Fonctionnement et utilisation du système de détection automatique de sirène de l'Arduino RP2040

Table des matières :

I. Fonctionnement des codes Arduino : 2

1. Installation des libraries 2

2. Enregistrement d'un signal audio avec le microphone PDM et envoi du signal sous format PCM par le port série avec le code "ArduinoRecording.ino" 3

3. Prédiction de la classe d'un signal audio avec le code "SirenDetection.ino" 4

II. Schéma de fonctionnement de la carte Arduino Nano RP2040 Connect 6

III. Utilisation de l'Arduino Nano RP2040 Connect 7

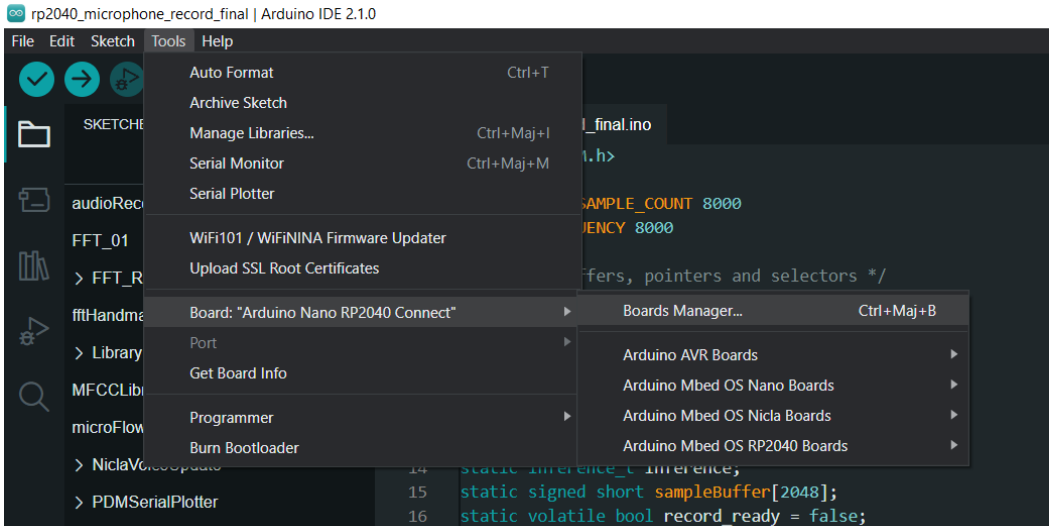
1. Par rapport au code git ("SirenDetection.ino") implémentant le modèle CNN de détection 7

2. En décommentant les parties suivantes du code "SirenDetection.ino" : envoi possible des prédictions par le port série 7

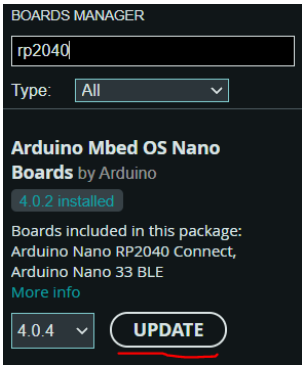
I. Fonctionnement des codes Arduino :

1. Installation des libraries

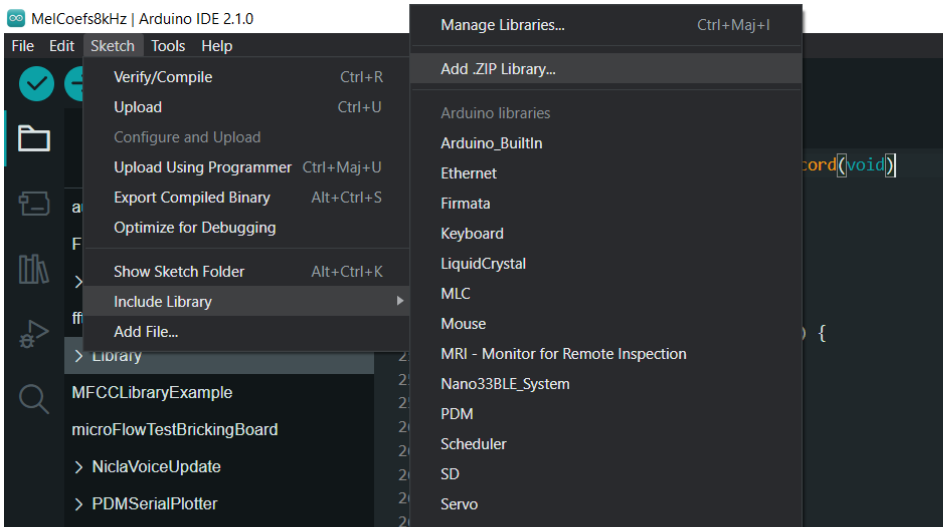
a. Installation des libraries de la carte : depuis un IDE Arduino (ici IDE 2.0) : Tools → Board → Boards Manager



