してファイルに保存する、という処理を行います。これにより録音された音声ファイルの時間軸が揃うため認識精度も上がります。

このようなリアルタイムの音声データ(＝サンプリングされた数値列)処理を行うために、SPRESENSE SDKでは MediaRecorderクラスが提供されています。

MediaRecorderクラスは下図のような流れで利用します(公式SDKガイドから引用)。



詳しくは公式ガイドを参照して下さい。基本的にはbegin → active → initメソッドを呼び出して準備を完了させた後、startメソッドを呼び出すと録音が開始され、バックグラウンドで録音データ(波形を数値化したデータ)が内部バッファ(SimpleFIFO)に書き込まれていきます。

内部バッファに溜まっているデータを取り出すには、readFramesメソッドを呼び出します。内部バッファの容量には限界があるので、頻繁にreadFramesを呼び出してバッファがあふれる前にデータを取り出す必要があります。

★作業3★ マイク音声録音プログラムの作成と実行

●Arduino IDEで以下のプログラムを入力して下さい(ファイル名は MicRec など)

```
#include <SDHCI.h>
#include <MediaRecorder.h>
#include <MemoryUtil.h>

//定数パラメタ定義
const String FILEPREFIX = "sound_"; //録音ファイル名の接頭文字
const int    RECFILECNT = 2;        //録音ファイル数
const float  RECTIME = 0.5f;        //録音時間
const int    REC_THRESHOLD = 250;   //録音を開始する振幅の閾値

//LED接続ピン設定
#define PIN1 7    //青LED接続ピン
#define PIN2 6    //黄LED接続ピン

//録音機能を実現するクラス
MediaRecorder *theRecorder;

//SDカードアクセス用のクラス
SDClass theSD;

//録音データー時格納用バッファ
const int32_t buffer_size = 4096;
uint8_t       s_buffer[buffer_size];

//WAVデータ格納用バッファ(48KHz・モノラルで RECTIME秒録音するために必要なサンプル数)
const int recSize = (int)(48000 * RECTIME);
short wavData[recSize];             //16bitで録音するので各サンプルはshort型
int recIdx = 0;

//録音回数カウント用
int fileCnt = 0;

//MediaRecorderのイベント(録音開始とか終了とか)処理用コールバック関数
static bool done_callback(AsRecorderEvent event, uint32_t result, uint32_t
 sub_result){
  //今回は特にやりたい事がないので、引数の値を適当に出力するだけ
  printf("## MediaRecorder Event : %x %x\n", event, result);
  return true;
}

//初期化処理
void setup()
{
  //LEDを初期化(消灯)
  pinMode(PIN1, OUTPUT);
  pinMode(PIN2, OUTPUT);
  digitalWrite(PIN1, LOW);
  digitalWrite(PIN2, LOW);
  theSD.begin();   //SDカードアクセスの準備
```

```
  //録音機能に必要なメモリを確保する
  initMemoryPools();
  createStaticPools(MEM_LAYOUT_RECORDER);

  //録音機能(MediaRecorder)を初期化
  theRecorder = MediaRecorder::getInstance();
  theRecorder->begin();

  //録音用のクロックを標準に設定
  theRecorder->setCapturingClkMode(MEDIARECORDER_CAPCLK_NORMAL);

  //アナログマイクからの入力に設定・イベントコールバック関数を設定
  theRecorder->activate(AS_SETRECDR_STS_INPUTDEVICE_MIC, done_callback);
  theRecorder->setMicGain(210);    //マイクの音量調整(0～210)

  //マイクの準備完了を少し待つ
  usleep(100000);

  //録音設定(WAV形式・モノラル・48KHz・16bit)
  theRecorder->init(AS_CODECTYPE_WAV, AS_CHANNEL_MONO,
 AS_SAMPLINGRATE_48000, AS_BITLENGTH_16, 0);
  //録音開始！
  theRecorder->start();
  puts("Recording Start!");
}

//メインループ
void loop()
{
  uint32_t size = 0;

  //録音データをs_bufferに取り出す
  err_t err = theRecorder->readFrames(s_buffer, buffer_size, &size);

  //エラーだった場合(OK・バッファ不足以外の場合)は終了処理(done)に飛ばす
  if(err != MEDIARECORDER_ECODE_OK && err !=
 MEDIARECORDER_ECODE_INSUFFICIENT_BUFFER_AREA)
  {
    puts("Recording Error!");
    done();
  }
  //録音データが取得できた場合、録音データ処理(signal_process)に飛ばす
  else if(size > 0)
  {
    signal_process(size);
    //RECFILECNT分の録音が完了していたら終了処理(done)に飛ばす
    if(fileCnt > RECFILECNT)
      done();
  }
}
```

```
//録音データ処理
void signal_process(uint32_t size)
{
  //録音データを16bit符号付き整数として扱えるようにキャスト
  short *p = (short*)s_buffer;

  //１サンプルごとのループ(１サンプル16bitなのでループ回数はsize/2)
  for(int i = 0; i < size/2; i++)
  {
    //録音開始前(WAVデータとして取り出し始める前)かつ振幅がREC_THRESHOLDを超えたら
    if(recIdx == 0 && abs(p[i]) > REC_THRESHOLD)
    {
      //録音開始(WAVデータとして取り出し始める)
      puts("Record to File Start");
      wavData[recIdx] = p[i];
      recIdx++;
      //青色LED点灯
      digitalWrite(PIN1, HIGH);
    }
    //録音中かつ録音時間がRECTIME未満の場合
    else if(recIdx != 0 && recIdx < recSize)
    {
      //サンプル値を取り出してwavDataに格納
      wavData[recIdx] = p[i];
      recIdx++;
    }
    //録音時間がRECTIMEになった(＝RECTIME秒ぶんWAVデータの取り出しが完了した)場合
    else if(recIdx == recSize)
    {
      puts("Record to File End");
      //録音停止
      theRecorder->stop();

      //青色LED消灯・黄色LED点灯
      digitalWrite(PIN1, LOW);
      digitalWrite(PIN2, HIGH);

      //拡張子を除くファイル名を作成しファイル書き込み処理(writeFile)を呼び出す
      String file = FILEPREFIX + fileCnt;
      writeFile(file.c_str(), wavData, recSize);

      //次の録音開始に備える
      fileCnt++;
      recIdx = 0;

      //黄色LED消灯
      digitalWrite(PIN2, LOW);
      //録音再開
      theRecorder->start();
    }
  }
}
```

```
//ファイル書き込み処理(wav, csv)
void writeFile(const char* file, short* buffer, int size){
  File f;
  char str[10];

  //書き込むためのWAVファイルを開く
  String wavFileName = String(file)+".wav";
  theSD.remove(wavFileName);      //同名のファイルがあったら先に削除しておく
  f = theSD.open(wavFileName, FILE_WRITE);
  puts("Saving to WAV File");

  //WAVファイルのヘッダを出力
  theRecorder->writeWavHeader(f);
  //WAVデータ本体を書き込んでファイルを閉じる
  f.write((uint8_t*)buffer, size*2);
  f.close();
}

//全録音終了処理
void done()
{
  //録音停止と録音オブジェクトの後片付け
  theRecorder->stop();
  theRecorder->deactivate();
  theRecorder->end();

  puts("End Recording");
  exit(1);
}
```

●SPRESENSEに**microSDカードを挿入して下さい**

●[マイコンボードに書き込む]ボタンをクリックし、プログラムの書き込みと実行を確認します
※書き込み完了後に「ツール」→「シリアルモニタ」を開いてデバグ情報を表示させて下さい
※シリアルモニタが文字化けする場合は、右下の速度指定を「**115200 bps**」に変更してください

●実行開始後、マイクに一定以上の音を流す(拍手など)と青LEDが点灯して0.5秒間だけ録音され、青LEDは消灯します。録音データをWAVファイルに書き込んでいる間は黄色LEDが点灯します

●2回録音されるとプログラムは終了しますので、MicroSDカードを抜き出してPCに接続し、「sound_0.wav」「sound_1.wav」を再生してみて下さい

このプログラムで重要なのは signal_process関数です。この関数は内部バッファからデータを取り出す度にメインループから呼び出しています。取り出したデータ(＝波形データの一部)は配列s_bufferに入っていますので、一定の振幅(音量)以上になった地点から0.5秒抜き出し、配列wavDataに格納しています。そして、0.5秒分のデータが揃った所でSDカードにWAVファイルとして出力しています。

また、録音中は内部バッファが溢れないように短時間で処理を終わらせる必要がありますが、ファイルにデータを書き出す処理はどうしても時間がかかります。そこで、一旦録音を停止(stop)させてからファイルに書き込み、書き込み完了後に再度録音開始(start)させています。

これで録音機能は実現できましたが、実はこのままではNNCの学習データとしては使えません。
NNCでは、画像以外のバイナリデータを学習データとして利用する場合、数値を文字列化して出力したCSVファイルにしておく必要があります。また、数値は-1.0～1.0の実数値に正規化した方がよいです。
また、最終的にSPRESENSEで推論させる事を考えた場合、メモリ制約のため大きな学習モデルは利用できません。48KHzのデータは入力データとして大きいため、データを間引いて8KHz程度にします。
WAVファイル→CSVファイルへの変換用に別途プログラムを作ってもよいですが、今回は先ほどのプログラムを書き換え、WAVファイルへの出力と同時にCSVファイルへの出力も行うようにしましょう。

★作業4★ **CSV書き込み処理の追加**

●作業3で作ったプログラム内の writeFile関数の最後(f.close();の後)に以下の処理を追加して下さい

```
//書き込むためのCSVファイルを開く
String csvFileName = String(file)+".csv";
theSD.remove(csvFileName);      //同名のファイルがあったら先に削除しておく
f = theSD.open(csvFileName, FILE_WRITE);
puts("Saving to CSV File");

//データの１／６(＝8KHzのデータとして)をfloat型の数値にして書き込む
for(int i = 0; i < size; i+=6){
  //値を実数文字列化する(32768ではなく4096で割っているのは元の音量が小さいため)
  sprintf(str, "%0.4f", (float)buffer[i]/4096.0);
  f.println(str);
}
f.close();
```

これで準備が整いましたので、実際に拍手の音と人の声をそれぞれ20個ずつ録音しましょう。

★作業5★ 拍手×20個と人の声×20個の録音作業

●作業4で作ったプログラムの6･7行目を以下のように書き換えて実行して下さい

```
const String FILEPREFIX = "clap_";   //録音ファイル名の接頭文字
const int    RECFILECNT = 20;        //録音ファイル数
```

●周りの音がなるべく入らないように注意しながら拍手を20回録音して下さい

※拍手1回だと先頭部分が録音されないので、「バンバン」といった感じで2回拍手(0.5秒以内)するとよいです

※黄色のLEDが点灯している間はmicroSDカードへの書き込み中なので録音されません。むやみに手を叩かず、全ての**LED**が消灯している状態で手を叩き始め、青**(**緑**)**の**LED**が点灯している間**(0.5**秒**)**に2回叩き終わるようにしましょう。精度の高いAIには同じ条件のデータが揃っている事が重要です

●作業4で作ったプログラムの6行目を以下のように書き換えて実行して下さい

```
const String FILEPREFIX = "voice_";   //録音ファイル名の接頭文字
```

●周りの音がなるべく入らないように注意しながら声を20回録音して下さい

※声は同じ音の方がいいのでマイクに向かって「わっ！」と言って下さい

●SPRESENSEから microSDカードを抜いてPCに接続し、「clap_0.csv」～「clap_19.csv」および「voice_0.csv」～「voice_19.csv」がある事を確認して下さい

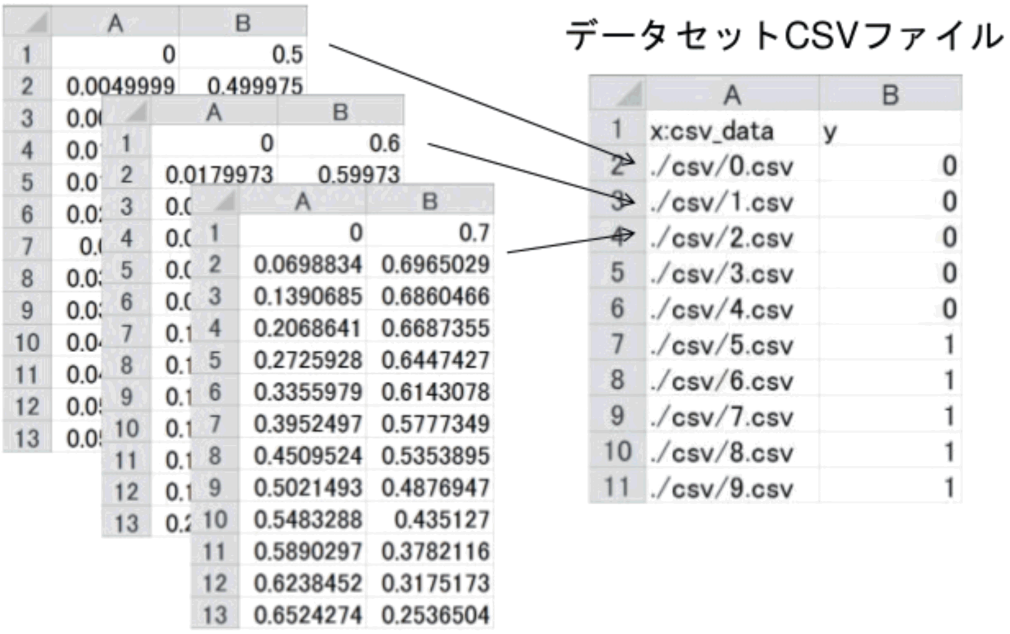
※もしうまく録音ができなかった場合は こちらで用意したこのファイル を解凍して使って下さい。

