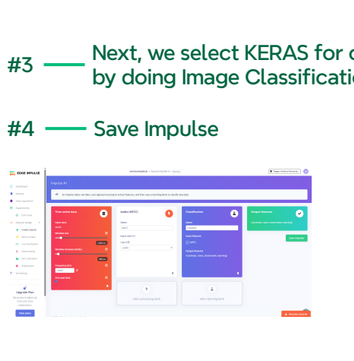
Embedded A.I - 5 AudioClassification Part2

TinyML
- TinyML refers to deploying machine learning models on tiny devices (like MCUs or edge devices) with:
- ML Workflow
  - #1 — Problem
  - #2 — Data Collection
  - #3 — Data Cleaning & Labeling
  - #4 — Feature Extraction
  - #5 — Model Training & Evaluation
  - #6 — Deployment
  - #7 — Inference

Create Edge Impulse Account

Collect Audio Dataset
- ✅ Method 1: OFFLINE METHOD using SDCard
  - #1
  - #2
  - #3
  - In Arduino Serial monitor
  - #10 — Uploading collected sound data
    - When the raw dataset is defined and collected, we should initiate a new project at Edge Impulse.
    - Once the project is created, select the Upload Existing Data tool in the Data Acquisition section.
    - And upload them to the Studio (You can automatically split data in train/test). Repete to all classes and all raw data.
    - All data on dataset have a 1s length, but the samples recorded in the previous section have 10s and must be split into 1s samples to be compatible.
    - Once inside the tool, split the data into 1-second records. If necessary, add or remove segments.
    - This procedure should be repeated for all samples.
    - Observe Train / Test Split
  - Note — Keep samples 1 second in length
- ❌ Method 2: ONLINE METHOD using CLI daemon

Create Impulse (Pre-Process / Model definition)
- An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.
- Go to Impulse Design > Create Impulse
- #1 — First, we will take the data points with a 1-second window, augmenting the data, sliding that window each 500ms. — Note that the option zero-pad data is set. — This is important to fill with zeros samples smaller than 1 second (in some cases, I reduced the 1000 ms window on the split tool to avoid noises and spikes).
- #2 — Each 1-second audio sample should be pre-processed and converted to an image (for example, 13 x 49 x 1).
  - Mel-frequency cepstral coefficients (MFCCs) are coefficients that collectively make up an MFC. — They are derived from a type of cepstral representation of the audio clip (a nonlinear "spectrum-of-a-spectrum").
  - We will use MFCC, which extracts features from audio signals using Mel Frequency Cepstral Coefficients, which are great for the human voice.
- #3 — Next, we select KERAS for classification that builds our model from scratch by doing Image Classification using Convolution Neural Network.
- #4 — Save Impulse



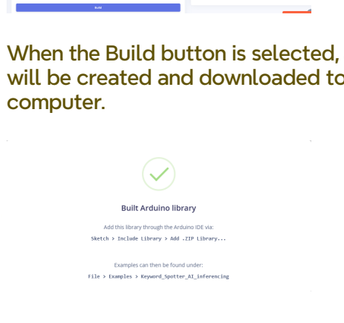
Pre-Processing (MFCC)
- The next step is to create the images to be trained in the next phase. — We can keep the default parameter values or take advantage of the DSP Autotuneparameters option, which we will do.
- Click — Autotune Parameters
- Save Parameters
- Click — Generate Features

Model Design and Training
- We will use a Convolution Neural Network (CNN) model. — The basic architecture is defined with two blocks of Conv1D + MaxPooling (with 8 and 16 neurons, respectively) and a 0.25 Dropout. — And on the last layer, after Flattening four neurons, one for each class.
- As hyper-parameters, we will have a Learning Rate of 0.005 and a model that will be trained by 100 epochs. — We will also include data augmentation, as some noise. The result seems OK.

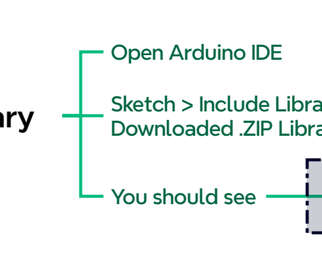
Retrain
- Optional

Test Model in Studio

Export for Deployment
- Edge Impulse will package all the needed libraries, preprocessing functions, and trained models, downloading them to your computer.