- Chercher “RP2040” ou “Arduino Mbed OS Nano Boards” et installer :



b. Importer les libraries git avec l’IDE Arduino (Sketch → Include Library → Add .ZIP Library) ou copier les libraries git dans le dossier “libraries” d’Arduino :



2. Enregistrement d'un signal audio avec le microphone PDM et envoi du signal sous format PCM par le port série avec le code "ArduinoRecording.ino"

- Envoi d'1s de valeurs PCM par le port série (fréquence de 8kHz); 1s d'enregistrement et ~1s de communication du port série.

a. Initialise le port série :

dans la fonction setup() :

```
// put your setup code
Serial.begin(115200);
// comment out the bel
while (!Serial);
```

b. Initialise le microphone PDM et l'espace mémoire dédié au stockage des valeurs :

appelé dans setup() :

```
if (microphone_inference_start(RAW_SAMPLE_COUNT) == false) {
```

```
static bool microphone_inference_start(uint32_t n_samples)
{
    inference.buffer = (int16_t *)malloc(n_samples * sizeof(int16_t));

    if(inference.buffer == NULL) {
        Serial.println("Could not allocate");
        return false;
    }

    inference.buf_count  = 0;
    inference.n_samples  = n_samples;
    inference.buf_ready  = 0;

    // configure the data receive callback
    PDM.onReceive(pdm_data_ready_inference_callback);

    PDM.setGain(40);

    PDM.setBufferSize(2048);
    delay(250);

    // initialize PDM with:
    // - one channel (mono mode)
    if (!PDM.begin(1, FREQUENCY)) {
        Serial.println("ERR: Failed to start PDM!");
        microphone_inference_end();
        return false;
    }

    return true;
}
```

c. Rempli le buffer de valeurs PCM :

appelé dans loop() :

```
bool m = microphone_inference_record();
```

```
static bool microphone_inference_record(void)
{
    bool ret = true;

    record_ready = true;
    while (inference.buf_ready == 0) {
        delay(10);
    }

    inference.buf_ready = 0;
    record_ready = false;

    return ret;
}
```

```
static void pdm_data_ready_inference_callback(void)
{
    int bytesAvailable = PDM.available();

    // read into the sample buffer
    int bytesRead = PDM.read((char *)&sampleBuffer[0], bytesAvailable);

    if ((inference.buf_ready == 0) && (record_ready == true)) {

        for(int i = 0; i < bytesRead>>1; i++) {
            inference.buffer[inference.buf_count++] = sampleBuffer[i];

            if(inference.buf_count >= inference.n_samples) {
                inference.buf_count = 0;
                inference.buf_ready = 1;
                break;
            }
        }
    }
}
```

d. Envoie les valeurs du buffer sur le port série :
dans la fonction loop() :

```
for(uint16_t i=0; i<inference.n_samples; i++){
  //count +=1;
  Serial.println(inference.buffer[i]);
}
```