4. Neural Network Console Cloudによる音声データの学習

Sonyが提供するNeural Network Console(NNC)は、数学的な知識やプログラミングの知識がなくても、直感的な作業でディープラーニングを実現できる(＝学習済みモデルを作れる)ソフトウェアです。
NNCはもともとWindowsのアプリとして無償提供されていましたが、2017年にそのクラウド版であるNNC Cloudがリリースされました。
これを利用すれば、高性能なPCを用意する必要がなくブラウザさえあれば学習環境を利用できます。
クレジットカードを登録すれば有料でGPUも使えますが、CPUのみであればクレジットカードを登録しなくても10時間まで無料で利用できます。今回はデータ数も少ないのでCPUでも十分です。

NNCで学習済みモデルを作成するには、学習用の入力データを作ってアップロードする → ニューラルネットワークモデルを作成する → 学習する、という作業が必要です。

まずは学習用データの作成とアップロードを行いましょう。
NNCでの学習データは、学習に使うデータと学習の検証に使うデータの2種類が必要です。通常は収集したデータ(今回は40個の音声データ)の8割程度を学習用データとし2割程度を検証用データとします。
また、画像以外の数値データ(音声波形データやセンセーデータ)を学習データとする場合は、下図のように、数値データのみのCSVファイル(作業5で作られた clap_0.csv など)の他に、そのファイルのラベル(＝答え。拍手か声か)をまとめたデータセットCSVファイルが必要です。



引用元

では、今回は作業5で録音した20個ずつの音声のうち15個を学習用、5個を検証用に利用することとし、データセットファイルを作ってアップロードしましょう。

★作業6★ 学習用・検証用のデータセットCSVファイルの作成とアップロード

●以下のようなファイルを作成し、作業5で作られたCSVファイルと同じ場所に「sounds_train.csv」というファイル名で保存して下さい

```
x:sound,y:label
./clap_0.csv,0
./clap_1.csv,0
./clap_2.csv,0
./clap_4.csv,0
./clap_5.csv,0
./clap_6.csv,0
```

```
./clap_8.csv,0
./clap_9.csv,0
./clap_10.csv,0
./clap_12.csv,0
./clap_13.csv,0
./clap_14.csv,0
./clap_16.csv,0
./clap_17.csv,0
./clap_18.csv,0
./voice_0.csv,1
./voice_1.csv,1
./voice_2.csv,1
./voice_4.csv,1
./voice_5.csv,1
./voice_6.csv,1
./voice_8.csv,1
./voice_9.csv,1
./voice_10.csv,1
./voice_12.csv,1
./voice_13.csv,1
./voice_14.csv,1
./voice_16.csv,1
./voice_17.csv,1
./voice_18.csv,1
```

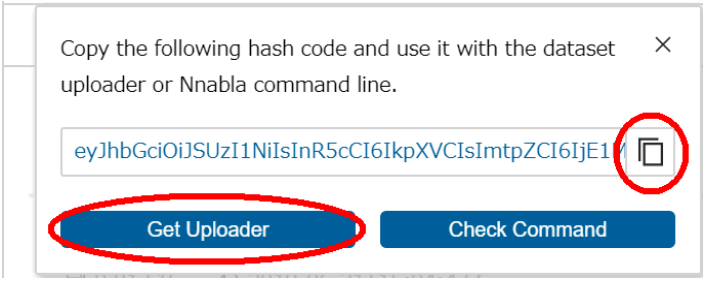
※x(1列目)が入力データ(のCSVファイル名)、y(2列目)が答えのラベル(拍手か声か)です。ラベルは文字列ではなく数値として入れます。今回は 0:拍手、1:声 と決めて入れました

●同様に以下のようなファイルを作成し、「sounds_validate.csv」というファイル名で保存して下さい

```
x:sound,y:label
./clap_3.csv,0
./clap_7.csv,0
./clap_11.csv,0
./clap_15.csv,0
./clap_19.csv,0
./voice_3.csv,1
./voice_7.csv,1
./voice_11.csv,1
./voice_15.csv,1
./voice_19.csv,1
```

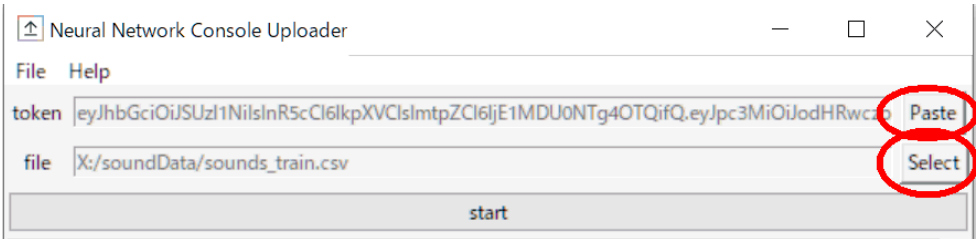
●NNC(http://dl.sony.com/console/)にログインし、「Dataset」→「Upload Dataset」を選びます

●表示されたトークン(hash code)をクリップボードにコピーしてから、[Get Uploader]ボタンをクリックして下さい

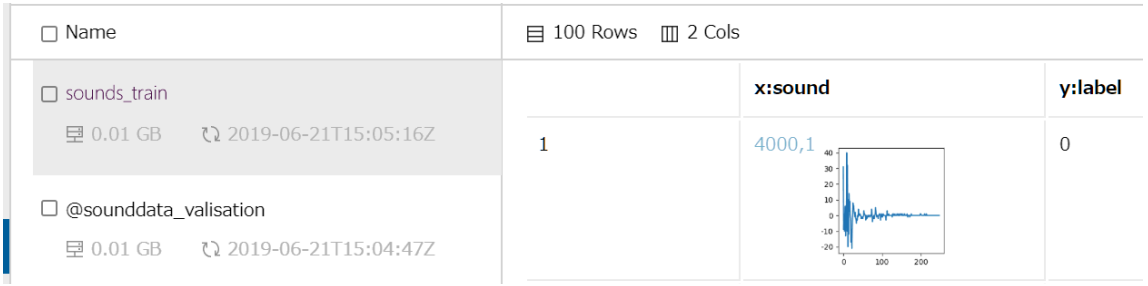


●表示されたページから、ご利用のOSに合わせたアップロードツールをダウンロードし、任意の場所に解凍(展開)し、中に入っている gui というツールを実行して下さい

●Pasteボタンでトークンを貼り付け、[Select]ボタンを押して先ほどの sounds_train.csv を選びます



●[Start]ボタンをクリックし、アップロードが完了するまで待ちます。その後、NNCの「Dataset」ページの表示が「Finish」になるのを待ちます



※自動的に clap_0.csv などの波形CSVもアップロードされるため波形データが表示されています。

●同様にして sounds_validate.csv をアップロードして下さい

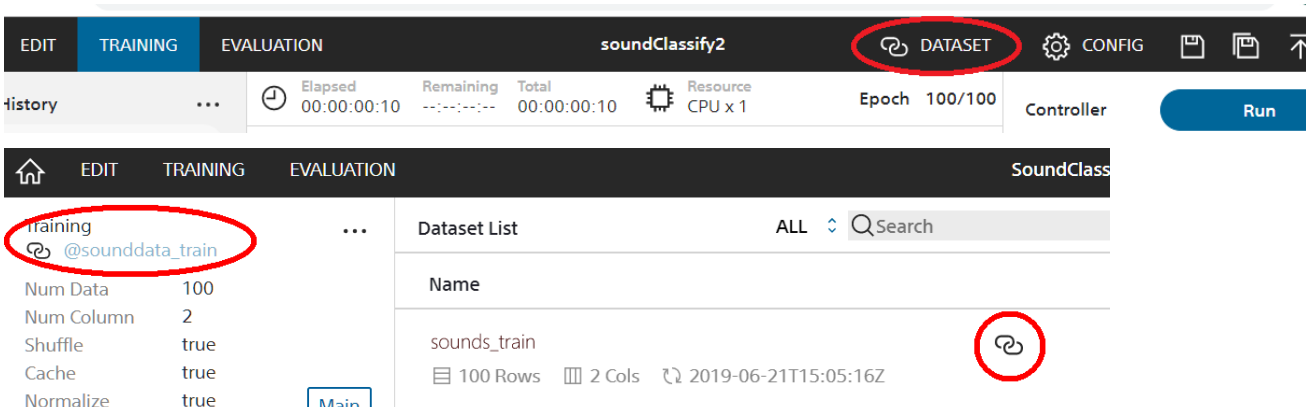
※アップロードにどうしても失敗する場合は、すべてのファイルを圧縮したファイルを「トークン名.zip」として保存し、こちらにアップロードしてください。講師側で代理アップロードします。

次に、ニューラルネットワークモデルを作成します。
NNCを使ったモデルの詳しい作り方についてはこちらのチュートリアルなどを参照して下さい。
基本的には「Convolution層＋Pooling層＋Activation関数」の組み合わせと「Affine層＋Activation関数」の組み合わせを何層にも重ね合わせ、最後にLoss関数を配置するような構成が一般的です。

★作業7★ ニューラルネットワークモデルの作成と学習

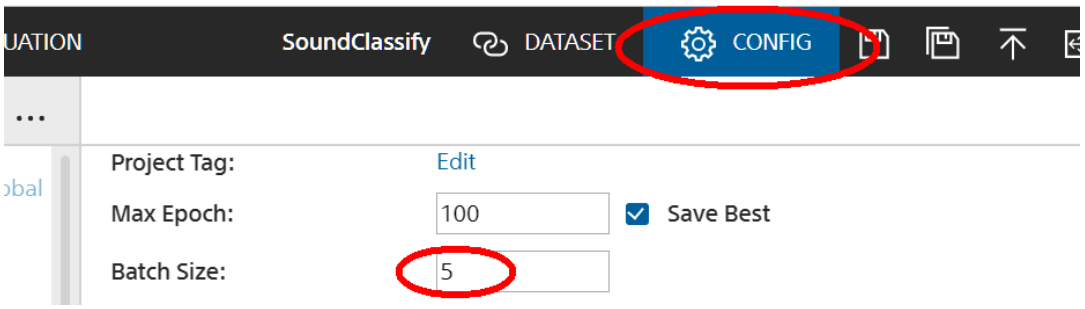
●NNCで、「Project」→ [+New Project] を選び、プロジェクト名に「SoundClassify」と入力して [OK]をクリックします

●右上の[DATASET]をクリックし、左側の一覧から「Training」を選び、右側の「Not Set」をクリックしてから、先ほど作った sounds_train の「🔗」をクリックして割り当てます

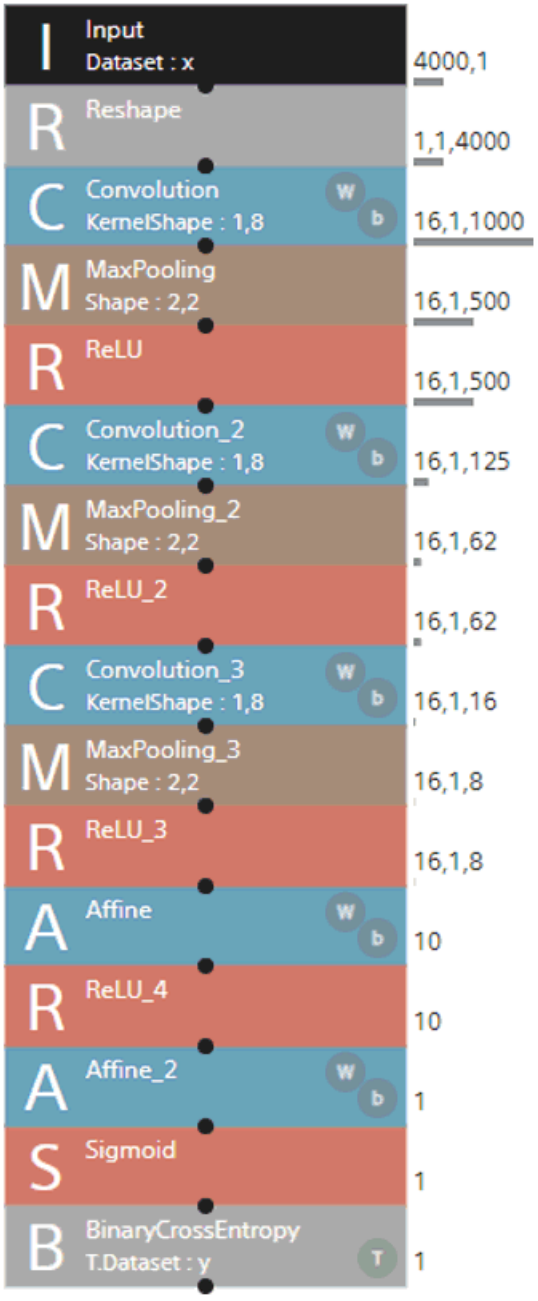


●同様にして左側の一覧の「Validation」に sounds_validate を割り当てます

●[CONFIG]をクリックして、Batch Sizeを「5」に変更します



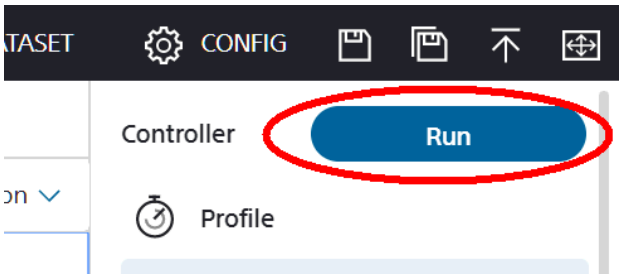
●[EDIT]に戻り、以下のようなニューラルネットワークモデルを作成します
※左上の「Components」から該当するパーツ(例えば最初は IO の中の Input)を選んでドラッグ &ドロップで配置した後、左下の「Layer Property」で下図右側の指定通り設定を変更して下さい。