- When the Build button is selected, a Zip file will be created and downloaded to your computer.



Upload Downloaded ZIP Library
- Open Arduino IDE
- Sketch > Include Library > Add your Downloaded .ZIP Library
- You should see — Library Installed

Board Settings
- Board: "ESP32S3 Dev Module"
- USB CDC On Boot: "Enabled"
- PSRAM: "OPI PSRAM"
- Flash Mode: "QIO 80MHz"
- Flash Size: "8MB (64Mb)"
- Partition Scheme: "8M with spiffs (3MB APP/1.5MB SPIFFS)"

Compile and upload the code
- Important Note ⭐⭐⭐ — Change the Header File (.h) name in the below code

#1 Testing the Classification

```
/* Edge Impulse Arduino examples
 * ESP32 Board Package 3.2.0 Compatible Version
 * For XIAO ESP32S3 Sense
 */

// If your target is limited in memory remove this macro to save 10K RAM
#define EIDSP_QUANTIZE_FILTERBANK   0

/* Includes ---------------------------------------------------------------- */
#include <Keyword_Spotter_AI_inferencing.h>
#include <ESP_I2S.h>

/** Audio buffers, pointers and selectors */
typedef struct {
    signed short *buffers[2];
    unsigned char buf_select;
    unsigned char buf_ready;
    unsigned int buf_count;
    unsigned int n_samples;
} inference_t;

static inference_t inference;
static bool record_ready = false;
static signed short *sampleBuffer;
static bool debug_nn = false;
static int print_results = -(EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW);

// Create I2S instance
I2SClass I2S;

/**
 * @brief      Arduino setup function
 */
void setup()
{
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Edge Impulse Inferencing Demo");
    Serial.println("XIAO ESP32S3 Sense - ESP32 v3.2.0 Compatible");

    // summary of inferencing settings
    ei_printf("Inferencing settings:\n");
    ei_printf("\tInterval: ");
    ei_printf_float((float)EI_CLASSIFIER_INTERVAL_MS);
    ei_printf(" ms.\n");
    ei_printf("\tFrame size: %d\n", EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE);
    ei_printf("\tSample length: %d ms.\n", EI_CLASSIFIER_RAW_SAMPLE_COUNT / 16);
    ei_printf("\tNo. of classes: %d\n", sizeof(ei_classifier_inferencing_categories) /
sizeof(ei_classifier_inferencing_categories[0]));

    // Print the classes
    ei_printf("Classes: ");
    for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
        ei_printf("%s", ei_classifier_inferencing_categories[ix]);
        if (ix != EI_CLASSIFIER_LABEL_COUNT - 1) {
            ei_printf(", ");
        }
    }
    ei_printf("\n");

    run_classifier_init();
    ei_printf("\nStarting continuous inference in 2 seconds...\n");
    delay(2000);

    if (microphone_inference_start(EI_CLASSIFIER_SLICE_SIZE) == false) {
        ei_printf("ERR: Could not allocate audio buffer (size %d)\r\n",
EI_CLASSIFIER_RAW_SAMPLE_COUNT);
        return;
    }

    ei_printf("Recording...\n");
    ei_printf("Say your keywords: activate, warning, stand down\n");
}

/**
 * @brief      Arduino main function
 */
void loop()
{
    bool m = microphone_inference_record();
    if (!m) {
        ei_printf("ERR: Failed to record audio...\n");
        return;
    }

    signal_t signal;
    signal.total_length = EI_CLASSIFIER_SLICE_SIZE;
    signal.get_data = &microphone_audio_signal_get_data;
    ei_impulse_result_t result = {0};

    EI_IMPULSE_ERROR r = run_classifier_continuous(&signal, &result, debug_nn);
    if (r != EI_IMPULSE_OK) {
        ei_printf("ERR: Failed to run classifier (%d)\n", r);
        return;
    }

    if (++print_results >= (EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW)) {
        // print the predictions
        ei_printf("Predictions ");
        ei_printf("(DSP: %d ms., Classification: %d ms., Anomaly: %d ms.)",
            result.timing.dsp, result.timing.classification, result.timing.anomaly);
        ei_printf(": \n");

        float max_confidence = 0;
        String predicted_class = "";

        for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
            ei_printf("    %s: ", result.classification[ix].label);
            ei_printf_float(result.classification[ix].value);
            ei_printf("\n");

            if (result.classification[ix].value > max_confidence) {
                max_confidence = result.classification[ix].value;
                predicted_class = result.classification[ix].label;
            }
        }

        // Show top prediction
