**Edge Computing Laboratory**

**Lab Assignment 9**

**Name:** Gaurav Gadekar

**Class:** TY AIEC Batch C

**Enrollment No:** MITU23BTCSD120

**Roll No:** D2233120

### **Title: The Object Detection using Camera on Edge Computing Devices**

### **Objective:** Build a project to detect an object using Edge Computing **Tasks:**

* Generate the dataset for customized object
* Configure Edge Impulse for Object Detection
* Building and Training a Model
* Deploy on Edge Computing Device

### **Introduction**

Edge Impulse is a development platform for machine learning on edge devices, targeted at developers who want to create intelligent device solutions. The " Camera "sensor reading equivalent in Edge Impulse would typically involve creating a simple machine learning model that can run on an edge device, like classifying sensor data or recognizing a basic pattern.

**Materials Required**

• Nano BLE Sense Board

### Theory

GPIO (General Purpose Input/Output) pins on the Raspberry Pi are used for interfacing with other electronic components. BCM numbering refers to the pin numbers in the Broadcom SOC channel, which is a more consistent way to refer to the GPIO pins across different versions of the

Here’s a high-level overview of steps you'd follow to create a "Hello World" project on Edge Impulse:

**Steps to Configure the Edge Impulse:**

1. Create an Account and New Project:
   * Sign up for an Edge Impulse account.
   * Create a new project from the dashboard.
2. Connect a Device:
   * You can use a supported development board or your smartphone as a sensor device.
   * Follow the instructions to connect your device to your Edge Impulse project.
3. Collect Data:
   * Use the Edge Impulse mobile app or the Web interface to collect data from the onboard sensors.
   * For a "Hello World" project, you could collect accelerometer data, for instance.
4. Create an Impulse:

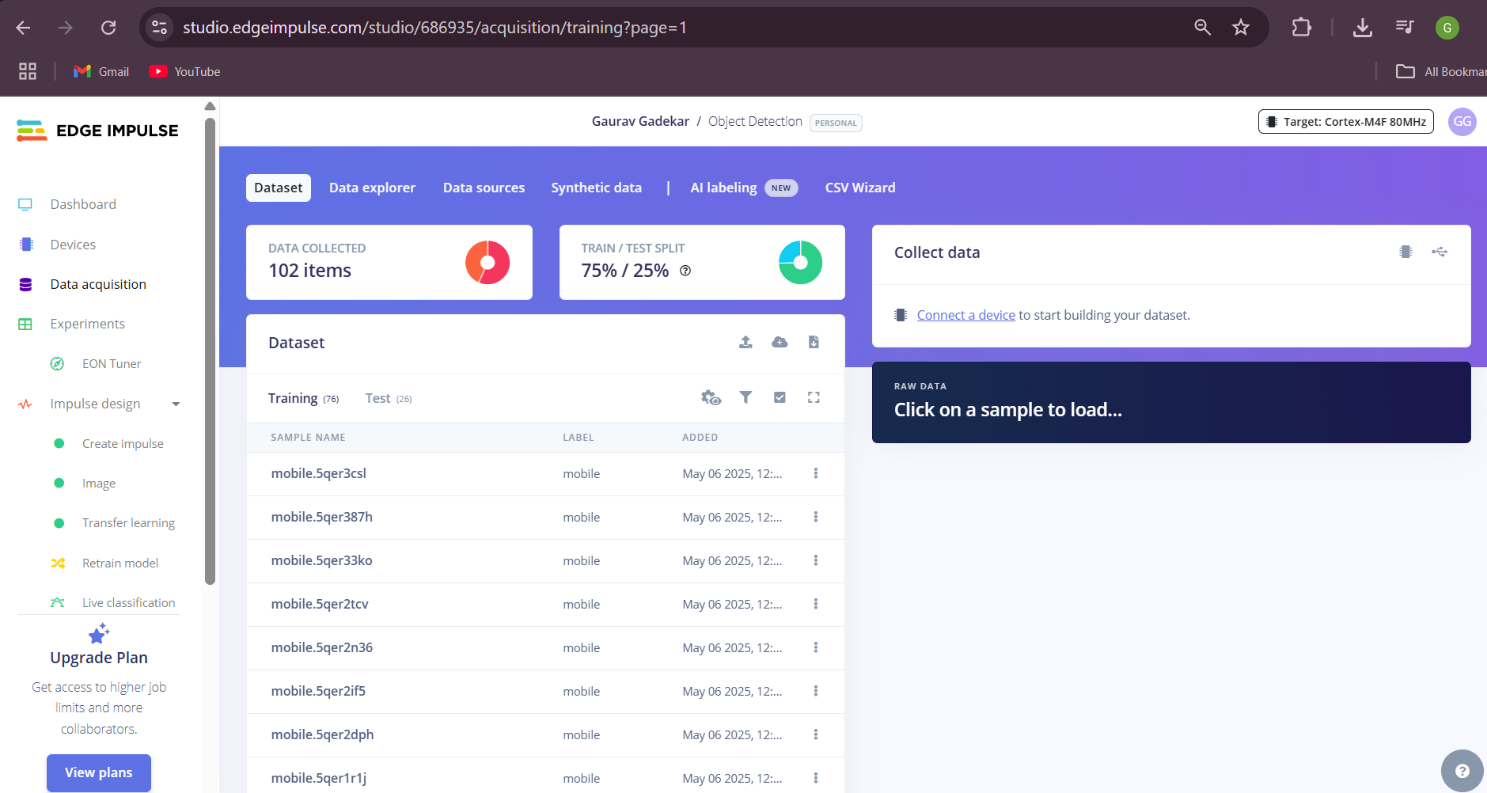
Go to the 'Create impulse' page.

* + Add a processing block (e.g., time-series data) and a learning block (e.g., classification).
  + Save the impulse, which defines the machine learning pipeline.