IO → Input Size : 4000,1
Others → Reshape OutShape : 1,1,4000
Basic → Convolution OutMaps : 16　KernelShape : 1,8　Strides : 1,4
Pooling → MaxPooling KernelShape : 2,2
Activation → ReLU
Basic → Convolution OutMaps : 16　KernelShape : 1,8　Strides : 1,4
Pooling → MaxPooling KernelShape : 2,2
Activation → ReLU
Basic → Convolution OutMaps : 16　KernelShape : 1,8　Strides : 1,4
Pooling → MaxPooling KernelShape : 2,2
Activation → ReLU
Basic → Affine OutShape : 10
Activation → ReLU
Basic → Affine OutShape : 1
Activation → Sigmoid
Loss → BinaryCrossEntropy

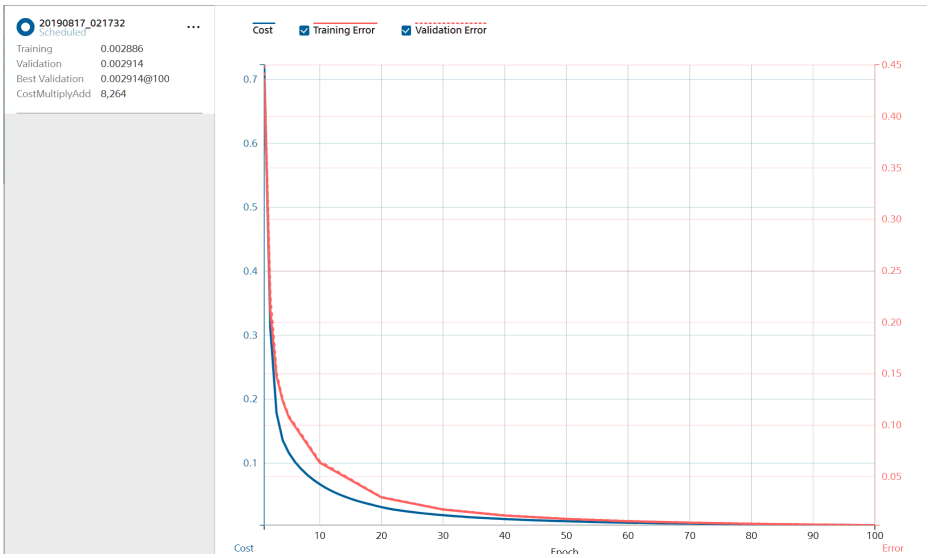
※Convolution・MaxPooling・ReLUのセットはコピー＆ペーストすると楽です
※今回のように2分類(拍手と声)の場合はLoss関数にBinaryCrossEmtropy を使いますが、3分類以上の場合は CategoricalCrossEmtropy を使って下さい

●右上の [Run] ボタンをクリックして学習を開始します



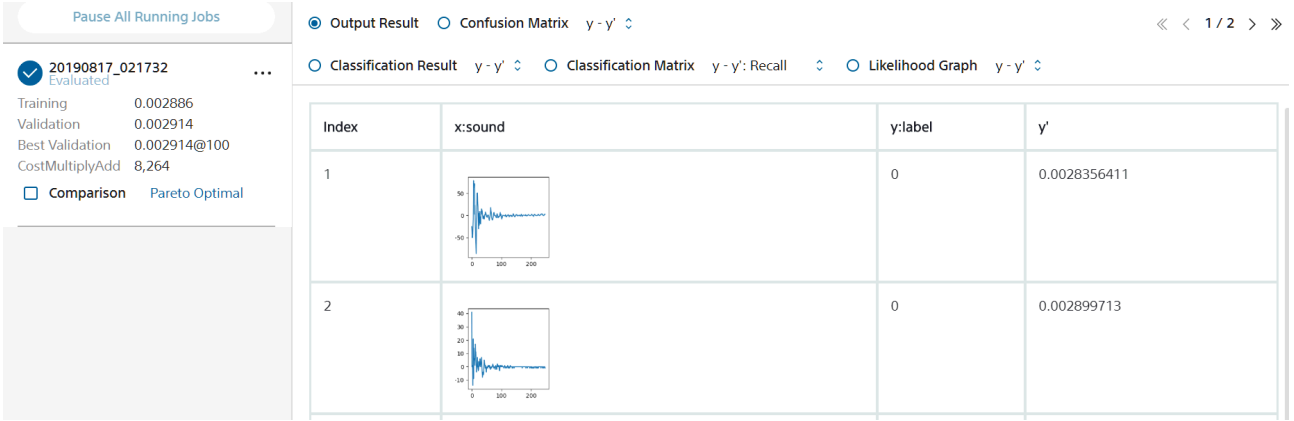
※同時に利用している人が多いと学習が開始されるまで時間がかかります。焦らず待ちましょう

●学習が始まれば1分もかからず完了し、以下のようなグラフが表示されます



※Trainingのデータ(15個ずつ)で学習が行われ、Validationのデータ(5個ずつ)で学習したモデルで検証(推論してみる)が行われています。これを100回(100epoch)繰り返した時の検証失敗率(推論の不正解率)の変化がValidation Errorとして表示されています。これが下がる方向に収束していれば基本的には学習は上手くいっていることになります。

●さらに[TRANING]ページ内 [Run] ボタンをクリックして評価すると、以下のように認識結果が表示されるはずです



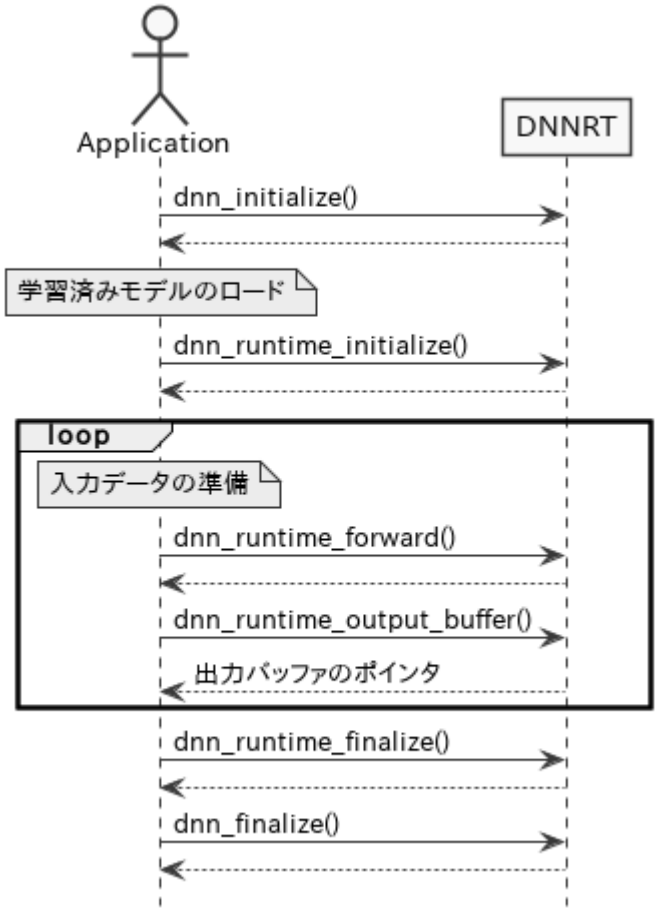
※yが正解ラベル、y'が推論の結果です。上図では yが0(＝拍手)のデータに対して、推論結果が0.0028と0に近い値になっているので、これは正しく推論できている、とう事になります。

これで学習は完了し認識精度の高い学習済みモデルが完成しました。
ただし、今回のような40個程度の音声データでは数が圧倒的に足りないので、本格的に使う場合は様々な人の拍手や声を集め、数百～数千のデータを用意して学習させて下さい。

5. 学習済みモデルを利用したリアルタイム音声認識

では、いよいよ学習済みモデルを利用してSPRESENSEでリアルタイムに音声認識するプログラムを作成します。

SPRESENSE SDKには、NNCで学習した学習済みモデルを利用して推論を行うために DNNRTというライブラリが用意されており、以下のような簡単な流れで推論を実装できます。



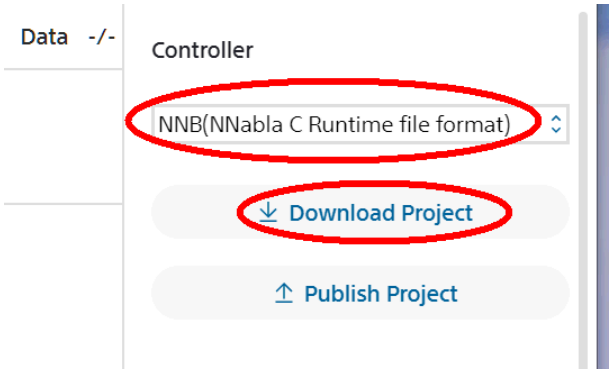
<< 引用元 >>

音声認識プログラムの大半は作業3で作成した録音プログラムと同じです。なぜなら録音処理部分については全く同じであり、異なるのは主に0.5秒分のデータを切り出した後の推論処理だけだからです。

★作業8★ 学習済みモデルのコピーと推論プログラムの作成

●NNCで、作業7で作成したプロジェクトを開き、「EVALUATION」をクリックして、右側のControllerから「NNB(NNabla C Runtime file format)」を選択して[Donwload Project]をクリックして下さい。

※これで、SPRESENSEで利用できる形式の学習済みモデルがダウンロードされます



●ダウンロードされた学習済みモデルのファイル名を「**soundClassify.nnb**」というファイル名に変更してmicroSDカードに保存し、SPRESENSEに差し込んで下さい。

※もし学習が終わっていない場合はこちらで用意した学習済みモデルを使って下さい→こちら

●Arduino IDEで以下のプログラムを入力して下さい(ファイル名は MicClassify など)
※参考までに、背景色を変えている部分以外は録音プログラムと同じ処理です

```
#include <SDHCI.h>
#include <MediaRecorder.h>
#include <MemoryUtil.h>
#include <DNNRT.h>

//定数パラメタ定義
const float  RECTIME = 0.5f;
const int    REC_THRESHOLD = 250;    //録音を開始する音量の閾値

//LED接続ピン設定
#define PIN1 7     //青LED接続ピン
#define PIN2 6     //黄LED接続ピン

//録音機能を実現するクラス
MediaRecorder *theRecorder;

//SDカードアクセス用のクラス
SDClass theSD;

//録音データー時格納用バッファ
const int32_t buffer_size = 8192;
uint8_t       s_buffer[buffer_size];

//WAVデータ格納用バッファ(48KHz・モノラルで RECTIME秒録音するために必要なサンプル数)
const int recSize = (int)(48000 * RECTIME);
short wavData[recSize];              //16bitで録音するので各サンプルはshort型
int recIdx = 0;

//ディープラーニング推論用クラス
DNNRT dnnrt;
DNNVariable input(4000);       //入力データ用(データは4000個の実数値)

//MediaRecorderのイベント(録音開始とか終了とか)処理用コールバック関数
static bool mediarecorder_done_callback(AsRecorderEvent event, uint32_t
 result, uint32_t sub_result)
{
  //今回は特にやりたい事がないので、引数の値を適当に出力するだけ
  printf("## MediaRecorder Event : %x %x\n", event, result);
  return true;
}

//初期化処理
void setup()
{
  //LEDを初期化(消灯)
  pinMode(PIN1, OUTPUT);
  pinMode(PIN2, OUTPUT);
  digitalWrite(PIN1, LOW);
  digitalWrite(PIN2, LOW);
  theSD.begin();   //SDカードアクセスの準備
```

```
  //録音機能に必要なメモリを確保する
  initMemoryPools();
  createStaticPools(MEM_LAYOUT_RECORDER);

  //録音機能(MediaRecorder)を初期化
  theRecorder = MediaRecorder::getInstance();
  theRecorder->begin();

  //録音用のクロックを標準に設定
  theRecorder->setCapturingClkMode(MEDIARECORDER_CAPCLK_NORMAL);

  //アナログマイクからの入力に設定・イベントコールバック関数を設定
  theRecorder->activate(AS_SETRECDR_STS_INPUTDEVICE_MIC,
 mediarecorder_done_callback);
  theRecorder->setMicGain(210);    //マイクの音量調整(0～210)

  //マイクの準備完了を少し待つ
  usleep(100000);

  //録音設定(WAV形式・モノラル・48KHz・16bit)
  theRecorder->init(AS_CODECTYPE_WAV, AS_CHANNEL_MONO,
 AS_SAMPLINGRATE_48000, AS_BITLENGTH_16, 0);

  //学習済みモデル soundClassify.nnb を読み込む
  puts("Loading network model");
  File nnbfile("soundClassify.nnb");
  if (!nnbfile) {
    puts("nnb not found");
    exit(1);
  }

  //読み込んだ学習済みモデルでディープラーニング推論用クラスdnnrtを準備
  puts("Initialize DNNRT");
  int ret = dnnrt.begin(nnbfile);
  if (ret < 0)
  {
    puts("DNNRT initialize error.");
    exit(1);
  }

  //録音開始！
  theRecorder->start();
  puts("Recording Start!");
}


//メインループ
void loop()
{
  //録音データをs_bufferに取り出す
  uint32_t size = 0;
```

```
  err_t err = theRecorder->readFrames(s_buffer, buffer_size, &size);

  //エラーだった場合(OK・バッファ不足以外の場合)は終了処理(done)に飛ばす
  if(err != MEDIARECORDER_ECODE_OK && err !=
 MEDIARECORDER_ECODE_INSUFFICIENT_BUFFER_AREA)
  {
    puts("Recording Error!");
    done();
  }
  //録音データが取得できた場合、録音データ処理(signal_process)に飛ばす
  else if(size > 0)
  {
    signal_process(size);
  }
}

//録音データ処理
void signal_process(uint32_t size)
{
  //録音データを16bit符号付き整数とした扱えるようにキャスト
  short *p = (short*)s_buffer;

  //１サンプルごとのループ(１サンプル16bitなのでループ回数はsize/2)
  for(int i = 0; i < size/2; i++)
  {
    //録音開始前(WAVデータとして取り出し始める前)かつ振幅がREC_THRESHOLDを超えたら
    if(recIdx == 0 && abs(p[i]) > REC_THRESHOLD)
    {
      //録音開始(WAVデータとして取り出し始める)
      puts("Record to File Start");
      wavData[recIdx] = p[i];
      recIdx++;
      //LEDはどちらも消灯
      digitalWrite(PIN1, LOW);
      digitalWrite(PIN2, LOW);
    }
    //録音中かつ録音時間がRECTIME未満の場合
    else if(recIdx != 0 && recIdx < recSize)
    {
      //サンプル値を取り出してwavDataに格納
      wavData[recIdx] = p[i];
      recIdx++;
    }
    //録音時間がRECTIMEになった場合
    else if(recIdx == recSize)
    {
      //推論用の入力データ領域を準備
      float *input_buffer = input.data();

      //データの１／６(＝8KHzのデータとして)をfloat型の値にして入力データ領域にコ
 ピー
      for(int i = 0; i < recSize; i+=6){
```

```
          input_buffer[i/6] = (float)wavData[i]/4096.0;
        }

        //dnnrtに入力データをセットして推論を実行
        dnnrt.inputVariable(input, 0);
        dnnrt.forward();

        //推論の結果を取り出す
        DNNVariable output = dnnrt.outputVariable(0);
        printf("%0.2f\n", output[0]);

        //推論の結果が0に近い場合は拍手(clap)と判断
        if(output[0] < 0.2)
        {
          printf("clap!\n");
          //青色LEDを点灯
          digitalWrite(PIN1, HIGH);
        }
        //推論の結果が1に近い場合は人の声(voice)と判断
        else if(output[0] > 0.8){
          printf("Voice!\n");
          //黄色LEDを点灯
          digitalWrite(PIN2, HIGH);
        }

        recIdx = 0;
      }
    }
  }

  //全録音終了処理
  void done()
  {
    //録音停止と録音オブジェクトの後片付け
    theRecorder->stop();
    theRecorder->deactivate();
    theRecorder->end();

    //LED消灯
    digitalWrite(PIN1, LOW);
    digitalWrite(PIN2, LOW);

    puts("End Recording");
    exit(1);
  }
```