        ei_printf("TOP: %s (", predicted_class.c_str());
        ei_printf_float(max_confidence);
        ei_printf(")\n");


        // Detection logic
        if (max_confidence > 0.6 && predicted_class != "noise") {
            ei_printf("\n*** KEYWORD DETECTED ***\n");
            ei_printf("Detected: %s\n", predicted_class.c_str());
            ei_printf("Confidence: ");
            ei_printf_float(max_confidence);
            ei_printf("\n");

            if (predicted_class == "activate") {
                ei_printf("-> ACTIVATE command recognized!\n");
            } else if (predicted_class == "warning") {
                ei_printf("-> WARNING command recognized!\n");
            } else if (predicted_class == "stand down") {
                ei_printf("-> STAND DOWN command recognized!\n");
            }
            ei_printf("*************************\n\n");
        }

#if EI_CLASSIFIER_HAS_ANOMALY == 1
        ei_printf("    anomaly score: ");
        ei_printf_float(result.anomaly);
        ei_printf("\n");
#endif

        print_results = 0;
    }
}

/**
 * @brief      Read audio data from I2S (polling approach for ESP32 v3.x)
 */
static void read_audio_samples(void)
{
    // Read available samples from I2S
    while (I2S.available() > 0 && inference.buf_count < inference.n_samples) {
        int sample = I2S.read();
        if (sample != -1) {
            // Store the sample in current buffer
            inference.buffers[inference.buf_select][inference.buf_count] = (signed short)(sample &
0xFFFF);
            inference.buf_count++;

            // If buffer is full, switch to next buffer
            if (inference.buf_count >= inference.n_samples) {
                inference.buf_select ^= 1;
                inference.buf_count = 0;
                inference.buf_ready = 1;
                break;
            }
        }
    }
}

/**
 * @brief      Init inferencing struct and setup/start PDM for ESP32 v3.x
 */
static bool microphone_inference_start(uint32_t n_samples)
{
    inference.buffers[0] = (signed short *)malloc(n_samples * sizeof(signed short));
    if(inference.buffers[0] == NULL) {
        ei_printf("ERROR: Could not allocate buffer 0\n");
        return false;
    }

    inference.buffers[1] = (signed short *)malloc(n_samples * sizeof(signed short));
    if(inference.buffers[1] == NULL) {
        ei_printf("ERROR: Could not allocate buffer 1\n");
        ei_free(inference.buffers[0]);
        return false;
    }

    sampleBuffer = inference.buffers[inference.buf_select];

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

    // ESP32 v3.x I2S setup for XIAO ESP32S3 Sense
    ei_printf("Setting up I2S for ESP32 v3.x...\n");

    // Set PDM pins for XIAO ESP32S3 Sense
    ei_printf("Setting PDM pins (CLK=42, DATA=41)...\n");
    I2S.setPinsPdmRx(42, 41);  // CLK=42, DATA=41

    // Begin I2S in PDM RX mode
    ei_printf("Starting I2S in PDM RX mode...\n");
    if (!I2S.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT,
I2S_SLOT_MODE_MONO)) {
        ei_printf("ERROR: Failed to start I2S PDM!\n");
        microphone_inference_end();
        return false;
    }

    record_ready = true;
    ei_printf("✅ I2S PDM microphone started successfully for ESP32 v3.x!\n");
    return true;
}

/**
 * @brief      Wait on new data
 */
static bool microphone_inference_record(void)
{
    bool ret = true;

    if (inference.buf_ready == 1) {
        ei_printf("Error sample buffer overrun. Decrease the number of slices per model
window\n");
        ret = false;
    }

    while (inference.buf_ready == 0) {
        // Continuously read audio samples (polling approach)
        read_audio_samples();
        delay(1);
    }

    inference.buf_ready = 0;
    return ret;
}

/**
 * Get raw audio signal data
 */
static int microphone_audio_signal_get_data(size_t offset, size_t length, float *out_ptr)
{
    numpy::int16_to_float(&inference.buffers[inference.buf_select ^ 1][offset], out_ptr, length);
    return 0;
}

/**
 * @brief      Stop PDM and release buffers
 */
static void microphone_inference_end(void)
{
    I2S.end();
    ei_free(inference.buffers[0]);
    ei_free(inference.buffers[1]);
}

#if !defined(EI_CLASSIFIER_SENSOR) || EI_CLASSIFIER_SENSOR !=
EI_CLASSIFIER_SENSOR_MICROPHONE
#error "Invalid model for current sensor."
#endif
```