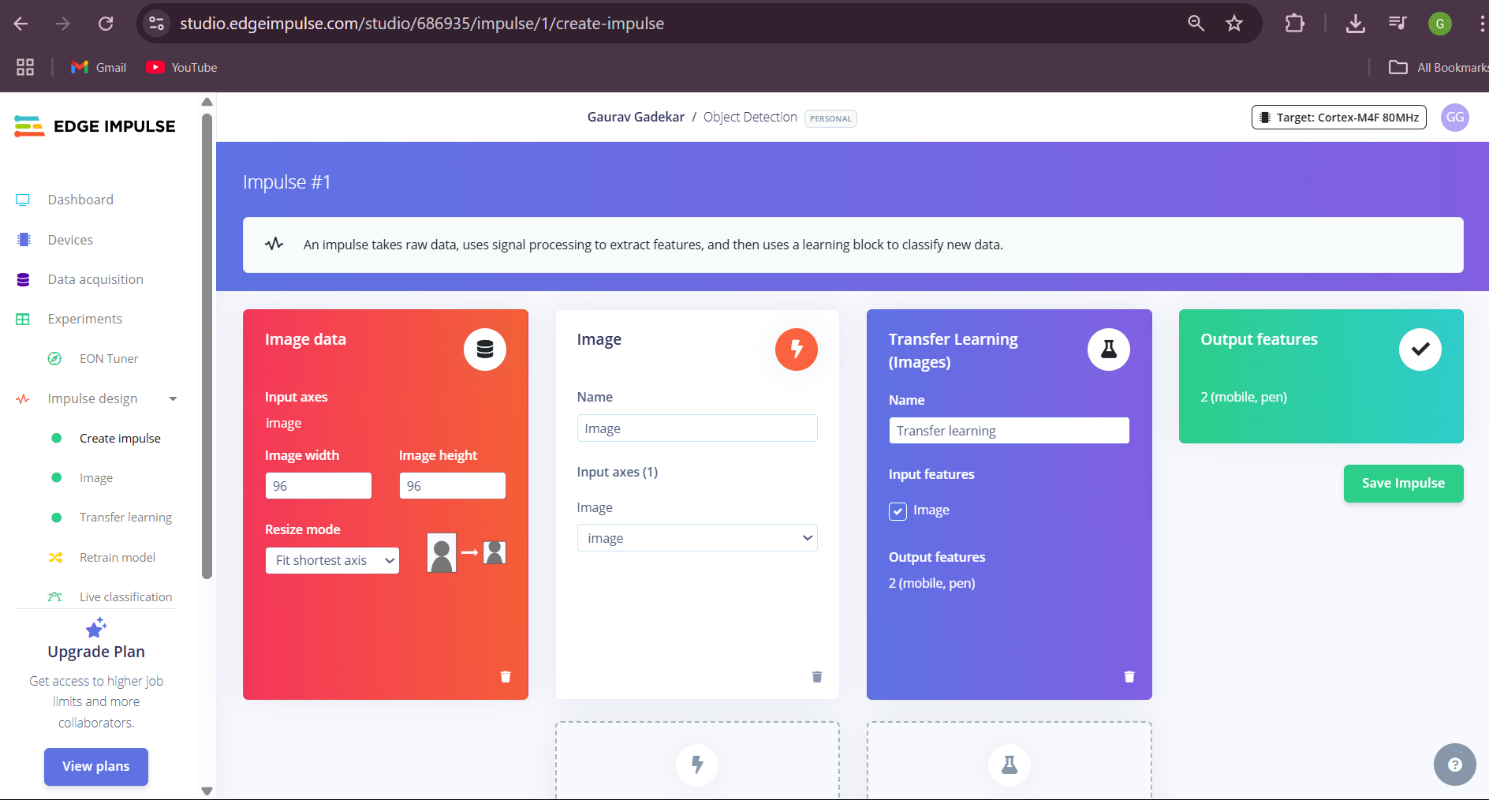
1. Design a Neural Network:
   * Navigate to the 'NN Classifier' under the 'Learning blocks'.
   * Design a simple neural network. Edge Impulse provides a default architecture that works well for most basic tasks.
2. Train the Model:
   * Click on the 'Start training' button to train your machine learning model with the collected data.
3. Test the Model:
   * Once the model is trained, you can test its performance with new data in the 'Model Testing' tab.
4. Deploy the Model:
   * Go to the 'Deployment' tab.
   * Select the deployment method that suits your edge device (e.g., Arduino library, WebAssembly, container, etc.).
   * Follow the instructions to deploy the model to your device.
5. Run Inference:
   * With the model deployed, run inference on the edge device to see it classifying data in real-time.
6. Monitor:
   * You can monitor the performance of your device through the Edge Impulse studio.