3. <u>Prédiction de la classe d'un signal audio avec le code "SirenDetection.ino"</u>

a. Traitement audio (après avoir rempli un buffer de valeurs PCM comme dans le programme "AudioRecording.ino") :
   i. Instantiation des métriques pour récupérer les features voulues lors du traitement audio :

```
#define RAW_SAMPLE_COUNT 2048 //8192          //longueur du signal à analyser
#define FREQUENCY 8000                       //fréquence d'enregistrement du signal
#define CHANNELS 1                           //nombre de canaux de l'enregistrement
#define SPLIT_AUDIO_LENGTH 1 //2             //permet de diviser le traitement du signal en plusi
                                             //et de concaténer ces différents traitements à la f

static const uint16_t windowSize = 512;//512; //taille de la fenêtre glissante
static const uint16_t hopLength = 384;//448;  //recouvrement de la fenêtre
static const uint8_t nbFilters = 32;          //nombre de filtre du Mel-scale filterbank
static const uint8_t nbFrames = int((RAW_SAMPLE_COUNT/SPLIT_AUDIO_LENGTH-windowSize)/hopLength)+1;
```

ii. dans la fonction loop() :

```
//Traitement du signal audio enregistré
uint8_t framesCounter = 0;
for(uint8_t s=0; s<SPLIT_AUDIO_LENGTH;s++){ //si le traitement du signal se fait sur des parties indépendantes

  double* framesBuffer = new double[nbFrames * (windowSize >> 1)];

  for(uint8_t n = 0; n<nbFrames; n++){
    double* bufferData = new double[windowSize];
    double* bufferImag = new double[windowSize];
    for(uint16_t i = 0; i<windowSize; i++){
      //Normalisation
      bufferData[i] = normalization(double(inference.buffer[i+(n*hopLength)+(s*(RAW_SAMPLE_COUNT/SPLIT_AUDIO_LENGTH))]), inference.min, inference.max);
      bufferImag[i] = 0;
    }

    //FFT et calcul des magnitudes
    FFT.Windowing(bufferData, windowSize, FFT_WIN_TYP_HAMMING, FFT_FORWARD); // Weigh data
    FFT.Compute(bufferData, bufferImag, windowSize, FFT_FORWARD); // Compute FFT
    FFT.ComplexToMagnitude(bufferData, bufferImag, windowSize); // Compute magnitudes

    delete[] bufferImag;

    for(uint16_t i = 0; i<windowSize>>1; i++){
      framesBuffer[n * (windowSize >> 1) + i] = bufferData[i];
    }

    delete[] bufferData;
  }

  //Calcul des Mel-scale filterbanks
  FFT.MelCoefficients(framesBuffer, nbFrames, windowSize, nbFilters, FREQUENCY);

  //convertion double vers float et stockage des Mel-scale filterbanks dans le bon t
  for (int i = 0 ; i < nbFrames; i++)
  {
    for (int j = 0; j < nbFilters; j++){
      coefficients[i+framesCounter][j] = (float) framesBuffer[i * nbFilters + j];
    }
  }

  delete[] framesBuffer;

  framesCounter += nbFrames;
```

Fenêtres glissantes sur les valeurs PCM → normalisation sur ces valeurs → pondération par fenêtre de Hamming → calcul des FFT de ces valeurs → calcul des magnitudes → calcul des Mel-scale filter bank coefficients

b. Prédiction à partir du modèle importé et des données traitées :
   i. Fonction utilisée par la librairie du modèle pour réaliser plus efficacement les prédictions :