#2 Voice-Activated Device

```
/* Edge Impulse Arduino examples
 * ESP32 Board Package 3.2.0 Compatible Version
 * For XIAO ESP32S3 Sense with 3-Color LED Control
 */

// If your target is limited in memory remove this macro to save 10K RAM
#define EIDSP_QUANTIZE_FILTERBANK   0

/* Includes ---------------------------------------------------------------- */
#include <Keyword_Spotter_AI_inferencing.h>
#include <ESP_I2S.h>

// LED Pin Definitions
#define LED_RED_PIN     5    // Red LED - for "stand down"
#define LED_GREEN_PIN   3    // Green LED - for "activate"
#define LED_YELLOW_PIN  4    // Yellow LED - for "warning"

/** Audio buffers, pointers and selectors */
typedef struct {
    signed short *buffers[2];
    unsigned char buf_select;
    unsigned char buf_ready;
    unsigned int buf_count;
    unsigned int n_samples;
} inference_t;

static inference_t inference;
static bool record_ready = false;
static signed short *sampleBuffer;
static bool debug_nn = false;
static int print_results = -(EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW);

// Create I2S instance
I2SClass I2S;

// LED control functions
void turnOffAllLEDs() {
    digitalWrite(LED_RED_PIN, LOW);
    digitalWrite(LED_GREEN_PIN, LOW);
    digitalWrite(LED_YELLOW_PIN, LOW);
}

void setLEDState(String keyword) {
    // Turn off all LEDs first
    turnOffAllLEDs();

    // Turn on appropriate LED based on keyword
    if (keyword == "activate") {
        digitalWrite(LED_RED_PIN, LOW);
        digitalWrite(LED_YELLOW_PIN, LOW);
        digitalWrite(LED_GREEN_PIN, HIGH);

        ei_printf("🟢 GREEN LED ON - Activate\n");
    } else if (keyword == "warning") {

        digitalWrite(LED_RED_PIN, LOW);
        digitalWrite(LED_YELLOW_PIN, HIGH);
        digitalWrite(LED_GREEN_PIN, LOW);

        ei_printf("🟡 YELLOW LED ON - Warning\n");
    } else if (keyword == "stand down") {
        digitalWrite(LED_GREEN_PIN, LOW);
        digitalWrite(LED_YELLOW_PIN, LOW);
        digitalWrite(LED_RED_PIN, HIGH);
        ei_printf("🔴 RED LED ON - Stand Down\n");
    }
}



/**
 * @brief      Arduino setup function
 */
void setup()
{
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Edge Impulse Inferencing Demo");
    Serial.println("XIAO ESP32S3 Sense - ESP32 v3.2.0 Compatible with LED Control");

    // Initialize LED pins
    pinMode(LED_RED_PIN, OUTPUT);
    pinMode(LED_GREEN_PIN, OUTPUT);
    pinMode(LED_YELLOW_PIN, OUTPUT);

    // Turn off all LEDs initially
    turnOffAllLEDs();

    // summary of inferencing settings
    ei_printf("Inferencing settings:\n");
    ei_printf("\tInterval: ");
    ei_printf_float((float)EI_CLASSIFIER_INTERVAL_MS);
    ei_printf(" ms.\n");
    ei_printf("\tFrame size: %d\n", EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE);
    ei_printf("\tSample length: %d ms.\n", EI_CLASSIFIER_RAW_SAMPLE_COUNT / 16);
    ei_printf("\tNo. of classes: %d\n", sizeof(ei_classifier_inferencing_categories) / sizeof(ei_classifier_inferencing_categories[0]));

    // Print the classes
    ei_printf("Classes: ");
    for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
        ei_printf("%s", ei_classifier_inferencing_categories[ix]);
        if (ix != EI_CLASSIFIER_LABEL_COUNT - 1) {
            ei_printf(", ");
        }
    }
    ei_printf("\n");

    ei_printf("\nLED Control:\n");
    ei_printf("🟢 GREEN LED (Pin %d) = 'activate'\n", LED_GREEN_PIN);
    ei_printf("🟡 YELLOW LED (Pin %d) = 'warning'\n", LED_YELLOW_PIN);
    ei_printf("🔴 RED LED (Pin %d) = 'stand down'\n", LED_RED_PIN);

    run_classifier_init();
    ei_printf("\nStarting continuous inference in 2 seconds...\n");
    delay(2000);

    if (microphone_inference_start(EI_CLASSIFIER_SLICE_SIZE) == false) {
        ei_printf("ERR: Could not allocate audio buffer (size %d)\r\n", EI_CLASSIFIER_RAW_SAMPLE_COUNT);
        return;
    }

    ei_printf("Recording...\n");
    ei_printf("Say your keywords: activate, warning, stand down\n");