**Paste your Edge Impulse project’s Results:**

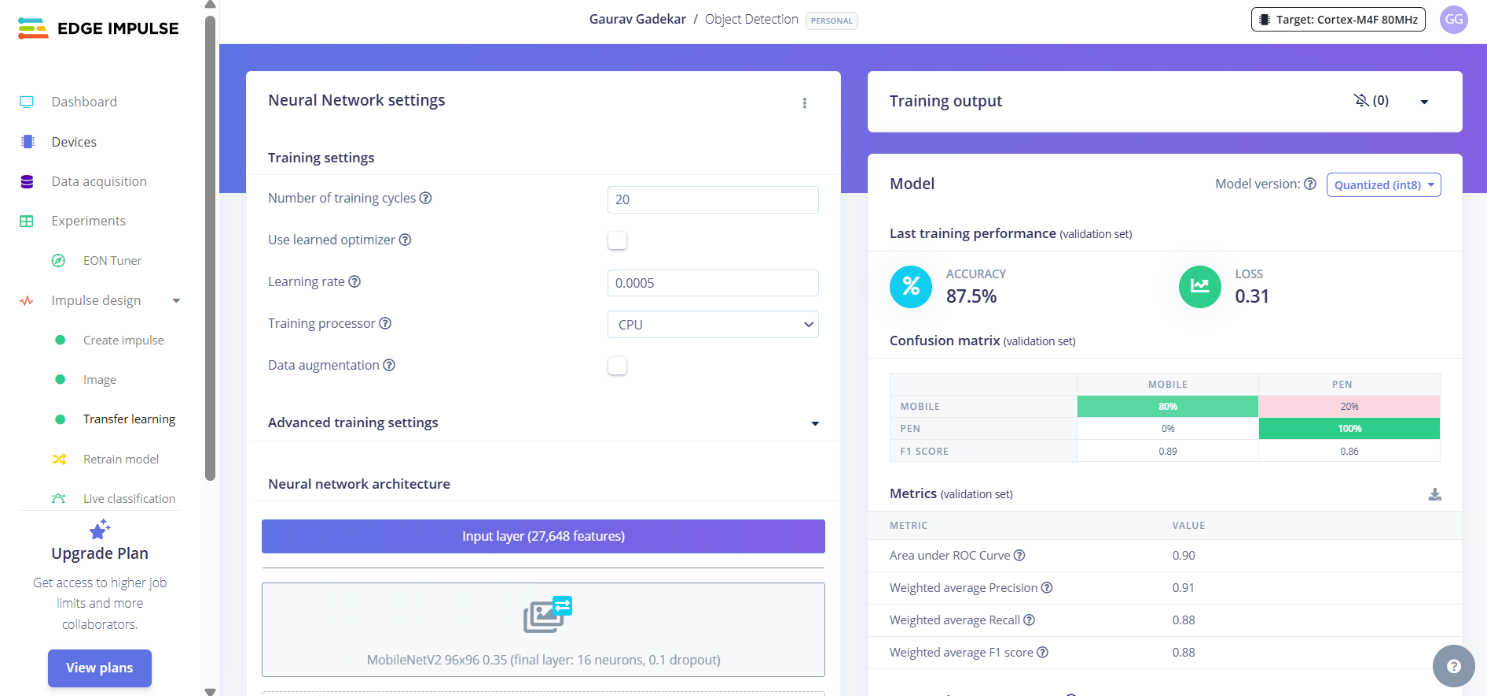
1. **Dataset Image**

****

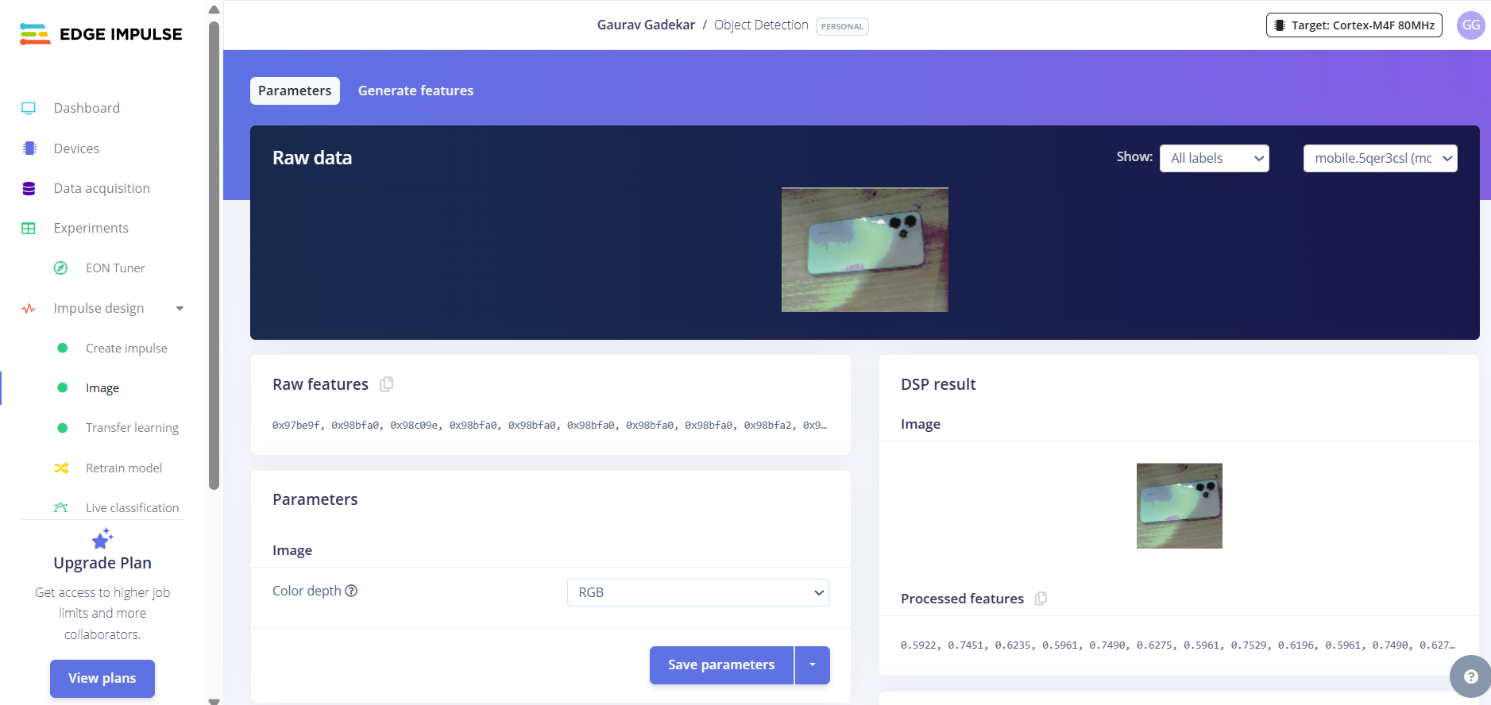
1. **Feature extraction - Image**

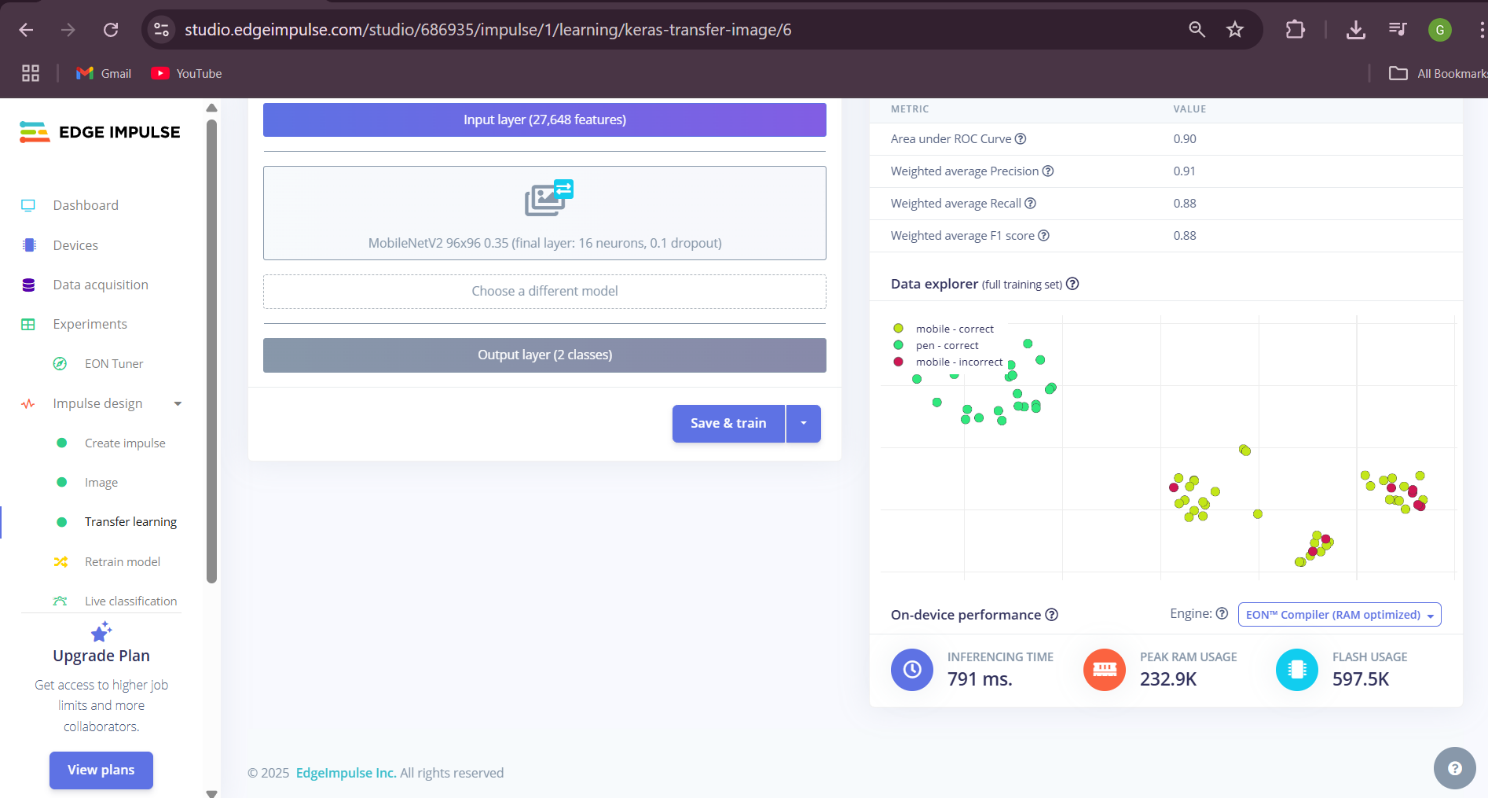
****

1. **Accuracy / Loss - Confusion Matrix – image**

****

1. **Validation Result – Image**





1. **Copy the code of Arduino Sketch**

/\* Edge Impulse ingestion SDK

\* Copyright (c) 2022 EdgeImpulse Inc.