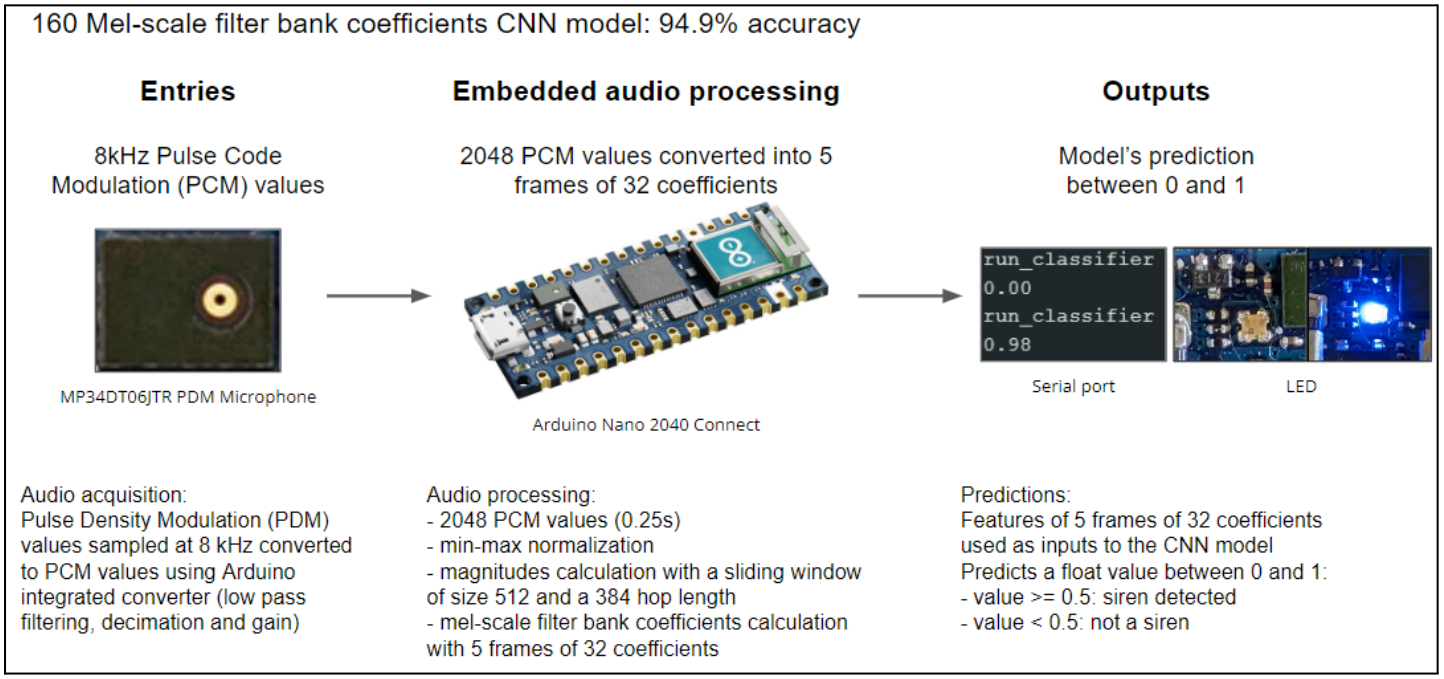
```
int raw_feature_get_data(size_t offset, size_t length, float *out_ptr) {
    memcpy(out_ptr, coefficients + offset, length * sizeof(float));
    return 0;
}
```

ii. Prédiction réalisée sur la totalité des Mel-scale filter bank coefficients de l'audio analysé :

dans la fonction loop() :

```
//Prédiction à partir des frames de Mel-scale filterbanks

ei_impulse_result_t result = { 0 };

// the features are stored into flash, and we don't want to load everything into
signal_t features_signal;
features_signal.total_length = sizeof(coefficients) / sizeof(coefficients[0][0]);
features_signal.get_data = &raw_feature_get_data;

  //Prédiction à partir des features traitées en mémoire flash et résultat stocké
EI_IMPULSE_ERROR res = run_classifier(&features_signal, &result, false);//  debug
if (res != EI_IMPULSE_OK) {
    //ei_printf("ERR: Failed to run classifier (%d)\n", res);
    return;
}

  //Accès à la valeur prédite
float value = result.classification[0].value;
```

II. Schéma de fonctionnement de la carte Arduino Nano RP2040 Connect



III. Utilisation de l'Arduino Nano RP2040 Connect

1. Par rapport au code git ("SirenDetection.ino") implémentant le modèle CNN de détection
   - brancher la carte par micro-usb pour l'alimenter (PC ou batterie externe),
   - le modèle tourne; lorsque le son analysé est prédit sirène, la led s'allume en bleue, sinon s'éteint.

2. En décommentant les parties suivantes du code "SirenDetection.ino" : envoi possible des prédictions par le port série

   dans la fonction setup() :

   -Permet d'établir la communication série à un baud rate de 115200 :

```
// put your setup code h
//Serial.begin(115200);

// comment out the below
//while (!Serial);
```

   dans la fonction loop() :

   -Permet d'envoyer la valeur prédite par le port série :

```
//Traitement de la valeur prédite

//Serial.println(value);
```

   - brancher la carte à un appareil permettant une communication série (PC),
   - les prédictions peuvent être lues par l'IDE Arduino ou par un programme récupérant les communications séries.