●[マイコンボードに書き込む]ボタンをクリックし、プログラムの書き込みと実行を確認します
※書き込み完了後に「ツール」→「シリアルモニタ」を開くとデバグ情報が表示されて便利です

●拍手の音を鳴らすと青色LEDが点灯し、「わっ！」と声をかけると黄色LEDが点灯することを確認して下さい

6. Azure IoT Centralによる可視化の準備

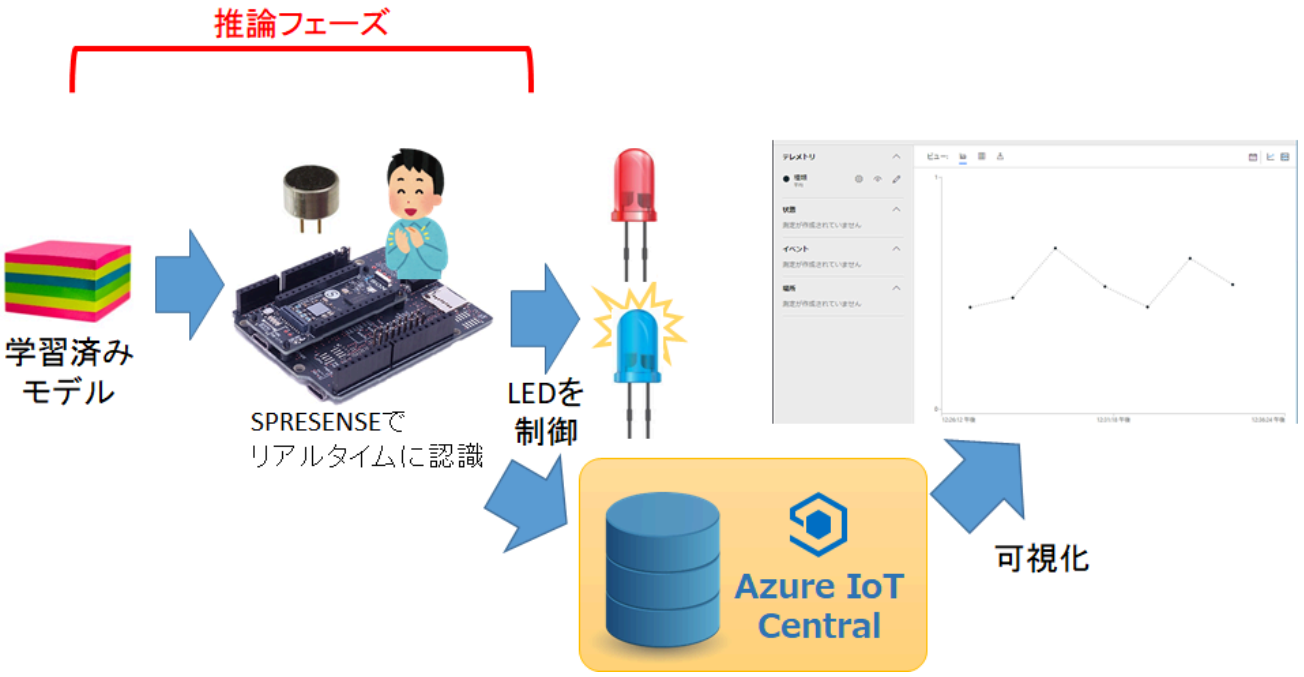
以上でエッジ側(マイコン側)でリアルタイムに音声分析を行えるようになりましたが、このままだと認識結果を後から集計したり分析したりといった事ができません(例えば特定の時間帯には人の声が多く聞こえる、といった知見は得られません)。

そこで、認識結果を蓄積する仕組みを作ってみましょう。この蓄積をSPRESENSE側(エッジ側)で行うことはもちろんできますが、例えば将来的に複数のSPRESENSEを設置し、全てのデータをまとめて分析する事などを考えると、クラウド上に蓄積すべきです。また、将来的な規模の予測ができない場合も多いため、スケールするサービスを使うべきです。

この条件にマッチするサービスとして、今回は2018/12にリリースされたAzureのフルマネージドIoTサービスである IoT Central (https://azure.microsoft.com/ja-jp/services/iot-central/)を利用します。

IoT Centralを利用すると、IoT機器からの情報収集、蓄積、可視化、アラートなどの機能を簡単に実現できます。

IoT Centralは高機能ですが、今回は音声を認識する度にその結果をIoT Centralに送って時系列で可視化(グラフ化)する、という基本的な機能のみ使ってみます。



この流れを習得して応用すれば、工場内で異常音が発生しやすい時間帯などを可視化できるはずです。

IoT Centralでは、デバイスから送られる値(認識結果である0や1)のことをテレメトリと呼び、テレメトリをまとめたセットをデバイステンプレートと呼びます。各デバイスはデバイステンプレートに紐付けて登録します。

このため、事前準備として、まずデバイステンプレートを作成し、その後デバイスを必要な数だけ登録する、という作業が必要です。

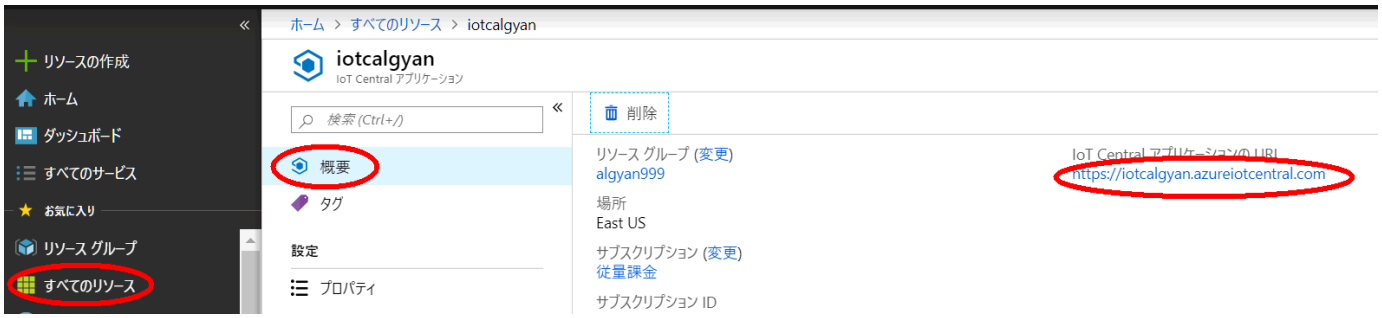
★作業9★ **IoT Centralの準備**

●Azure Portalから ＋リソースの作成 →「モノのインターネット(IoT)」→「IoT Centralアプリケーション」を選びます

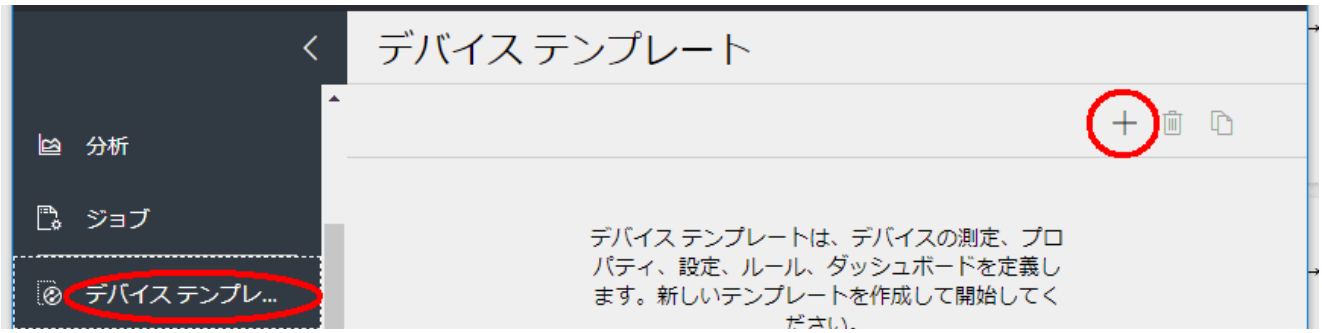
●以下の通り設定し、[作成] ボタンをクリックします(※これ以外の項目は任意で構いません)

■リソース名：iotc[自分の名前等]　　←例えば iotc**sato**
■リソースグループ：任意に選択 or 新規作成
■テンプレート　　：カスタムアプリケーション
■場所　　：米国東部

●作成が完了するのを待って、Azure Portalから → すべてのリソース → iotc～～ を選び、「IoT Centralアプリケーション」のURLをクリックして開きます



●IoT Centralのページで、左側の「デバイステンプレート」をクリック → [＋] ボタンをクリックします



●「カスタム」を選び、「デバイステンプレートの名前」に「spresense-sound」などと入力して [作成] をクリックします

●作成されたテンプレートを選び、「＋新規」→「テレメトリ」を選んで、以下の通り設定し、[保存] ボタンをクリックします

■Display Name　：種類 ← 可視化の際にグラフに表示する項目名
■フィールド名　：type　　← デバイスから送るデータにつける名前(JSONのキー)
■最小値　：0
■最大値　：1
■小数点以下桁数　：0

●「デバイス」→「spresense-sound」を選び →[＋] ボタンをクリック →「実際」を選択し、以下の通り設定し、[作成] ボタンをクリックします

■デバイス名　　：spr-**[自分の名前等]**　　←例えば spr-**sato**

※現場で実際に使う場合は、利用する(＝IoT Centralにデータ送信する)デバイスの数分だけこれを繰り返します

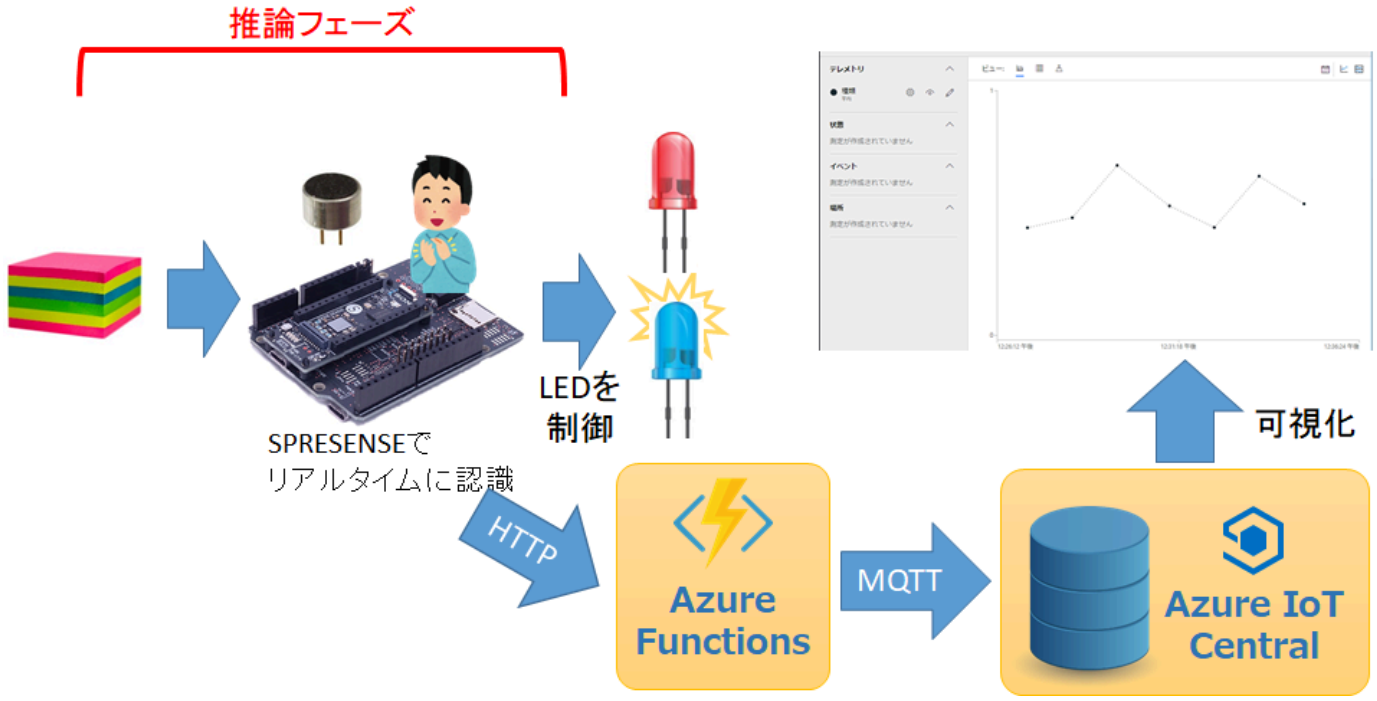
●作成された spr-～ を選択して右上の [接続] をクリックし、表示された「スコープID」「デバイスID」「主キー」をコピーしてメモ帳等に貼り付けておきます(後で使います)