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*

\*/

/\* Includes ---------------------------------------------------------------- \*/

#include <Object\_Detection\_inferencing.h>

#include <Arduino\_OV767X.h> //Click here to get the library: https://www.arduino.cc/reference/en/libraries/arduino\_ov767x/

#include <stdint.h>

#include <stdlib.h>

/\* Constant variables ------------------------------------------------------- \*/

#define EI\_CAMERA\_RAW\_FRAME\_BUFFER\_COLS 160

#define EI\_CAMERA\_RAW\_FRAME\_BUFFER\_ROWS 120

#define DWORD\_ALIGN\_PTR(a) ((a & 0x3) ?(((uintptr\_t)a + 0x4) & ~(uintptr\_t)0x3) : a)

/\*

\*\* 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.

\*/

/\* Edge Impulse ------------------------------------------------------------- \*/

class OV7675 : public OV767X {

public:

int begin(int resolution, int format, int fps);

void readFrame(void\* buffer);

private:

int vsyncPin;

int hrefPin;

int pclkPin;

int xclkPin;

volatile uint32\_t\* vsyncPort;

uint32\_t vsyncMask;

volatile uint32\_t\* hrefPort;

uint32\_t hrefMask;

volatile uint32\_t\* pclkPort;

uint32\_t pclkMask;

uint16\_t width;

uint16\_t height;

uint8\_t bytes\_per\_pixel;

uint16\_t bytes\_per\_row;

uint8\_t buf\_rows;

uint16\_t buf\_size;

uint8\_t resize\_height;

uint8\_t \*raw\_buf;

void \*buf\_mem;

uint8\_t \*intrp\_buf;

uint8\_t \*buf\_limit;

void readBuf();

int allocate\_scratch\_buffs();

int deallocate\_scratch\_buffs();

};

typedef struct {

size\_t width;

size\_t height;

} ei\_device\_resize\_resolutions\_t;

/\*\*

\* @brief Check if new serial data is available

\*

\* @return Returns number of available bytes

\*/

int ei\_get\_serial\_available(void) {

return Serial.available();

}

/\*\*

\* @brief Get next available byte

\*

\* @return byte

\*/

char ei\_get\_serial\_byte(void) {

return Serial.read();

}

/\* Private variables ------------------------------------------------------- \*/

static OV7675 Cam;

static bool is\_initialised = false;

/\*

\*\* @brief points to the output of the capture

\*/

static uint8\_t \*ei\_camera\_capture\_out = NULL;

uint32\_t resize\_col\_sz;

uint32\_t resize\_row\_sz;

bool do\_resize = false;

bool do\_crop = false;

static bool debug\_nn = false; // Set this to true to see e.g. features generated from the raw signal

/\* Function definitions ------------------------------------------------------- \*/

bool ei\_camera\_init(void);

void ei\_camera\_deinit(void);

bool ei\_camera\_capture(uint32\_t img\_width, uint32\_t img\_height, uint8\_t \*out\_buf) ;

int calculate\_resize\_dimensions(uint32\_t out\_width, uint32\_t out\_height, uint32\_t \*resize\_col\_sz, uint32\_t \*resize\_row\_sz, bool \*do\_resize);

void resizeImage(int srcWidth, int srcHeight, uint8\_t \*srcImage, int dstWidth, int dstHeight, uint8\_t \*dstImage, int iBpp);

void cropImage(int srcWidth, int srcHeight, uint8\_t \*srcImage, int startX, int startY, int dstWidth, int dstHeight, uint8\_t \*dstImage, int iBpp);

/\*\*

\* @brief Arduino setup function

\*/

void setup()

{

// put your setup code here, to run once:

Serial.begin(115200);

// comment out the below line to cancel the wait for USB connection (needed for native USB)

while (!Serial);

Serial.println("Edge Impulse Inferencing Demo");

// summary of inferencing settings (from model\_metadata.h)

ei\_printf("Inferencing settings:\n");

ei\_printf("\tImage resolution: %dx%d\n", EI\_CLASSIFIER\_INPUT\_WIDTH, EI\_CLASSIFIER\_INPUT\_HEIGHT);

ei\_printf("\tFrame size: %d\n", EI\_CLASSIFIER\_DSP\_INPUT\_FRAME\_SIZE);

ei\_printf("\tNo. of classes: %d\n", sizeof(ei\_classifier\_inferencing\_categories) / sizeof(ei\_classifier\_inferencing\_categories[0]));

}

/\*\*

\* @brief Get data and run inferencing

\*

\* @param[in] debug Get debug info if true

\*/

void loop()

{

bool stop\_inferencing = false;

while(stop\_inferencing == false) {

ei\_printf("\nStarting inferencing in 2 seconds...\n");

// instead of wait\_ms, we'll wait on the signal, this allows threads to cancel us...

if (ei\_sleep(2000) != EI\_IMPULSE\_OK) {

break;

}

ei\_printf("Taking photo...\n");

if (ei\_camera\_init() == false) {

ei\_printf("ERR: Failed to initialize image sensor\r\n");

break;

}

// choose resize dimensions

uint32\_t resize\_col\_sz;

uint32\_t resize\_row\_sz;

bool do\_resize = false;

int res = calculate\_resize\_dimensions(EI\_CLASSIFIER\_INPUT\_WIDTH, EI\_CLASSIFIER\_INPUT\_HEIGHT, &resize\_col\_sz, &resize\_row\_sz, &do\_resize);

if (res) {

ei\_printf("ERR: Failed to calculate resize dimensions (%d)\r\n", res);

break;

}

void \*snapshot\_mem = NULL;

uint8\_t \*snapshot\_buf = NULL;

snapshot\_mem = ei\_malloc(resize\_col\_sz\*resize\_row\_sz\*2);

if(snapshot\_mem == NULL) {

ei\_printf("failed to create snapshot\_mem\r\n");

break;

}

snapshot\_buf = (uint8\_t \*)DWORD\_ALIGN\_PTR((uintptr\_t)snapshot\_mem);

if (ei\_camera\_capture(EI\_CLASSIFIER\_INPUT\_WIDTH, EI\_CLASSIFIER\_INPUT\_HEIGHT, snapshot\_buf) == false) {

ei\_printf("Failed to capture image\r\n");

if (snapshot\_mem) ei\_free(snapshot\_mem);

break;