    ei_printf("Watch the LEDs for visual feedback!\n");
}

/**
 * @brief      Arduino main function
 */
void loop()
{
    bool m = microphone_inference_record();
    if (!m) {
        ei_printf("ERR: Failed to record audio...\n");
        return;
    }

    signal_t signal;
    signal.total_length = EI_CLASSIFIER_SLICE_SIZE;
    signal.get_data = &microphone_audio_signal_get_data;
    ei_impulse_result_t result = {0};

    EI_IMPULSE_ERROR r = run_classifier_continuous(&signal, &result, debug_nn);
    if (r != EI_IMPULSE_OK) {
        ei_printf("ERR: Failed to run classifier (%d)\n", r);
        return;
    }

    if (++print_results >= (EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW)) {
        // print the predictions
        ei_printf("Predictions ");
        ei_printf("(DSP: %d ms., Classification: %d ms., Anomaly: %d ms.)",
            result.timing.dsp, result.timing.classification, result.timing.anomaly);
        ei_printf(": \n");

        float max_confidence = 0;
        String predicted_class = "";

        for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
            ei_printf("    %s: ", result.classification[ix].label);
            ei_printf_float(result.classification[ix].value);
            ei_printf("\n");

            if (result.classification[ix].value > max_confidence) {
                max_confidence = result.classification[ix].value;
                predicted_class = result.classification[ix].label;
            }
        }

        // Show top prediction
        ei_printf("TOP: %s (", predicted_class.c_str());
        ei_printf_float(max_confidence);
        ei_printf(")\n");

        // DEBUG: Show if any keyword has confidence above lower threshold
        for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
            if (String(result.classification[ix].label) != "noise" && result.classification[ix].value > 0.2
                ei_printf("DEBUG: %s detected with confidence ", result.classification[ix].label);
                ei_printf_float(result.classification[ix].value);
                ei_printf("\n");
            }
        }

        // Detection logic with lower threshold for better detection
        if (max_confidence > 0.5 && predicted_class != "noise") {
            ei_printf("\n*** KEYWORD DETECTED ***\n");
            ei_printf("Detected: %s\n", predicted_class.c_str());
            ei_printf("Confidence: ");
            ei_printf_float(max_confidence);
            ei_printf("\n");

            // Set LED state based on detected keyword
            setLEDState(predicted_class);

            if (predicted_class == "activate") {
                ei_printf("-> ACTIVATE command recognized!\n");
                ei_printf("   System is now ACTIVE\n");
            } else if (predicted_class == "warning") {
                ei_printf("-> WARNING command recognized!\n");
                ei_printf("   System is in WARNING mode\n");
            } else if (predicted_class == "stand down") {
                ei_printf("-> STAND DOWN command recognized!\n");
                ei_printf("   System is now in STANDBY\n");
            }
            ei_printf("*************************\n\n");

            // Keep LED on for 3 seconds, then turn off
            delay(3000);
            ei_printf("LEDs turned off - ready for next command\n\n");

        } else if (max_confidence > 0.3 && predicted_class != "noise") {
            // Low confidence detection - brief LED flash to indicate detection attempt
            ei_printf("Low confidence detection: %s (", predicted_class.c_str());
            ei_printf_float(max_confidence);
            ei_printf(")\n");

            // Brief LED flash
            setLEDState(predicted_class);
            delay(200);
            turnOffAllLEDs();
        }

#if EI_CLASSIFIER_HAS_ANOMALY == 1
        ei_printf("    anomaly score: ");
        ei_printf_float(result.anomaly);
