Diff Direction

268 lines - 26 Removals

```
/* Edge Impulse Arduino examples
 * Copyright (c) 2021 EdgeImpulse Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */


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

/**
 * Define the number of slices per model window. E.g. a model window of 1000 ms
 * with slices per model window set to 4. Results in a slice size of 250 ms.
 * For more info: https://docs.edgeimpulse.com/docs/continuous-audio-sampling

```

332 lines + 89 Additions

```
/* Edge Impulse Arduino examples
 * Copyright (c) 2021 EdgeImpulse Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
#include <LiquidCrystal_I2C.h>
#include <Wire.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

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

/**
 * Define the number of slices per model window. E.g. a model window of 1000 ms
 * with slices per model window set to 4. Results in a slice size of 250 ms.
 * For more info: https://docs.edgeimpulse.com/docs/continuous-audio-sampling

 */
//#define EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW 3

/*
 ** NOTE: If you run into TFLite arena allocation issue.
 **
 ** This may be due to may dynamic memory fragmentation.
 ** Try defining "-DEI_CLASSIFIER_ALLOCATION_STATIC" in boards.local.txt (create
 ** if it doesn't exist) and copy this file to
 ** `<ARDUINO_CORE_INSTALL_PATH>/arduino/hardware/<mbed_core>/<core_version>/`.
 **
 ** See
 ** (https://support.arduino.cc/hc/en-us/articles/360012076960-Where-are-the-installed-cores-located-)
 ** to find where Arduino installs cores on your machine.
 **
 ** If the problem persists then there's not enough memory for this model and application.
 */

/* Includes ---------------------------------------------------------------- */
#include <PDM.h>
#include <FYP_Direction_inferencing.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 volatile bool record_ready = false;
// static signed short *sampleBuffer;
static signed short sampleBuffer[2048];
static bool debug_nn = false; // Set this to true to see e.g. features generated from the raw signal
static int print_results = -(EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW);

/**
 * @brief      Arduino setup function
 */
```

```
 */
//#define EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW 3

/*
 ** NOTE: If you run into TFLite arena allocation issue.
 **
 ** This may be due to may dynamic memory fragmentation.
 ** Try defining "-DEI_CLASSIFIER_ALLOCATION_STATIC" in boards.local.txt (create
 ** if it doesn't exist) and copy this file to
 ** `<ARDUINO_CORE_INSTALL_PATH>/arduino/hardware/<mbed_core>/<core_version>/`.
 **
 ** See
 ** (https://support.arduino.cc/hc/en-us/articles/360012076960-Where-are-the-installed-cores-located-)
 ** to find where Arduino installs cores on your machine.
 **
 ** If the problem persists then there's not enough memory for this model and application.
 */

/* Includes ---------------------------------------------------------------- */
#include <FYP_Direction_inferencing.h>

#include "RPC.h"

/* Mic Setting */
int sound_analog = A6;

  int Pass;
  int outtoM4 = D5;

  int setVar(int a) {
    Pass = (int)a;
    return a;
  }


/** 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 volatile bool record_ready = false;
//static signed short *sampleBuffer;
static signed short sampleBuffer[2048];
static bool debug_nn = false; // Set this to true to see e.g. features generated from the raw signal
static int print_results = -(EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW);

/**
 * @brief      Arduino setup function
 */
```

```
void setup()
{
    // put your setup code here, to run once:
    Serial.begin(115200);

    Serial.println("Edge Impulse Inferencing Demo");
    // summary of inferencing settings (from model_metadata.h)
    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]));

    run_classifier_init();
    if (microphone_inference_start(EI_CLASSIFIER_SLICE_SIZE) == false) {
        ei_printf("ERR: Failed to setup audio sampling\r\n");
        return;
    }
}

/**
 * @brief      Arduino main function. Runs the inferencing loop.
 */
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(&sig
```

```
void setup()
{
    // put your setup code here, to run once:
    LL_RCC_ForceCM4Boot();
    RPC.begin();
    Serial.begin(115200);
    pinMode(outtoM4, OUTPUT);
    digitalWrite(outtoM4, LOW);
     RPC.bind("setVar", setVar); // do these have to be the same?
     Pass = 0;

    Serial.println("Edge Impulse Inferencing Demo");
    lcd.begin();
    lcd.backlight();
    // summary of inferencing settings (from model_metadata.h)
    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]));

    run_classifier_init();
    if (microphone_inference_start(EI_CLASSIFIER_SLICE_SIZE) == false) {
        ei_printf("ERR: Failed to setup audio sampling\r\n");
        return;
    }
}

/**
 * @brief      Arduino main function. Runs the inferencing loop.
 */
void loop()
{ 
  if (Pass == 1){
    lcd.setCursor(0, 0);
    lcd.print("Stage: 2         ");
    lcd.setCursor(0, 1);
    lcd.print("Wait for Command");
    digitalWrite(outtoM4, HIGH);
    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(&sig
nal, &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");
        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 EI_CLASSIFIER_HAS_ANOMALY == 1
        ei_printf("    anomaly score: ");