}

ei::signal\_t signal;

signal.total\_length = EI\_CLASSIFIER\_INPUT\_WIDTH \* EI\_CLASSIFIER\_INPUT\_HEIGHT;

signal.get\_data = &ei\_camera\_cutout\_get\_data;

// run the impulse: DSP, neural network and the Anomaly algorithm

ei\_impulse\_result\_t result = { 0 };

EI\_IMPULSE\_ERROR ei\_error = run\_classifier(&signal, &result, debug\_nn);

if (ei\_error != EI\_IMPULSE\_OK) {

ei\_printf("Failed to run impulse (%d)\n", ei\_error);

ei\_free(snapshot\_mem);

break;

}

// print the predictions

ei\_printf("Predictions (DSP: %d ms., Classification: %d ms., Anomaly: %d ms.): \n",

result.timing.dsp, result.timing.classification, result.timing.anomaly);

#if EI\_CLASSIFIER\_OBJECT\_DETECTION == 1

ei\_printf("Object detection bounding boxes:\r\n");

for (uint32\_t i = 0; i < result.bounding\_boxes\_count; i++) {

ei\_impulse\_result\_bounding\_box\_t bb = result.bounding\_boxes[i];

if (bb.value == 0) {

continue;

}

ei\_printf(" %s (%f) [ x: %u, y: %u, width: %u, height: %u ]\r\n",

bb.label,

bb.value,

bb.x,

bb.y,

bb.width,

bb.height);

}

// Print the prediction results (classification)

#else

ei\_printf("Predictions:\r\n");

for (uint16\_t i = 0; i < EI\_CLASSIFIER\_LABEL\_COUNT; i++) {

ei\_printf(" %s: ", ei\_classifier\_inferencing\_categories[i]);

ei\_printf("%.5f\r\n", result.classification[i].value);

}

#endif

// Print anomaly result (if it exists)

#if EI\_CLASSIFIER\_HAS\_ANOMALY

ei\_printf("Anomaly prediction: %.3f\r\n", result.anomaly);

#endif

#if EI\_CLASSIFIER\_HAS\_VISUAL\_ANOMALY

ei\_printf("Visual anomalies:\r\n");

for (uint32\_t i = 0; i < result.visual\_ad\_count; i++) {

ei\_impulse\_result\_bounding\_box\_t bb = result.visual\_ad\_grid\_cells[i];

if (bb.value == 0) {

continue;

}

ei\_printf(" %s (%f) [ x: %u, y: %u, width: %u, height: %u ]\r\n",

bb.label,

bb.value,

bb.x,

bb.y,

bb.width,

bb.height);

}

#endif

while (ei\_get\_serial\_available() > 0) {

if (ei\_get\_serial\_byte() == 'b') {

ei\_printf("Inferencing stopped by user\r\n");

stop\_inferencing = true;

}

}

if (snapshot\_mem) ei\_free(snapshot\_mem);

}

ei\_camera\_deinit();

}

/\*\*

\* @brief Determine whether to resize and to which dimension

\*

\* @param[in] out\_width width of output image

\* @param[in] out\_height height of output image

\* @param[out] resize\_col\_sz pointer to frame buffer's column/width value

\* @param[out] resize\_row\_sz pointer to frame buffer's rows/height value

\* @param[out] do\_resize returns whether to resize (or not)

\*

\*/

int calculate\_resize\_dimensions(uint32\_t out\_width, uint32\_t out\_height, uint32\_t \*resize\_col\_sz, uint32\_t \*resize\_row\_sz, bool \*do\_resize)

{

size\_t list\_size = 2;

const ei\_device\_resize\_resolutions\_t list[list\_size] = { {42,32}, {128,96} };

// (default) conditions

\*resize\_col\_sz = EI\_CAMERA\_RAW\_FRAME\_BUFFER\_COLS;

\*resize\_row\_sz = EI\_CAMERA\_RAW\_FRAME\_BUFFER\_ROWS;

\*do\_resize = false;

for (size\_t ix = 0; ix < list\_size; ix++) {

if ((out\_width <= list[ix].width) && (out\_height <= list[ix].height)) {

\*resize\_col\_sz = list[ix].width;

\*resize\_row\_sz = list[ix].height;

\*do\_resize = true;

break;

}

}

return 0;

}

/\*\*

\* @brief Setup image sensor & start streaming

\*

\* @retval false if initialisation failed

\*/

bool ei\_camera\_init(void) {

if (is\_initialised) return true;

if (!Cam.begin(QQVGA, RGB565, 1)) { // VGA downsampled to QQVGA (OV7675)

ei\_printf("ERR: Failed to initialize camera\r\n");

return false;

}

is\_initialised = true;

return true;

}

/\*\*

\* @brief Stop streaming of sensor data

\*/

void ei\_camera\_deinit(void) {

if (is\_initialised) {

Cam.end();

is\_initialised = false;

}

}

/\*\*

\* @brief Capture, rescale and crop image

\*

\* @param[in] img\_width width of output image

\* @param[in] img\_height height of output image

\* @param[in] out\_buf pointer to store output image, NULL may be used

\* when full resolution is expected.

\*

\* @retval false if not initialised, image captured, rescaled or cropped failed

\*

\*/

bool ei\_camera\_capture(uint32\_t img\_width, uint32\_t img\_height, uint8\_t \*out\_buf)

{

if (!is\_initialised) {

ei\_printf("ERR: Camera is not initialized\r\n");

return false;

}

if (!out\_buf) {

ei\_printf("ERR: invalid parameters\r\n");

return false;

}

// choose resize dimensions

int res = calculate\_resize\_dimensions(img\_width, img\_height, &resize\_col\_sz, &resize\_row\_sz, &do\_resize);

if (res) {

ei\_printf("ERR: Failed to calculate resize dimensions (%d)\r\n", res);

return false;

}

if ((img\_width != resize\_col\_sz)

|| (img\_height != resize\_row\_sz)) {

do\_crop = true;

}

Cam.readFrame(out\_buf); // captures image and resizes

if (do\_crop) {

uint32\_t crop\_col\_sz;

uint32\_t crop\_row\_sz;

uint32\_t crop\_col\_start;

uint32\_t crop\_row\_start;

crop\_row\_start = (resize\_row\_sz - img\_height) / 2;

crop\_col\_start = (resize\_col\_sz - img\_width) / 2;

crop\_col\_sz = img\_width;

crop\_row\_sz = img\_height;

//ei\_printf("crop cols: %d, rows: %d\r\n", crop\_col\_sz,crop\_row\_sz);

cropImage(resize\_col\_sz, resize\_row\_sz,

out\_buf,

crop\_col\_start, crop\_row\_start,

crop\_col\_sz, crop\_row\_sz,

out\_buf,

16);

}

// The following variables should always be assigned

// if this routine is to return true

// cutout values

//ei\_camera\_snapshot\_is\_resized = do\_resize;