以上で IoT Central側の準備は完了しました。なお、デバイステンプレートを作ると自動的にデバイスのシミュレータが1つ追加されるので、実機(SPRESENSE)からデータを送らなくても可視化の見た目などを確認できます。

もし確認したい場合は IoT Centralで「デバイスエクスプローラー」→「spresense-sound」→「spresense-sound-1(シミュレート済み)」を選ぶと確認できます(しばらく待つとランダムなデータによるグラフが表示されるはずです)。シミュレータが不必要なら削除しても構いません。

これで準備は整いましたので、後はSPRESENSEからIoT Centralに認識結果を送るだけで自動的にグラフ化(可視化)をしてくれます。

しかし、SPRESENSEからIoT Centralに直接値を送ることは、現時点でのWiFiモジュールライブラリの制約(MQTTプロトコルによるデータ送信時のサイズ制限)により不可能なため、今回は Azure Functions を間に入れることで実現します。



Azure Functionsは、Azureでサーバレスアーキテクチャを実現するサービスで、その名の通りAzure側に関数を定義しておき、これをHTTP経由などで実行させる仕組みです。
サーバを借りる訳ではないので、あくまでも実行した回数によって課金されるのが特徴です。よって全く呼び出さなければ費用は1円もかかりません。
Azure Functionsから IoT Centralを利用することは当然できます(Azureのサービス同士ですし)し、WiFiモジュールからHTTPアクセスを行う分には制限がないので、この形であれば問題なく実現できます。

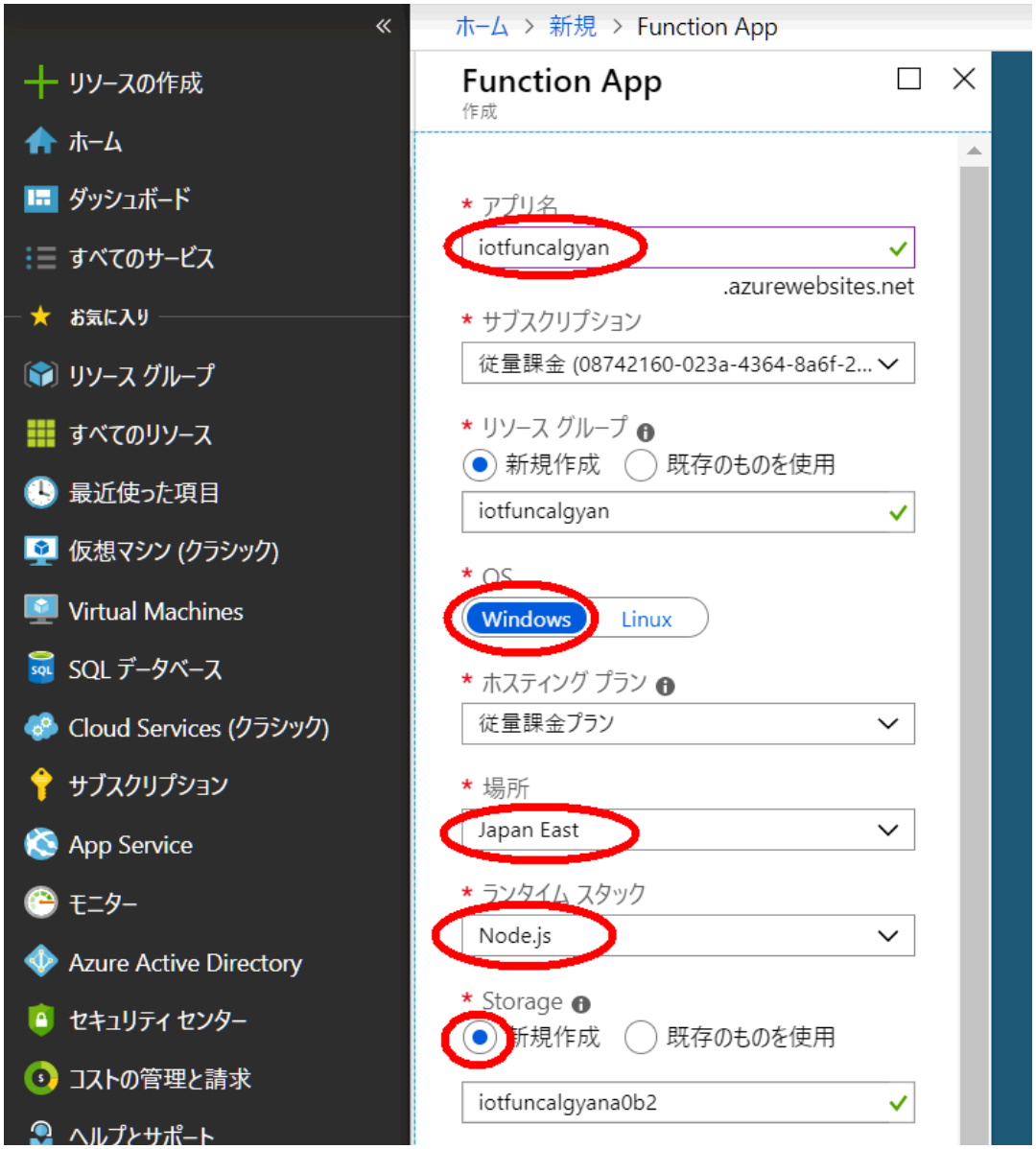
では、IoT Centralにデータを送る Functionsを作り、動作を確認してみましょう。

★作業10★ Azure Functions

●Azure Portalから ＋リソースの作成 →「Compute」→「Function App」を選びます

●以下の通り設定し、[作成] ボタンをクリックします(※これ以外の項目は任意で構いません)

■アプリ名　　：iotfunc[自分の名前等]
■リソースグループ　：任意に選択 or 新規作成
■OS　　：Windows
■場所　　：Japan East
■ランタイムスタック：Node.js
■Storage　　：新規作成



●作成が完了するのを待って(約2分)、すべてのリソース → iotfunc～～ を選びます

●[プラットフォーム機能] → [App Service Editor]をクリックして開きます

●[Ctrl] + [Shift] + [C]キーを押してコンソールを開き、以下のコマンドを入力します

```
\> npm i -g dps-keygen
\> dps-keygen -si: 作業9でコピーしたスコープID -di: 作業9でコピーしたデバイスID
 -dk: 作業9でコピーした主キー
```

※出力されたConnection String:の内容(HostName=～)をメモ帳等にコピーしておきます

※**HostName=の部分から**SharedAccessKey=～の部分まで全てをコピーして下さい(長いです)

```
\> npm i --save azure-iothub azure-event-hubs azure-iot-device-mqtt
 azure-iot-device
```

※これにより、Node.js用のIoT Central(IoT HUB)用のライブラリがインストールされ、後ほど作る関数から利用できるようになります

●iotfunc～～のページに戻り、[関数]をクリック →「＋新しい関数」をクリックします

●「HTTP trigger」を選び、関数名に「sendType」、承認レベルに「Anonymous」を選んで [作成]ボタンをクリックします

※承認レベルに「Anonymous」を選ぶと誰でも関数を呼び出せてしまうため、セキュリティ上は望ましくありませんが、今回は処理を簡略化するためにAnonymousにしています

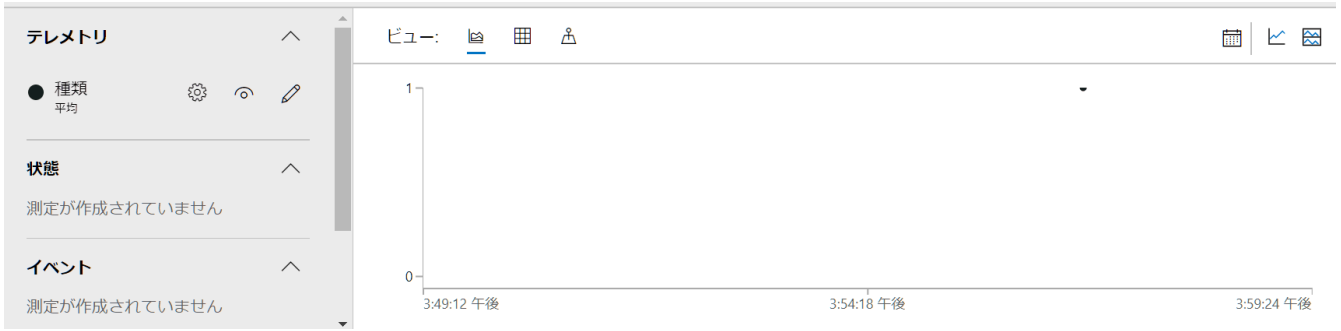
●index.jsの内容として以下のプログラムを入力します

```
//IoT Centralへの接続文字列
const connectionString = " さっき作ったConnection String(HostName=～) ";

//IoT Centralへの通信に必要なライブラリ読み込み・オブジェクト作成
var devicemqtt = require('azure-iot-device-mqtt');
var device = require('azure-iot-device');
var client = devicemqtt.clientFromConnectionString(connectionString);
var Message = device.Message;

//HTTP経由で呼び出された時に実行される処理本体
module.exports = async function (context, req) {
    //リクエストパラメタにlabelがあれば
    if (req.query.label) {
        //IoT Centralに送信するデータを作り出す(JSONを作ってMessageでカプセル化)
        const json = createJson(req.query.label);
        var message = new Message(json);

        //IoT Centralに送信し、呼び出し元には送信成功を返す
        client.sendEvent(message, printResultFor('send'));
        context.res = {
            body: "Send " + req.query.label
        };
    }
    //リクエストパラメタにlabelがない場合は呼び出し元にエラーを返す
    else {
        context.res = {
            status: 400,
            body: "Please pass a data on the query string"
        };
    }
};

//IoT Centralに送信するデータをJSON形式で作り出す関数
function createJson(label)
{
    //現在時刻を取得
    const current_time = new Date().toISOString();
    //現在時刻とlabelを合わせたオブジェクトを作成
    const ddx = { timestamp: current_time, type: label };
    //JSON文字列化して返す
    return JSON.stringify(ddx);
}

//IoT Centralへの送信結果を受け取るコールバック関数
function printResultFor(op) {
    return function printResult(err, res) {
        if (err) console.log(op + ' error: ' + err.toString());
        if (res) console.log(op + ' status: ' + res.constructor.name);
    };
}
```

●右端の「テスト」をクリックし、HTTPメソッドに「GET」、クエリに「label」「1」と入力して、[実行]ボタンをクリックします

※プログラムと接続文字列に間違いがなければ、「出力」に「Send 1」と表示されるはずです

●IoT Centralのページから 「デバイス」→「spresense-sound」→「spr-～～」を選び、しばらく(1分くらい)すると、今送ったダミーの認識結果がグラフ化されて表示される事を確認して下さい



●Azureポータル → すべてのリソース → iotfunc～～ → [関数] → sendType → [</>関数のURLの取得] をクリックして、表示されたURLをコピーします

●ブラウザにURLを貼り付け、URLの後ろに「?label=0」などとパラメタを付与してアクセスし、「Send 0」などと表示される事、およびIoT Centralのグラフにデータが追加されることを確認します

これで「http://～～.azurewebsites.net/api/sendType?label=認識結果」にHTTPアクセスすれば、IoT Centralに送られてグラフ化される仕組みが作られました。あとはこれを SPRESENSEから呼び出すだけです。

7. Azure IoT Centralへの通信処理(サブコア)の実装

さて、いよいよ SPRESENSEからFunctionsを呼び出す(＝HTTP通信する)処理を追加します。呼び出すタイミングは当然ディープラーニングによる推論結果(認識結果)が出たタイミングです。

ここで問題になるのが、この通信処理にはある程度の時間がかかる(通常1秒未満ですが)という事です。このため、録音後の処理にそのまま記述すると、録音バッファからデータを取り出す処理も一時的に止まってしまうため、バッファが溢れてしまいます。