        ei_printf("\n");
#endif

        print_results = 0;
    }
}

/**
 * @brief      Read audio data from I2S (polling approach for ESP32 v3.x)
 */
static void read_audio_samples(void)
{
    // Read available samples from I2S
    while (I2S.available() > 0 && inference.buf_count < inference.n_samples) {
        int sample = I2S.read();
        if (sample != -1) {
            // Store the sample in current buffer
            inference.buffers[inference.buf_select][inference.buf_count] = (signed short)(sample &
            inference.buf_count++;

            // If buffer is full, switch to next buffer
            if (inference.buf_count >= inference.n_samples) {
                inference.buf_select ^= 1;
                inference.buf_count = 0;
                inference.buf_ready = 1;
                break;
            }
        }
    }
}

/**
 * @brief      Init inferencing struct and setup/start PDM for ESP32 v3.x
 */
static bool microphone_inference_start(uint32_t n_samples)
{
    inference.buffers[0] = (signed short *)malloc(n_samples * sizeof(signed short));
    if(inference.buffers[0] == NULL) {
        ei_printf("ERROR: Could not allocate buffer 0\n");
        return false;
    }

    inference.buffers[1] = (signed short *)malloc(n_samples * sizeof(signed short));
    if(inference.buffers[1] == NULL) {
        ei_printf("ERROR: Could not allocate buffer 1\n");
        ei_free(inference.buffers[0]);
        return false;
    }

    sampleBuffer = inference.buffers[inference.buf_select];

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

    // ESP32 v3.x I2S setup for XIAO ESP32S3 Sense
    ei_printf("Setting up I2S for ESP32 v3.x...\n");

    // Set PDM pins for XIAO ESP32S3 Sense
    ei_printf("Setting PDM pins (CLK=42, DATA=41)...\n");
    I2S.setPinsPdmRx(42, 41);  // CLK=42, DATA=41

    // Begin I2S in PDM RX mode
    ei_printf("Starting I2S in PDM RX mode...\n");
    if (!I2S.begin(I2S_MODE_PDM_RX, 16000, I2S_DATA_BIT_WIDTH_16BIT, I2S_SLOT_MODE_MONO)) {
        ei_printf("ERROR: Failed to start I2S PDM!\n");
        microphone_inference_end();
        return false;
    }

    record_ready = true;
    ei_printf("✅ I2S PDM microphone started successfully for ESP32 v3.x!\n");
    return true;
}

/**
 * @brief      Wait on new data
 */
static bool microphone_inference_record(void)
{
    bool ret = true;

    if (inference.buf_ready == 1) {
        ei_printf("Error sample buffer overrun. Decrease the number of slices per model window\n");
        ret = false;
    }

    while (inference.buf_ready == 0) {
        // Continuously read audio samples (polling approach)
        read_audio_samples();
        delay(1);
    }

    inference.buf_ready = 0;
    return ret;
}

/**
 * Get raw audio signal data
 */
static int microphone_audio_signal_get_data(size_t offset, size_t length, float *out_ptr)
{
    numpy::int16_to_float(&inference.buffers[inference.buf_select ^ 1][offset], out_ptr, length);
    return 0;
}

/**
 * @brief      Stop PDM and release buffers
 */
static void microphone_inference_end(void)
{
    I2S.end();
    ei_free(inference.buffers[0]);
    ei_free(inference.buffers[1]);
}

#if !defined(EI_CLASSIFIER_SENSOR) || EI_CLASSIFIER_SENSOR != EI_CLASSIFIER_SENSOR_MICROPHONE
#error "Invalid model for current sensor."
#endif
```

Tuning
- #1 — Noise data should be only noise not human sound
- #2 — We need more samples for all class
- #3 — Inclusive of Noise class, for all the class data needed to be balanced
- #4 — Instead of double word Stand down , we can give single word — coz, window is 1s
- #5 — Testing result is not a REAL TIME testing result — so dont judge the model based on test result
- #6 — Try Data Augmentation

Materials - Sanjay Trained Model
- best model, will be uploaded inside your course materials — LATER
- ei-keyword-spotter-ai-arduino-1.0.1.zip — OK
- ei-tinyguy_keywordspotter-arduino-1.0.1.zip — Not Good