//ei\_camera\_snapshot\_is\_cropped = do\_crop;

ei\_camera\_capture\_out = out\_buf;

return true;

}

/\*\*

\* @brief Convert RGB565 raw camera buffer to RGB888

\*

\* @param[in] offset pixel offset of raw buffer

\* @param[in] length number of pixels to convert

\* @param[out] out\_buf pointer to store output image

\*/

int ei\_camera\_cutout\_get\_data(size\_t offset, size\_t length, float \*out\_ptr) {

size\_t pixel\_ix = offset \* 2;

size\_t bytes\_left = length;

size\_t out\_ptr\_ix = 0;

// read byte for byte

while (bytes\_left != 0) {

// grab the value and convert to r/g/b

uint16\_t pixel = (ei\_camera\_capture\_out[pixel\_ix] << 8) | ei\_camera\_capture\_out[pixel\_ix+1];

uint8\_t r, g, b;

r = ((pixel >> 11) & 0x1f) << 3;

g = ((pixel >> 5) & 0x3f) << 2;

b = (pixel & 0x1f) << 3;

// then convert to out\_ptr format

float pixel\_f = (r << 16) + (g << 8) + b;

out\_ptr[out\_ptr\_ix] = pixel\_f;

// and go to the next pixel

out\_ptr\_ix++;

pixel\_ix+=2;

bytes\_left--;

}

// and done!

return 0;

}

// This include file works in the Arduino environment

// to define the Cortex-M intrinsics

#ifdef \_\_ARM\_FEATURE\_SIMD32

#include <device.h>

#endif

// This needs to be < 16 or it won't fit. Cortex-M4 only has SIMD for signed multiplies

#define FRAC\_BITS 14

#define FRAC\_VAL (1<<FRAC\_BITS)

#define FRAC\_MASK (FRAC\_VAL - 1)

//

// Resize

//

// Assumes that the destination buffer is dword-aligned

// Can be used to resize the image smaller or larger

// If resizing much smaller than 1/3 size, then a more rubust algorithm should average all of the pixels

// This algorithm uses bilinear interpolation - averages a 2x2 region to generate each new pixel

//

// Optimized for 32-bit MCUs

// supports 8 and 16-bit pixels

void resizeImage(int srcWidth, int srcHeight, uint8\_t \*srcImage, int dstWidth, int dstHeight, uint8\_t \*dstImage, int iBpp)

{

uint32\_t src\_x\_accum, src\_y\_accum; // accumulators and fractions for scaling the image

uint32\_t x\_frac, nx\_frac, y\_frac, ny\_frac;

int x, y, ty, tx;

if (iBpp != 8 && iBpp != 16)

return;

src\_y\_accum = FRAC\_VAL/2; // start at 1/2 pixel in to account for integer downsampling which might miss pixels

const uint32\_t src\_x\_frac = (srcWidth \* FRAC\_VAL) / dstWidth;

const uint32\_t src\_y\_frac = (srcHeight \* FRAC\_VAL) / dstHeight;

const uint32\_t r\_mask = 0xf800f800;

const uint32\_t g\_mask = 0x07e007e0;

const uint32\_t b\_mask = 0x001f001f;

uint8\_t \*s, \*d;

uint16\_t \*s16, \*d16;

uint32\_t x\_frac2, y\_frac2; // for 16-bit SIMD

for (y=0; y < dstHeight; y++) {

ty = src\_y\_accum >> FRAC\_BITS; // src y

y\_frac = src\_y\_accum & FRAC\_MASK;

src\_y\_accum += src\_y\_frac;

ny\_frac = FRAC\_VAL - y\_frac; // y fraction and 1.0 - y fraction

y\_frac2 = ny\_frac | (y\_frac << 16); // for M4/M4 SIMD

s = &srcImage[ty \* srcWidth];

s16 = (uint16\_t \*)&srcImage[ty \* srcWidth \* 2];

d = &dstImage[y \* dstWidth];

d16 = (uint16\_t \*)&dstImage[y \* dstWidth \* 2];

src\_x\_accum = FRAC\_VAL/2; // start at 1/2 pixel in to account for integer downsampling which might miss pixels

if (iBpp == 8) {

for (x=0; x < dstWidth; x++) {

uint32\_t tx, p00,p01,p10,p11;

tx = src\_x\_accum >> FRAC\_BITS;

x\_frac = src\_x\_accum & FRAC\_MASK;

nx\_frac = FRAC\_VAL - x\_frac; // x fraction and 1.0 - x fraction

x\_frac2 = nx\_frac | (x\_frac << 16);

src\_x\_accum += src\_x\_frac;

p00 = s[tx]; p10 = s[tx+1];

p01 = s[tx+srcWidth]; p11 = s[tx+srcWidth+1];

#ifdef \_\_ARM\_FEATURE\_SIMD32

p00 = \_\_SMLAD(p00 | (p10<<16), x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // top line

p01 = \_\_SMLAD(p01 | (p11<<16), x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

p00 = \_\_SMLAD(p00 | (p01<<16), y\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // combine

#else // generic C code

p00 = ((p00 \* nx\_frac) + (p10 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // top line

p01 = ((p01 \* nx\_frac) + (p11 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

p00 = ((p00 \* ny\_frac) + (p01 \* y\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // combine top + bottom

#endif // Cortex-M4/M7

\*d++ = (uint8\_t)p00; // store new pixel

} // for x

} // 8-bpp

else

{ // RGB565

for (x=0; x < dstWidth; x++) {

uint32\_t tx, p00,p01,p10,p11;

uint32\_t r00, r01, r10, r11, g00, g01, g10, g11, b00, b01, b10, b11;

tx = src\_x\_accum >> FRAC\_BITS;

x\_frac = src\_x\_accum & FRAC\_MASK;

nx\_frac = FRAC\_VAL - x\_frac; // x fraction and 1.0 - x fraction

x\_frac2 = nx\_frac | (x\_frac << 16);

src\_x\_accum += src\_x\_frac;

p00 = \_\_builtin\_bswap16(s16[tx]); p10 = \_\_builtin\_bswap16(s16[tx+1]);

p01 = \_\_builtin\_bswap16(s16[tx+srcWidth]); p11 = \_\_builtin\_bswap16(s16[tx+srcWidth+1]);

#ifdef \_\_ARM\_FEATURE\_SIMD32

{

p00 |= (p10 << 16);

p01 |= (p11 << 16);

r00 = (p00 & r\_mask) >> 1; g00 = p00 & g\_mask; b00 = p00 & b\_mask;

r01 = (p01 & r\_mask) >> 1; g01 = p01 & g\_mask; b01 = p01 & b\_mask;

r00 = \_\_SMLAD(r00, x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // top line

r01 = \_\_SMLAD(r01, x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

r00 = \_\_SMLAD(r00 | (r01<<16), y\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // combine

g00 = \_\_SMLAD(g00, x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // top line

g01 = \_\_SMLAD(g01, x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

g00 = \_\_SMLAD(g00 | (g01<<16), y\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // combine

b00 = \_\_SMLAD(b00, x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // top line

b01 = \_\_SMLAD(b01, x\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

b00 = \_\_SMLAD(b00 | (b01<<16), y\_frac2, FRAC\_VAL/2) >> FRAC\_BITS; // combine