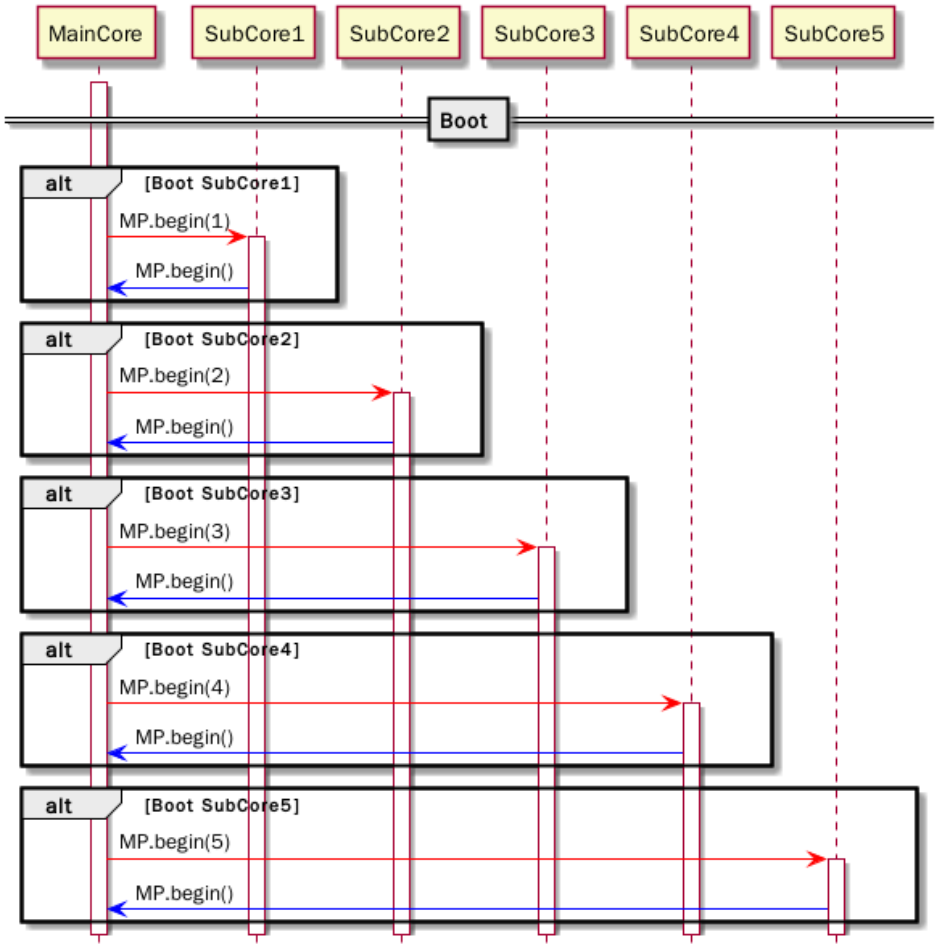
作業3で作成した録音プログラムのファイル出力処理と同じように、録音を停止→通信処理→録音を再開、という仕組みにしてもよいですが、実用で考えた場合、録音停止中に重要な音が発生した場合に聞き漏らして(録り逃して)しまう可能性があります。

そこで、SPRESENSEの特徴の一つであるマルチコアCPU(＝複数の処理を並行動作させられる)を活用しましょう。

すなわち、通信処理を全てサブコアに実装し、メインコアはサブコアに通信指示を送る処理だけにして並列動作させます。こうすればメインコアは録音と推論だけに集中できるため、音を聞き漏らす心配はありません。

Arduino IDEでサブコア側のプログラムを作るためには、サブコア用に新規のプログラムを作成した後、「ツール」→「Core」で SubCore1 ～ SubCore5 を選択して書き込みます。

サブコア用に書き込んだプログラムはそのままでは実行されません。メインコア(側に書き込んだプログラム)で MultiCore MP ライブラリを利用し、「MP.begin(コア番号)」という処理を実行すると、指定されたサブコア(に書き込んだプログラム)の実行が開始されます。



<< 引用元 >>

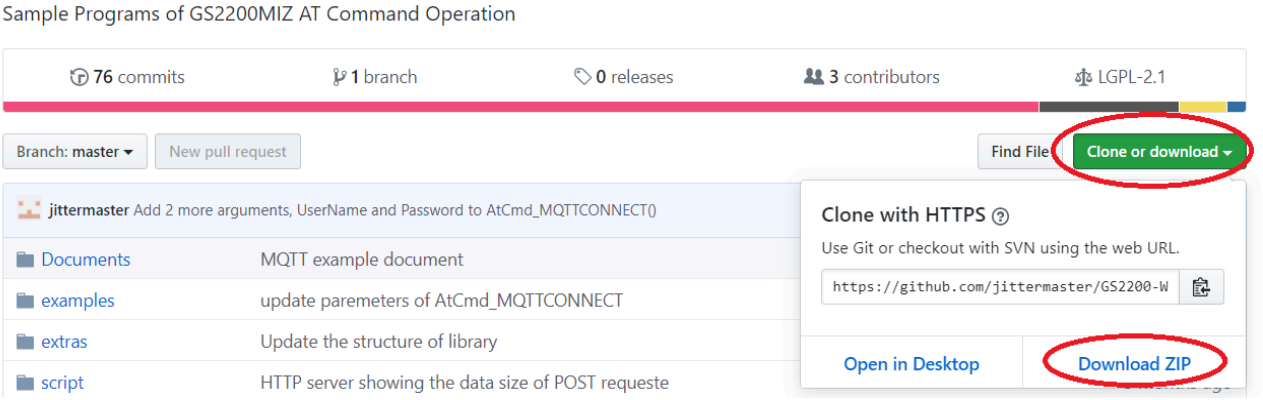
また、コア同士の通信には、MP.Send() と MP.Recv() 関数を利用します。これらの使い方は公式ドキュメントを見て下さい。今回はメインコアから認識結果を MP.Send関数で送信し、サブコア側でその認識結果を MP.Recv 関数で受信します。

★作業11★ 通信処理の実装

●SPRESENSEからUSBケーブルを抜いて電源を切り、WiFiモジュール(iS110B)を以下の通り装着します

※差し込む位置と方向に注意して下さい
※スペーサーを付けていない場合は、拡張ボードからメインボードを外してから装着してください。

●WiFiモジュールのArduinoライブラリがあるGitHub (https://github.com/jittermaster/GS2200-WiFi)にアクセスし、[Clone or download] → [Download ZIP] をクリックしてダウンロードします



●Arduino IDEで「スケッチ」→「ライブラリをインクルード」→「ZIP形式のライブラリをインクルード」を選び、先ほどダウンロードしたファイル(GS2200-WiFi-master.zip)を選択してインストールします

●Arduino IDEで「ファイル」→「新規ファイル」を選びそのまま保存します。名前は「MicClassify_sub」などとします

●MicClassify_subに以下のプログラムを入力して下さい

```
#include <MP.h>
#include <GS2200Hal.h>
#include <GS2200AtCmd.h>
#include <TelitWiFi.h>

//通信関連パラメタ定義
#define  AP_SSID        " WiFiのSSID "
#define  PASSPHRASE     " WiFiのパスワード "
#define  HTTP_SRVR_IP   " 作業10で作ったFunctionsのドメイン名 (もしうまく作れてない
 場合は iotfuncalgyan.azurewebsites.net として下さい) "
#define  HTTP_PORT      "80"
```

```
//LED接続ピン設定
#define PIN1 7    //青LED
#define PIN2 6    //黄LED

//WiFi接続関連クラス等
TelitWiFi gs2200;
TWIFI_Params gsparams;
char server_cid = 0;
extern uint8_t ESCBuffer[];
extern uint32_t ESCBufferCnt;

//HTTPレスポンスをチェックする処理
void parse_httpresponse(char *message)
{
  char *p;
  if( (p=strstr( message, "200 OK\r\n" )) != NULL ){
    printf( "Response : %s\n", p+8 );
  }
}

//初期化処理
void setup() {
  //WiFiモジュールの初期化
  Init_GS2200_SPI();
  gsparams.mode = ATCMD_MODE_STATION;
  gsparams.psave = ATCMD_PSAVE_DEFAULT;
  if( gs2200.begin( gsparams ) ){
    puts( "GS2200 Initilization Fails" );
    while(1);
  }

  //WiFiに接続
  if( gs2200.connect( AP_SSID, PASSPHRASE ) ){
    printf( "Association Fails" );
    while(1);
  }

  //HTTP通信先などを設定
  AtCmd_HTTPCONF( HTTP_HEADER_HOST, HTTP_SRVR_IP );
  WiFi_InitESCBuffer();

  //サブコアの準備完了をメインコアに通知
  MP.RecvTimeout(MP_RECV_POLLING);
  MP.begin();
  puts("SubCore 1 start");
}

//メインループ
void loop() {
  int8_t msgid;
  void* msgdata;
  static int waitCnt = 0;
```

```
  //メインコアからのメッセージ(アドレス)を取り出す
  int ret = MP.Recv(&msgid, &msgdata);

  //メッセージが取得できた場合
  if(ret >= 0){
    //送信時に指定されたメッセージIDが1なら
    if(msgid == 1){
      //受信した認識結果を取り出す
      int label = *(int*)msgdata;
      //Azureに認識結果を送信する処理を呼び出す
      sendType(label);
      waitCnt = 0;
    }
  }
  //取得できなかった場合は10msだけ待って繰り返す
  else
  {
    usleep(10000);
    waitCnt++;
    //一定時間通信しなかった場合は適当な通信を行ってWiFiコネクションを維持
    if(waitCnt == 6000){
      char w[20];
      AtCmd_DNSLOOKUP(HTTP_SRVR_IP, w);
      waitCnt = 0;
    }
  }
}

//Azureに認識結果を送信する(=Functionsを呼び出す)処理をまとめた関数
void sendType(int label){
  ATCMD_RESP_E resp;
  uint32_t start;
  char url[256];

  puts("Open Socket");

  //Webサーバ(Azure Functions)に接続
  do {
    resp = AtCmd_HTTPOPEN( &server_cid, HTTP_SRVR_IP, HTTP_PORT );
  } while (ATCMD_RESP_OK != resp);

  //URLを組み立てる
  String buf = "/api/sendType?label=" + String(label);

  //URLをchar型配列変換してWebサーバ(Azure Functions)に送信
  buf.toCharArray(url, buf.length()+1);
  resp = AtCmd_HTTPSEND( server_cid, HTTP_METHOD_GET, 10, url, "", 0);

  //送信に成功した場合はレスポンスをチェック
  if( ATCMD_RESP_BULK_DATA_RX == resp ){
    if( Check_CID( server_cid ) ){
```

```
      parse_httpresponse( (char *)(ESCBuffer+1) );
    }
    WiFi_InitESCBuffer();
  }
  //送信に失敗した場合はエラー出力
  else{
    printf( "?? Unexpected HTTP Response ?? : %d\n", (int)resp );
    return;
  }

  //レスポンスが空になるまでループ
  start = millis();
  while(1){
    if( Get_GPIO37Status() ){
      resp = AtCmd_RecvResponse();
      if( ATCMD_RESP_OK == resp ){
        break;
      }
    }
    //５秒以上レスポンスがなかった場合はタイムアウト
    if( msDelta(start)>5000 ){
      puts("Socket Timeout");
      break;
    }
  }

  //Webサーバ(Azure Functions)から切断
  do {
    resp = AtCmd_HTTPCLOSE( server_cid );
  } while( ATCMD_RESP_OK != resp && ATCMD_RESP_INVALID_CID != resp );
  puts( "Socket Closed" );
}
```

●「ツール」→「Core」に「**SubCore1**」を選んでから、[マイコンボードに書き込む]ボタンをクリックし、プログラムをサブコアに書き込みます
※メインコアから処理開始指示(MP.begin)をしない限りサブコアのプログラムは実行されないため、この時点では何も起きません

●MicClassify(メインコア側のプログラム)に以下の通り処理を追加して下さい

◆#include群の最後に以下を追加

```
#include <MP.h>
```

◆setup関数の最初に以下を追加

```
//サブコア１(通信用)を起動
MP.begin(1);
```

◆プログラムの最初(#include群の後)に以下を追加(グローバル変数として宣言)

```
int label;  //認識結果を格納する変数
```

◆signal_process関数内の「推論の結果が0に近い場合は拍手(clap)と判断」のif文を以下のように書き換え(赤色の行を追加)

```
      //推論の結果が0に近い場合は拍手(clap)と判断
```