```

```
nal, &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");
        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");
        }

        for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
            if ( (ix != 2) && (result.classification[ix].value >=0.5)){
              Pass=0;
              digitalWrite(outtoM4, LOW);
              digitalWrite(outtoM4, 0);
              switch (ix) {
                case 0:
                  lcd.setCursor(0, 1);
                  lcd.print("Down            ");
                  break;

                case 1:
                  lcd.setCursor(0, 1);
                  lcd.print("Left            ");
                  break;

                  case 3:
                  lcd.setCursor(0, 1);
                  lcd.print("Right           ");
                  break;

                  case 4:
                  lcd.setCursor(0, 1);
                  lcd.print("Up              ");
                  break;

                default:
                break;
                }
        Serial.println("Transforming to M4...");
        delay(100);
            }
        }

#if EI_CLASSIFIER_HAS_ANOMALY == 1
        ei_printf("    anomaly score: ");

        ei_printf_float(result.anomaly);
        ei_printf("\n");
#endif

        print_results = 0;
    }
}

/**
 * @brief      PDM buffer full callback
 *             Copy audio data to app buffers
 */
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.buffers[inference.buf_select][inference.buf_count++] = sampleBuffer[i];

            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
 *
 * @param[in]  n_samples  The n samples
 *
 * @return     { description_of_the_return_value }
 */
static bool microphone_inference_start(uint32_t n_samples)
{
    inference.buffers[0] = (signed short *)malloc(n_sam
```

```
        ei_printf_float(result.anomaly);
        ei_printf("\n");
#endif

        print_results = 0;
    }
  }
  else{

      digitalWrite(outtoM4, LOW);
      lcd.setCursor(0, 0);
      lcd.print("Stage: 1        ");
     while (RPC.available()) {
       Serial.write(RPC.read()); // check if the M4 has sent an RPC println
     }

  }
}

/**
 * @brief      PDM buffer full callback
 *             Copy audio data to app buffers
 */
static void pdm_data_ready_inference_callback(void)
{
    for (int i=0; i<2048; i++){

  sampleBuffer[i] = analogRead(sound_analog);


  }

  size_t bytesRead;
  bytesRead = 2048;

    if ((inference.buf_ready == 0) && (record_ready == true)) {
        for(int i = 0; i < bytesRead; i++) {
            inference.buffers[inference.buf_select][inference.buf_count++] = sampleBuffer[i];

            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
 *
 * @param[in]  n_samples  The n samples
 *
 * @return     { description_of_the_return_value }
 */
static bool microphone_inference_start(uint32_t n_samples)
{
    inference.buffers[0] = (signed short *)malloc(n_sam
ples * sizeof(signed short));

    if (inference.buffers[0] == NULL) {
        return false;
    }

    inference.buffers[1] = (signed short *)malloc(n_samples * sizeof(signed short));

```

```
ples * sizeof(signed short));

    if (inference.buffers[0] == NULL) {
        return false;
    }

    inference.buffers[1] = (signed short *)malloc(n_samples * sizeof(signed short));

```