}

#else // generic C code

{

r00 = (p00 & r\_mask) >> 1; g00 = p00 & g\_mask; b00 = p00 & b\_mask;

r10 = (p10 & r\_mask) >> 1; g10 = p10 & g\_mask; b10 = p10 & b\_mask;

r01 = (p01 & r\_mask) >> 1; g01 = p01 & g\_mask; b01 = p01 & b\_mask;

r11 = (p11 & r\_mask) >> 1; g11 = p11 & g\_mask; b11 = p11 & b\_mask;

r00 = ((r00 \* nx\_frac) + (r10 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // top line

r01 = ((r01 \* nx\_frac) + (r11 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

r00 = ((r00 \* ny\_frac) + (r01 \* y\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // combine top + bottom

g00 = ((g00 \* nx\_frac) + (g10 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // top line

g01 = ((g01 \* nx\_frac) + (g11 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

g00 = ((g00 \* ny\_frac) + (g01 \* y\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // combine top + bottom

b00 = ((b00 \* nx\_frac) + (b10 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // top line

b01 = ((b01 \* nx\_frac) + (b11 \* x\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // bottom line

b00 = ((b00 \* ny\_frac) + (b01 \* y\_frac) + FRAC\_VAL/2) >> FRAC\_BITS; // combine top + bottom

}

#endif // Cortex-M4/M7

r00 = (r00 << 1) & r\_mask;

g00 = g00 & g\_mask;

b00 = b00 & b\_mask;

p00 = (r00 | g00 | b00); // re-combine color components

\*d16++ = (uint16\_t)\_\_builtin\_bswap16(p00); // store new pixel

} // for x

} // 16-bpp

} // for y

} /\* resizeImage() \*/

//

// Crop

//

// Assumes that the destination buffer is dword-aligned

// optimized for 32-bit MCUs

// Supports 8 and 16-bit pixels

//

void cropImage(int srcWidth, int srcHeight, uint8\_t \*srcImage, int startX, int startY, int dstWidth, int dstHeight, uint8\_t \*dstImage, int iBpp)

{

uint32\_t \*s32, \*d32;

int x, y;

if (startX < 0 || startX >= srcWidth || startY < 0 || startY >= srcHeight || (startX + dstWidth) > srcWidth || (startY + dstHeight) > srcHeight)

return; // invalid parameters

if (iBpp != 8 && iBpp != 16)

return;

if (iBpp == 8) {

uint8\_t \*s, \*d;

for (y=0; y<dstHeight; y++) {

s = &srcImage[srcWidth \* (y + startY) + startX];

d = &dstImage[(dstWidth \* y)];

x = 0;

if ((intptr\_t)s & 3 || (intptr\_t)d & 3) { // either src or dst pointer is not aligned

for (; x<dstWidth; x++) {

\*d++ = \*s++; // have to do it byte-by-byte

}

} else {

// move 4 bytes at a time if aligned or alignment not enforced

s32 = (uint32\_t \*)s;

d32 = (uint32\_t \*)d;

for (; x<dstWidth-3; x+= 4) {

\*d32++ = \*s32++;

}

// any remaining stragglers?

s = (uint8\_t \*)s32;

d = (uint8\_t \*)d32;

for (; x<dstWidth; x++) {

\*d++ = \*s++;

}

}

} // for y

} // 8-bpp

else

{

uint16\_t \*s, \*d;

for (y=0; y<dstHeight; y++) {

s = (uint16\_t \*)&srcImage[2 \* srcWidth \* (y + startY) + startX \* 2];

d = (uint16\_t \*)&dstImage[(dstWidth \* y \* 2)];

x = 0;

if ((intptr\_t)s & 2 || (intptr\_t)d & 2) { // either src or dst pointer is not aligned

for (; x<dstWidth; x++) {

\*d++ = \*s++; // have to do it 16-bits at a time

}

} else {

// move 4 bytes at a time if aligned or alignment no enforced

s32 = (uint32\_t \*)s;

d32 = (uint32\_t \*)d;

for (; x<dstWidth-1; x+= 2) { // we can move 2 pixels at a time

\*d32++ = \*s32++;

}

// any remaining stragglers?

s = (uint16\_t \*)s32;

d = (uint16\_t \*)d32;

for (; x<dstWidth; x++) {

\*d++ = \*s++;

}

}

} // for y

} // 16-bpp case

} /\* cropImage() \*/

#if !defined(EI\_CLASSIFIER\_SENSOR) || EI\_CLASSIFIER\_SENSOR != EI\_CLASSIFIER\_SENSOR\_CAMERA

#error "Invalid model for current sensor"

#endif

// OV767X camera library override

#include <Arduino.h>

#include <Wire.h>

#define digitalPinToBitMask(P) (1 << (digitalPinToPinName(P) % 32))

#define portInputRegister(P) ((P == 0) ? &NRF\_P0->IN : &NRF\_P1->IN)

//

// OV7675::begin()

//

// Extends the OV767X library function. Some private variables are needed

// to use the OV7675::readFrame function.

//

int OV7675::begin(int resolution, int format, int fps)

{

pinMode(OV7670\_VSYNC, INPUT);

pinMode(OV7670\_HREF, INPUT);

pinMode(OV7670\_PLK, INPUT);

pinMode(OV7670\_XCLK, OUTPUT);

vsyncPort = portInputRegister(digitalPinToPort(OV7670\_VSYNC));

vsyncMask = digitalPinToBitMask(OV7670\_VSYNC);

hrefPort = portInputRegister(digitalPinToPort(OV7670\_HREF));

hrefMask = digitalPinToBitMask(OV7670\_HREF);

pclkPort = portInputRegister(digitalPinToPort(OV7670\_PLK));

pclkMask = digitalPinToBitMask(OV7670\_PLK);

// init driver to use full image sensor size

bool ret = OV767X::begin(VGA, format, fps);

width = OV767X::width(); // full sensor width

height = OV767X::height(); // full sensor height

bytes\_per\_pixel = OV767X::bytesPerPixel();

bytes\_per\_row = width \* bytes\_per\_pixel; // each pixel is 2 bytes

resize\_height = 2;

buf\_mem = NULL;

raw\_buf = NULL;

intrp\_buf = NULL;

//allocate\_scratch\_buffs();

return ret;

} /\* OV7675::begin() \*/

int OV7675::allocate\_scratch\_buffs()

{

//ei\_printf("allocating buffers..\r\n");

buf\_rows = height / resize\_row\_sz \* resize\_height;

buf\_size = bytes\_per\_row \* buf\_rows;

buf\_mem = ei\_malloc(buf\_size);

if(buf\_mem == NULL) {

ei\_printf("failed to create buf\_mem\r\n");

return false;

}

raw\_buf = (uint8\_t \*)DWORD\_ALIGN\_PTR((uintptr\_t)buf\_mem);

//ei\_printf("allocating buffers OK\r\n");

return 0;

}

int OV7675::deallocate\_scratch\_buffs()

{

//ei\_printf("deallocating buffers...\r\n");

ei\_free(buf\_mem);

buf\_mem = NULL;

//ei\_printf("deallocating buffers OK\r\n");

return 0;

}

//

// OV7675::readFrame()

//

// Overrides the OV767X library function. Fixes the camera output to be

// a far more desirable image. This image utilizes the full sensor size

// and has the correct aspect ratio. Since there is limited memory on the

// Nano we bring in only part of the entire sensor at a time and then

// interpolate to a lower resolution.