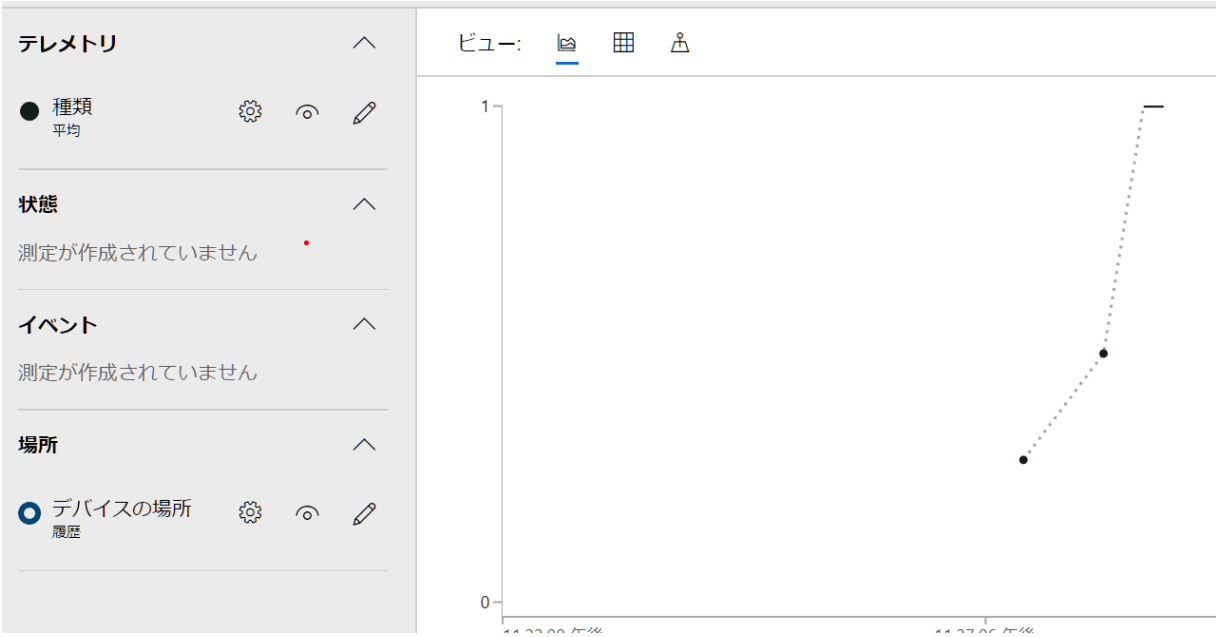
```
  if(output[0] < 0.2)
  {
    printf("clap!\n");
    //青色LEDを点灯
    digitalWrite(PIN1, HIGH);
    label = 0;
    MP.Send(1,(void*)&label,1);
  }
  //推論の結果が1に近い場合は人の声(voice)と判断
  else if(output[0] > 0.8){
    printf("Voice!\n");
    //黄色LEDを点灯
    digitalWrite(PIN2, HIGH);
    label = 1;
    MP.Send(1,(void*)&label,1);
  }
```

※MP.Sendの第１引数はメッセージID、第２引数は送信したいデータ(今回は音声認識結果)が入っているアドレス、第３引数は送信したいサブコア番号です。こうして送ったデータを、サブコア側では、MP.Recvで取り出します(先ほど作成しました)

●「ツール」→「Core」から「**MainCore**」を選んでから、[マイコンボードに書き込む]ボタンをクリックし、プログラムを書き込んで実行します

●何度か拍手の音や「わっ！」と声をかけて、LEDが点灯することを確認して下さい

●IoT Centralのページから 「デバイス」→「spresense-sound」→「spr-～～」を選び、しばらくすると先ほどの認識結果がグラフ化されて表示される事を確認して下さい



8. GPS処理の追加と可視化

SPRESENSEの特徴の一つとしてGPSが搭載されている点が挙げられます。GPSを活用すれば、例えば音声を認識した際の位置情報を取得できます。
また、IoT Centralも2019/7月のアップデートで位置情報の履歴(移動履歴)を可視化する機能が付きました。そこで今回はこれを用いて、音声認識した場所の履歴を地図上に可視化する機能を追加してみます。

SPRESENSE SDKでGPSを利用する流れは以下の通りです。

- ●Gnss.begin() 関数でGPSモジュールを初期化する
- ●Gnss.select() 関数を呼び出し、利用する衛星の種類を選択する(複数種類併用する場合は複数回呼び出す)
- ●Gnss.start() 関数でGPS信号の受信を開始する(バックグラウドで動作します)
- ●定期的に Gnss.getNavData() 関数を呼び出し、受信したGPS信号データ(＝位置情報)を取得する

詳しくは公式ドキュメントを参照して下さい。

GPS機能自体はバックグラウンドで動作するため、サブコアで実装する必要はなく、メインコアのプログラムに追加して問題ありません(そもそもGPSライブラリはメインコアでしか利用できないという制約があります)。

まずは IoT Central側で位置情報を受け取れるように、デバイステンプレートを書き換えましょう。
合わせて、Azure Functionsで作成した sendTypeを、URLパラメタとして緯度経度も受け取ってIoT Centralに送るように書き換えます。

★作業12★ デバイステンプレートの変更とAzure Functionsの変更

●IoT Centralのページから「デバイステンプレート」→「spresense-sound」を選びます

●「＋新規」→「場所」を選んで、以下の通り設定し、[保存] ボタンをクリックします
- ■Display Name　：デバイスの場所
- ■フィールド名　：location

●Azure Portalのページから「すべてのリソース」→「iotfunc～～」を選び、「関数」→「sendType」を選びます

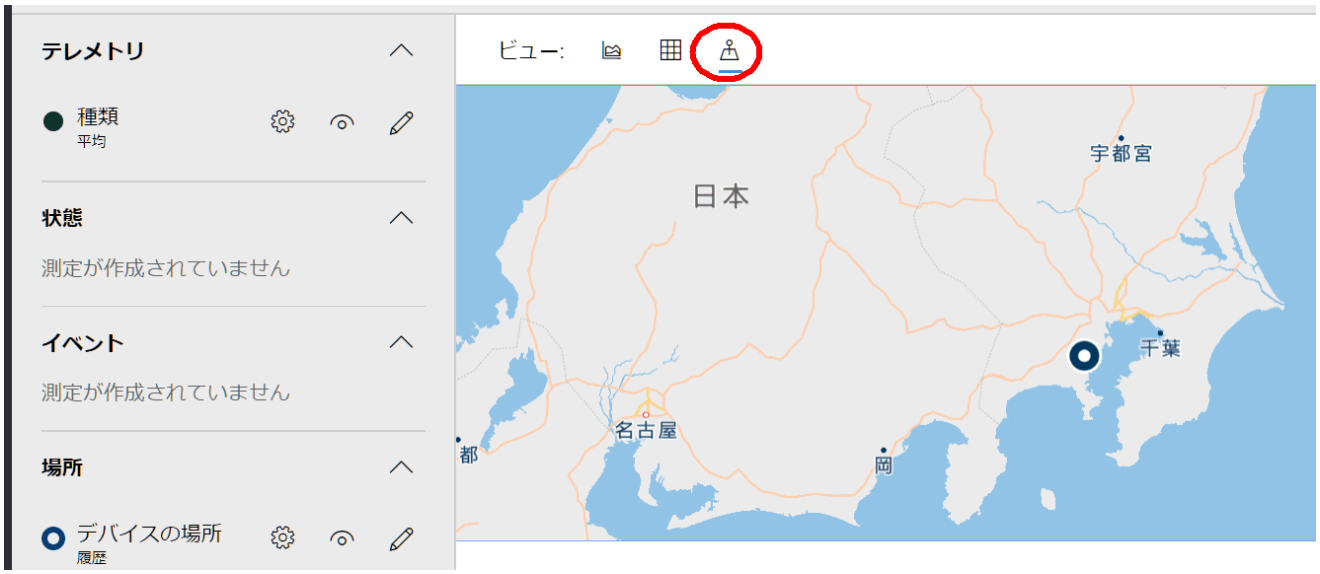
●index.jsのプログラムを以下のように書き換えて[保存]します(赤字部分のみ追加)

```
//IoT Centralへの接続文字列
const connectionString = " さっき作った接続文字列(HostName=～) ";

//IoT Centralへの通信に必要なライブラリ読み込み・オブジェクト作成
var devicemqtt = require('azure-iot-device-mqtt');
```

```
var device = require('azure-iot-device');
var client = devicemqtt.clientFromConnectionString(connectionString);
var Message = device.Message;

//HTTP経由で呼び出された時に実行される処理本体
module.exports = async function (context, req) {
    //リクエストパラメタにlabelがあれば
    if (req.query.label) {
        //IoT Centralに送信するデータを作り出す(JSONを作ってMessageでカプセル化)
        const json = createJson(req.query.label, req.query.lat,
 req.query.lon);
        var message = new Message(json);

        //IoT Centralに送信し、呼び出し元には送信成功を返す
        client.sendEvent(message, printResultFor('send'));
        context.res = {
            body: "Send " + req.query.label
        };
    }
    //リクエストパラメタにlabelがない場合は呼び出し元にエラーを返す
    else {
        context.res = {
            status: 400,
            body: "Please pass a data on the query string"
        };
    }
};

//IoT Centralに送信するデータをJSON形式で作り出す関数
function createJson(label, lat, lon)
{
    //現在時刻を取得
    const current_time = new Date().toISOString();
    //現在時刻とlabelを合わせたオブジェクトを作成
    const ddx = { timestamp: current_time, type: label, location: {lon:
 lon, lat: lat} };
    //JSON文字列化して返す
    return JSON.stringify(ddx);
}

//IoT Centralへの送信結果を受け取るコールバック関数
function printResultFor(op) {
    return function printResult(err, res) {
        if (err) console.log(op + ' error: ' + err.toString());
        if (res) console.log(op + ' status: ' + res.constructor.name);
    };
}
```

●[</>関数のURLの取得]で取得したURLを利用し、ブラウザで「https://～～.azurewebsites.net/api/sendType?label=0&lat=35.46821&lon=139.62227」にアクセスします

●IoT Centralのページから「デバイス」→「spresense-sound」→「spr-～～」を選び、ビューを に切り替えます



※横浜付近を拡大すると上記のようにマークが表示される事を確認して下さい

では最後に、SPRESENSE側のプログラムでGPSモジュールを有効化し、音の認識結果をFunctions経由で送る処理を、位置情報が取得できている場合はURLに緯度経度情報も付けてアクセスするよう、書き換えてみましょう。

★作業13★ GPS情報の取得と位置情報の送信処理の追加

●Arduino IDEで MicClassify(メインコア側のプログラム)に以下の通り処理を追加して下さい

◆#include群の最後に以下を追加

```
#include <GNSS.h>
```

◆「//MediaRecorderのイベント(録音開始とか終了とか)処理用コールバック関数」の前に以下を追加

```
//GPS用クラスと取得データ格納用変数
static SpGnss Gnss;
struct pos {
  int posDataExist;  //位置情報が取得できたかどうか
  double latitude;   //緯度
  double longitude;  //経度
} position;
```

◆setup関数の「//録音開始！」の前に以下を追加

```
//GPS(GNSS)モジュールを初期化
int result;
result = Gnss.begin();
if (result != 0)
{
  puts("Gnss begin error!!");
  exit(1);
}

//利用する衛星を選択(日本では以下の３つが利用できる)
Gnss.select(GPS);
Gnss.select(QZ_L1CA);
```

```
  Gnss.select(QZ_L1S);

  //GPSによる位置情報の取得を開始
  result = Gnss.start(COLD_START);
  if (result != 0)
  {
    puts("Gnss start error!!");
    exit(1);
  }
  else
  {
    puts("Gnss setup OK");
  }
```

◆loop関数の最後に以下を追加

```
  //GPSのデータが更新されていたら
  if(Gnss.isUpdate() == 1)
  {
    //GPSデータを取得
    SpNavData NavData;
    Gnss.getNavData(&NavData);

    //取得した位置情報を構造体変数positionにコピー
    position.posDataExist = NavData.posDataExist;  //位置情報が取得できたか
    position.latitude = NavData.latitude;          //緯度
    position.longitude = NavData.longitude;        //経度

    //位置情報が取得できていたらサブコアに通知＆デバグ用に出力
    if(position.posDataExist != 0)
    {
      MP.Send(2, (void*)&position, 1);    //メッセージIDは２
      printf("pos:%3.5f, %3.5f\n", position.latitude,
 position.longitude);
    }
  }
```

●「ツール」→「Core」から「**MainCore**」を選んでから、[マイコンボードに書き込む]ボタンをクリックし、プログラムを書き込んで実行します

※GPS信号を受信できる場所(基本的には屋外)に1～2分置いておくと、シリアルモニタに位置情報が表示されるはずです

●Arduino IDEで MicClassify_sub(サブコア側のプログラム)に以下の通り処理を追加して下さい

◆「//HTTPレスポンスをチェックする処理」の直前に以下を追加

```
  //位置情報格納用変数
  struct pos {
    int posDataExist;  //位置情報が取得できたかどうか
    double latitude;   //緯度
    double longitude;  //経度
  } *position;
```

◆loop関数内の「//メッセージが取得できた場合」のif文を以下のように書き換える

```
    //メッセージが取得できた場合
```

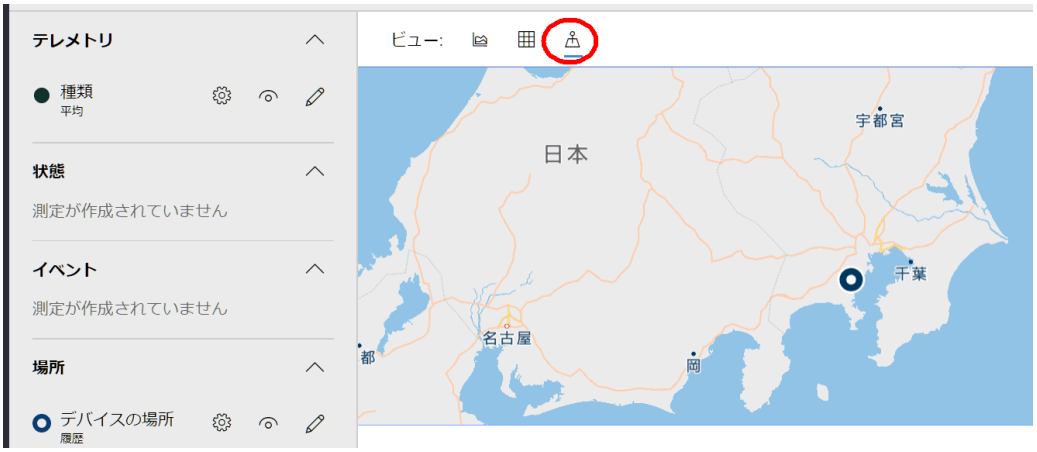
```
if(ret >= 0){
  //送信時に指定されたメッセージIDが1なら
  if(msgid == 1){
    //受信した認識結果を取り出す
    int label = *(int*)msgdata;
    //Azureに認識結果を送信する処理を呼び出す
    sendType(label);
    waitCnt = 0;
  }
  //送信時に指定されたメッセージIDが2なら
  else if(msgid == 2)
  {
    //受信した位置情報のメモリアドレスを取得
    position = (struct pos*)msgdata;
  }
}
```