//

void OV7675::readFrame(void\* buffer)

{

allocate\_scratch\_buffs();

uint8\_t\* out = (uint8\_t\*)buffer;

noInterrupts();

// Falling edge indicates start of frame

while ((\*vsyncPort & vsyncMask) == 0); // wait for HIGH

while ((\*vsyncPort & vsyncMask) != 0); // wait for LOW

int out\_row = 0;

for (int raw\_height = 0; raw\_height < height; raw\_height += buf\_rows) {

// read in 640xbuf\_rows buffer to work with

readBuf();

resizeImage(width, buf\_rows,

raw\_buf,

resize\_col\_sz, resize\_height,

&(out[out\_row]),

16);

out\_row += resize\_col\_sz \* resize\_height \* bytes\_per\_pixel; /\* resize\_col\_sz \* 2 \* 2 \*/

}

interrupts();

deallocate\_scratch\_buffs();

} /\* OV7675::readFrame() \*/

//

// OV7675::readBuf()

//

// Extends the OV767X library function. Reads buf\_rows VGA rows from the

// image sensor.

//

void OV7675::readBuf()

{

int offset = 0;

uint32\_t ulPin = 33; // P1.xx set of GPIO is in 'pin' 32 and above

NRF\_GPIO\_Type \* port;

port = nrf\_gpio\_pin\_port\_decode(&ulPin);

for (int i = 0; i < buf\_rows; i++) {

// rising edge indicates start of line

while ((\*hrefPort & hrefMask) == 0); // wait for HIGH

for (int col = 0; col < bytes\_per\_row; col++) {

// rising edges clock each data byte

while ((\*pclkPort & pclkMask) != 0); // wait for LOW

uint32\_t in = port->IN; // read all bits in parallel

in >>= 2; // place bits 0 and 1 at the "bottom" of the register

in &= 0x3f03; // isolate the 8 bits we care about

in |= (in >> 6); // combine the upper 6 and lower 2 bits

raw\_buf[offset++] = in;

while ((\*pclkPort & pclkMask) == 0); // wait for HIGH

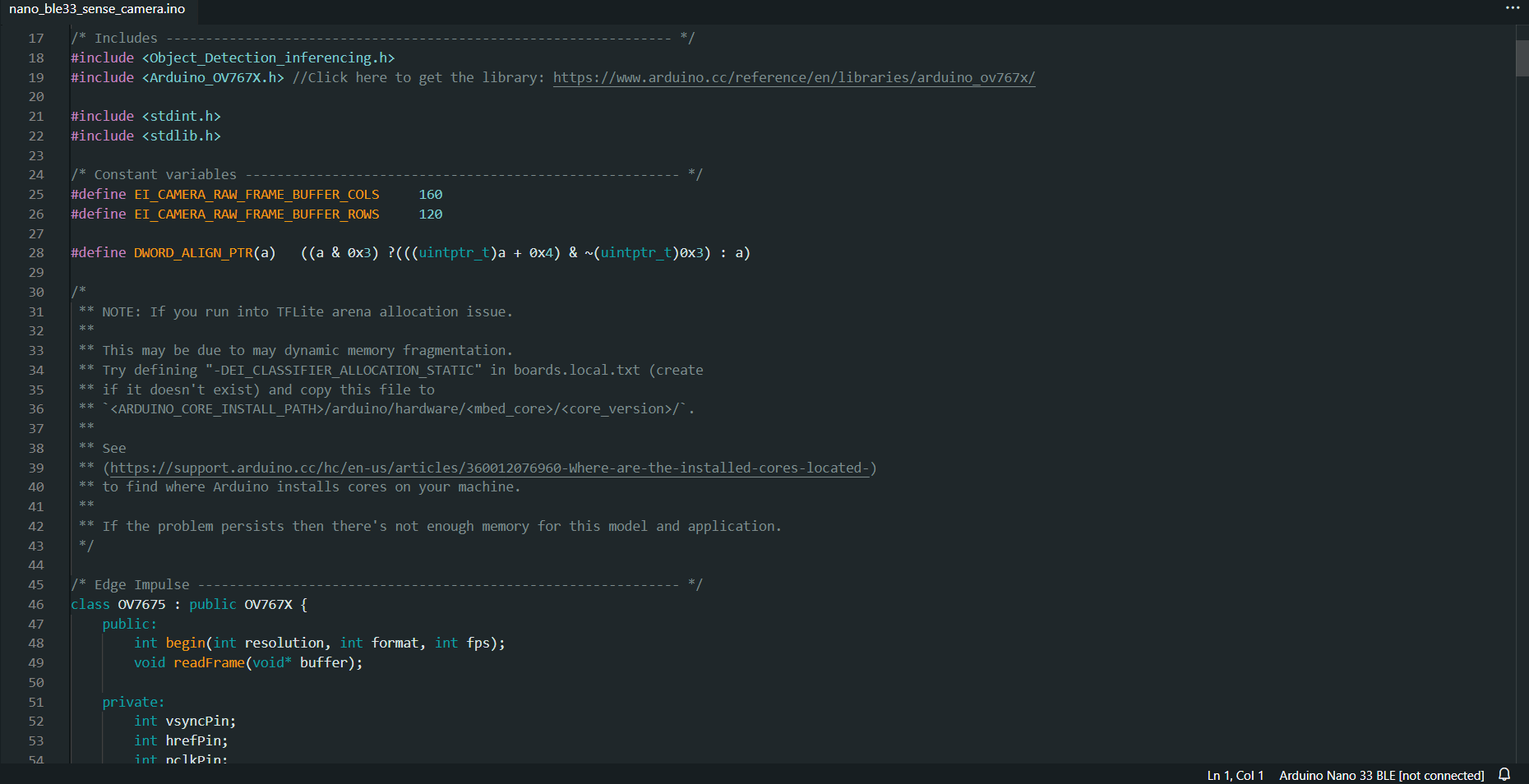
}

while ((\*hrefPort & hrefMask) != 0); // wait for LOW

}

} /\* OV7675::readBuf() \*/

1. **Screen shot of Arduino Terminal – Result**



**Conclusion:** Practically implemented detect an object using Edge Computing. Understood about custom object detection.