◆sendType関数内の「//URLを組み立てる」の部分を以下のように書き換える

```
//URLを組み立てる
String buf = "/api/sendType?typ=" + String(label);
if(position->posDataExist == 1){
  buf += String("&lat=") + String(position->latitude,6) +
String("&lon=") + String(position->longitude,6);
}
```

●「ツール」→「Core」から「**SubCore1**」を選んでから、[マイコンボードに書き込む]ボタンをクリックし、プログラムを書き込んで実行します

※GPS信号を受信できる場所に1～2分置いた後、IoT Centralのページを見ると、位置情報が反映されているはずです



以上でAI×IoTのシステムを構成する主要な仕組みは全て実装できました。あとはぜひ応用してビジネスやものづくりに活かして下さい。お疲れ様でした！

なお、DoS攻撃で課金される可能性もゼロではありませんので、気になる場合は Azure Functionsを削除しておいた方が無難です。

9. その他の機能追加

最後に、より実用的にするために、IoT Centralに送られてきた情報がある条件にマッチした場合にSMS等で通知する、という機能を追加してみます。これがあれば人が常に値を監視する必要がなくなります。

今回は、5分以内に音声を10回以上認識した場合(にぎやかになった場合)にSMSで通知する仕組みを作ってみます。

IoT Central自体にはSMS送信する機能はありませんが、Azureの Azure Monitorというサービスを利用すると、「SMSやメールを送信する」といったアクションを定義できます。

そして、IoT Centralではテレメトリの条件(＝ルールと呼びます)によって Azure Monitorのアクションを呼び出せるため、これを組み合わせれば「条件に合致した場合にSMSを送る」という仕組みを作れます。

★作業14★ Azure Monitorによるアクションの定義と IoT Centralによるルールの定義

●Azure Portalのページで、ページ上部の検索窓に「モニター」と入力し、出てきたサービスの「モニター」をクリックします

●「アラート」をクリックし、続いて [アクションの管理] をクリックします

●アクションの管理画面で、[アクショングループの追加] をクリックし、以下の通り設定して[OK] → [OK] をクリックします

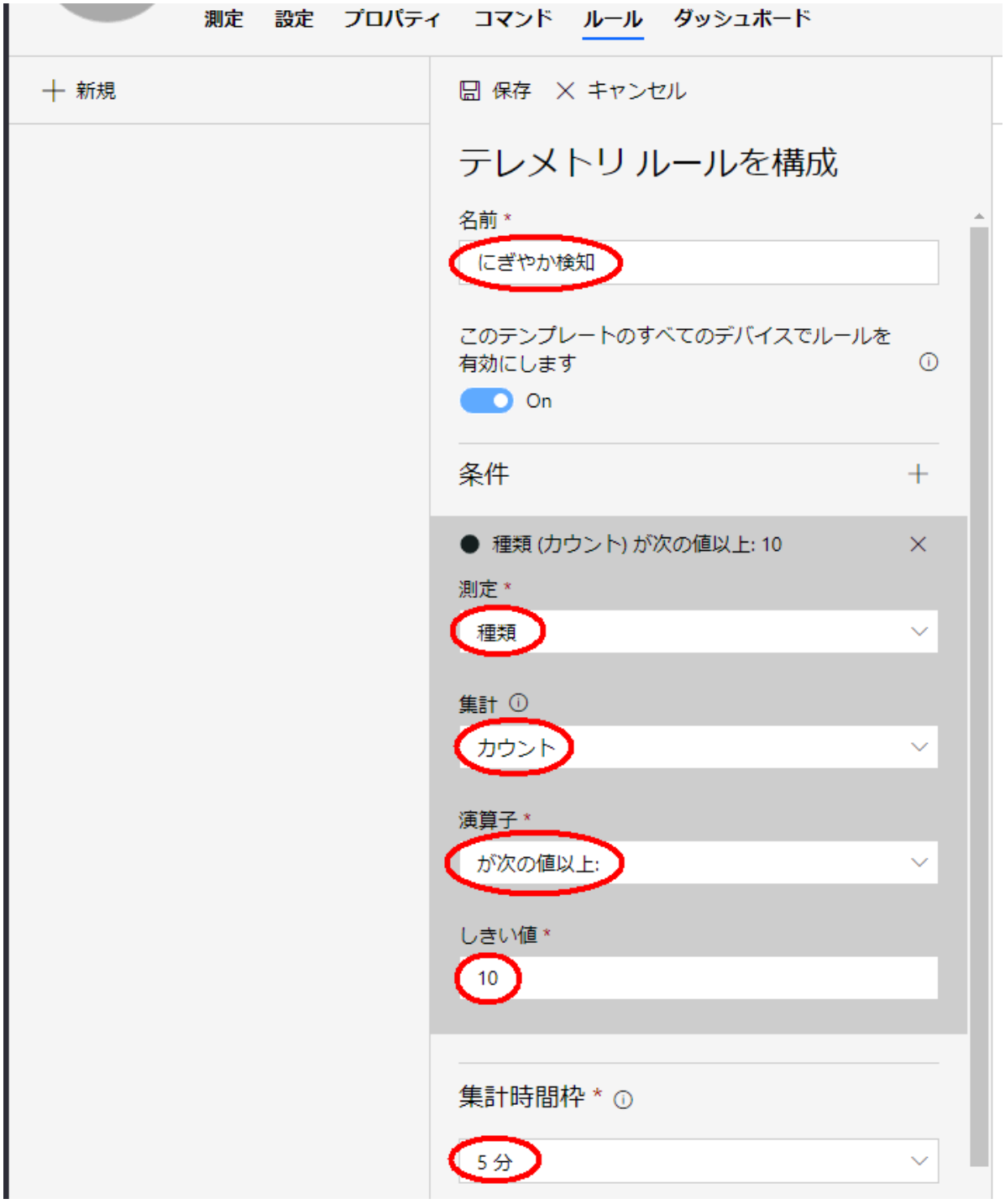
■アクショングループ名　　：任意(SMSNotify など)
■短い名前　　：任意(SMS など)
■サブスクリプション　：任意
■リソースグループ　　：任意 or 新規作成
■アクション名　　：任意(SMSNotify など)
■アクションタイプ　　：電子メール/SMS/プッシュ/音声
■SMS　　：チェックを付ける
■国コード　　：81
■電話番号　　：ご自身の携帯電話番号から先頭の0を除いた番号(9012345678 など)

※SMSの発信は1回あたり5円程度の費用が発生します

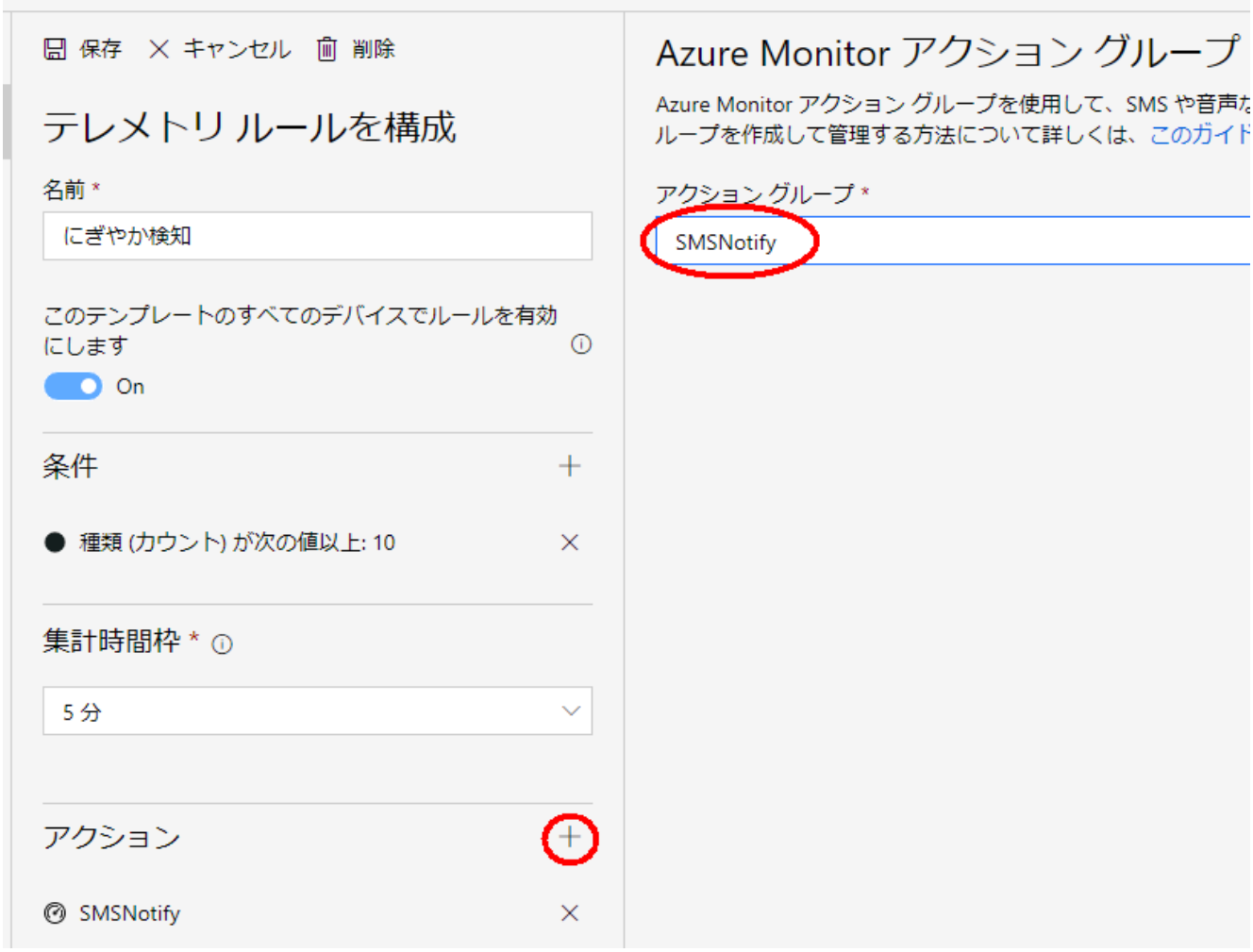
●1分程度で指定した電話番号宛に「All done.」といったメッセージのSMSが送られてくるので確認して下さい。もし送られてこない場合は上記の設定を見直して下さい

●IoT Centralのページから「デバイステンプレート」→「spresense-sound」を選び、「ルール」をクリックします

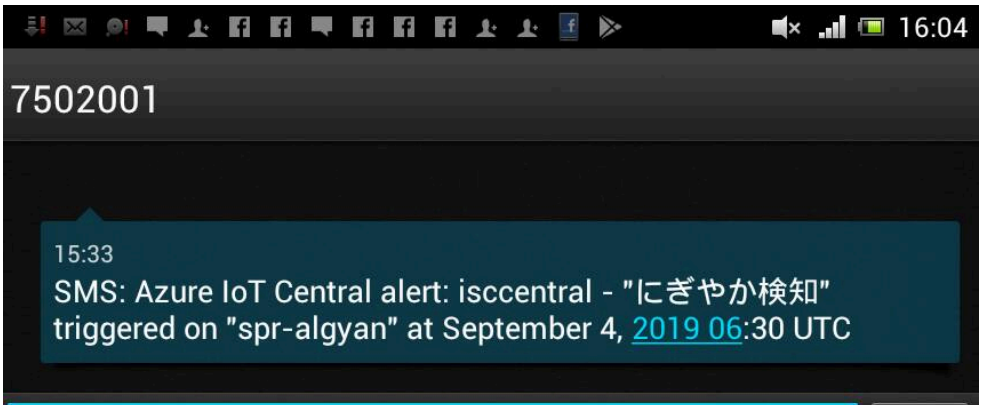
●「＋新規」→「テレメトリ」を選び、以下の通り設定し、[保存]をクリックします
■名前　　：任意 (「にぎやか検知」など。SMSで送られるメッセージにこの名前が含まれます)
■測定　　：種類 (これは作業9で作ったテレメトリの名前です)
■集計　　：カウント
■集計　　：が次の値以上
■しきい値　：10
■集計時間枠：5分



●さらに、「アクション」の右隣の「＋」をクリックして「Azure Monitor アクショングループ」を選択し、以下の通りアクショングループとして「SMSNotify」(さっき作ったもの)を選び、[保存]をクリックします



●SPRESENSEで実際に拍手か声を5分以内に10回以上認識させるか、または Azure Functionsを直接10回呼び出す(**`http://～～.azurewebsites.net/api/sendType?label=0`** に10回アクセスする)と、SMSでメッセージが届く事を確認して下さい



※SMSが届かない場合は、まずデバイスのダッシュボードでグラフが表示されているかどうか(データが受信できているかどうか)を確認し、その上でデバイステンプレートのルールを見直して下さい

今回はあまり意味のある通知ではありませんが、応用すれば、例えば「工作機械の動作音がいつもと違ってきたら目メンテナンス会社に通知する」や「不審者がガラスを割って入ってきた音を検知したら住人に通知する」といった仕組みが作れるはずです。

10. 参考資料等

- Spresense Arduino Library 開発ガイド(公式)
- クラスライブラリリファレンス(公式)
- SPRESENSEユーザーグループ
- じゃんけん画像認識ハンズオン資料

※Adafruit ILI9341のLCD(例えばこれ)と、LCD接続コネクタを用意すれば試せます

本テキストの内容についてご質問やご指摘がありましたら
情報科学専門学校の武藤まで遠慮なくご連絡ください → muto@gn.iwasaki.ac.jp