Name: Tanmay Sandip Khedekar

Class: TY-15 (Batch A)

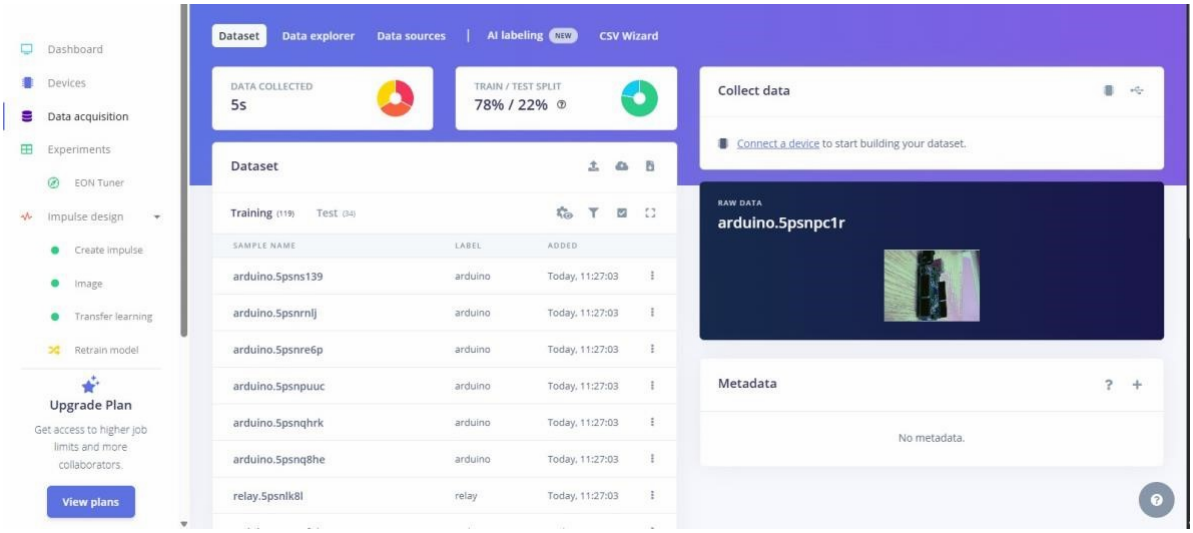
Roll Number: 2223122

Er.no:MITU22BTCS0906

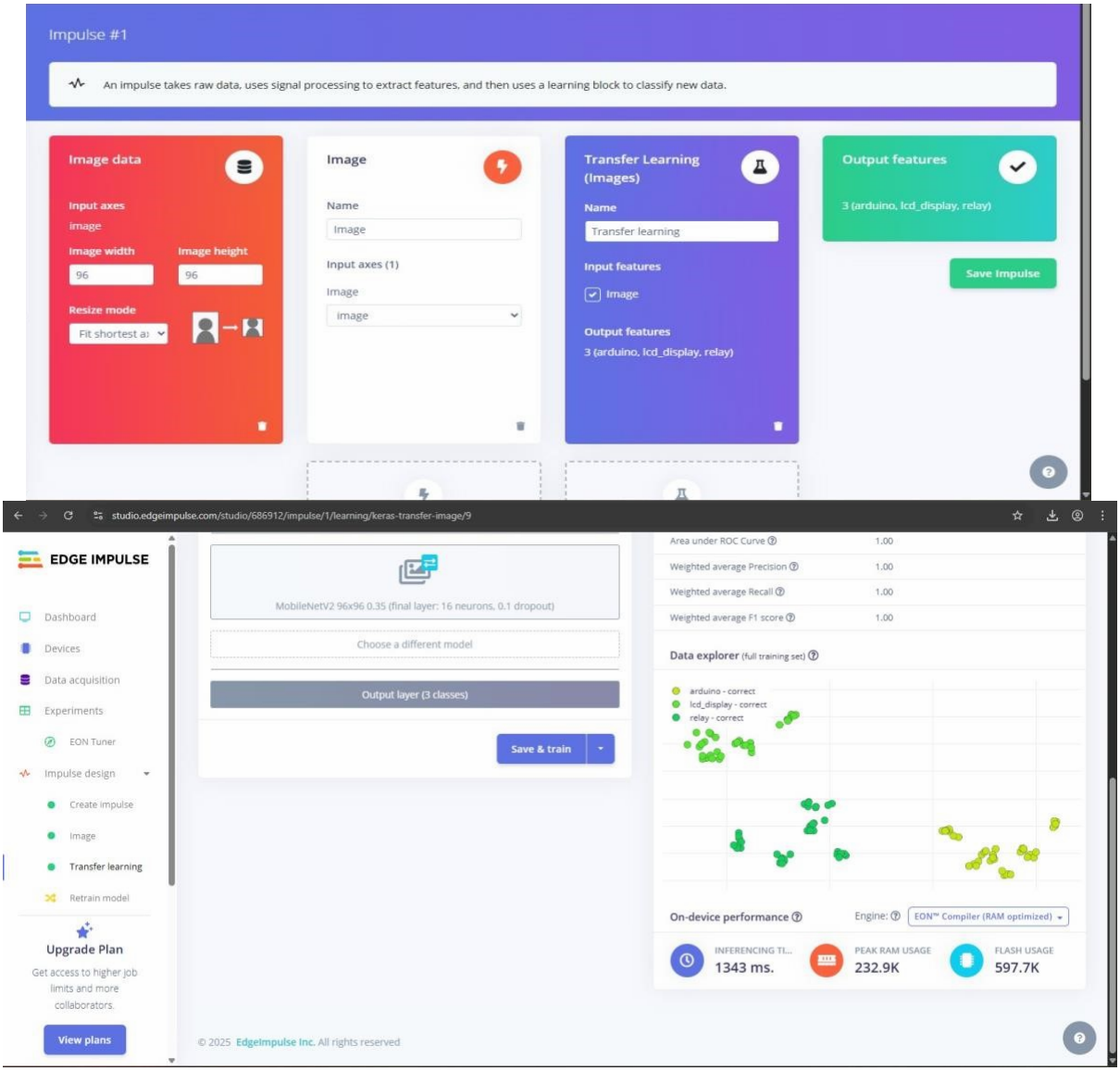
SUB:ECL

ECL Experiment 9

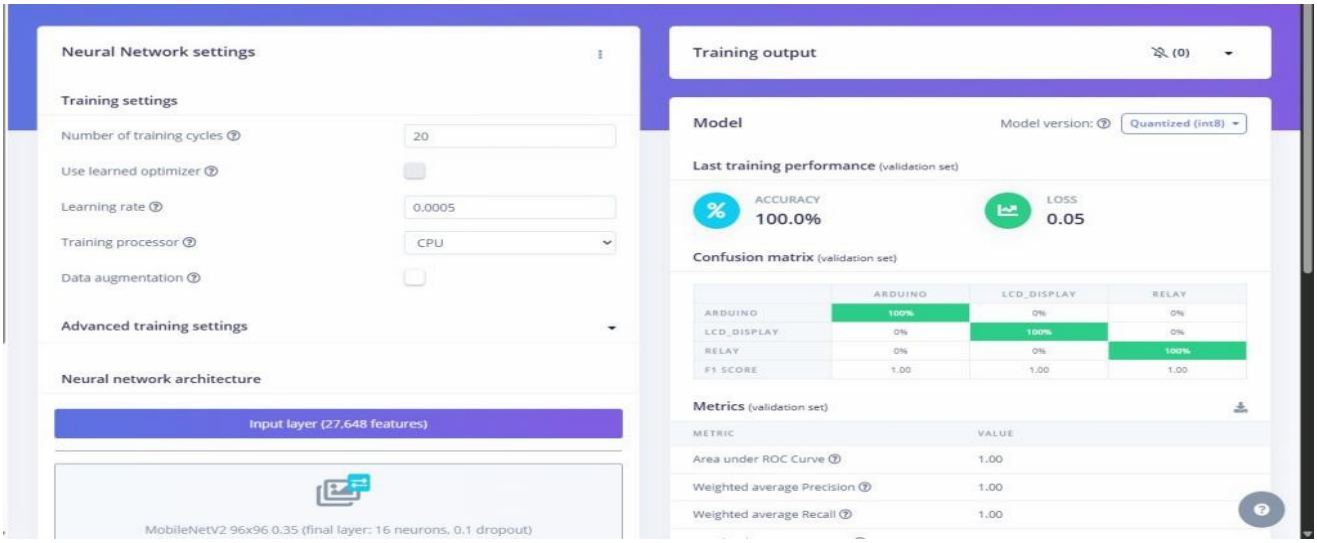
1. Dataset Image



2. Feature Extraction Image



3. Accuracy / Loss Confusion Matrix Image



5. Copy of the Arduino Code

```
 /* Edge Impulse ingestion SDK  *
Copyright © 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 <camera_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-coreslocated-
)
 ** 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)) {
```

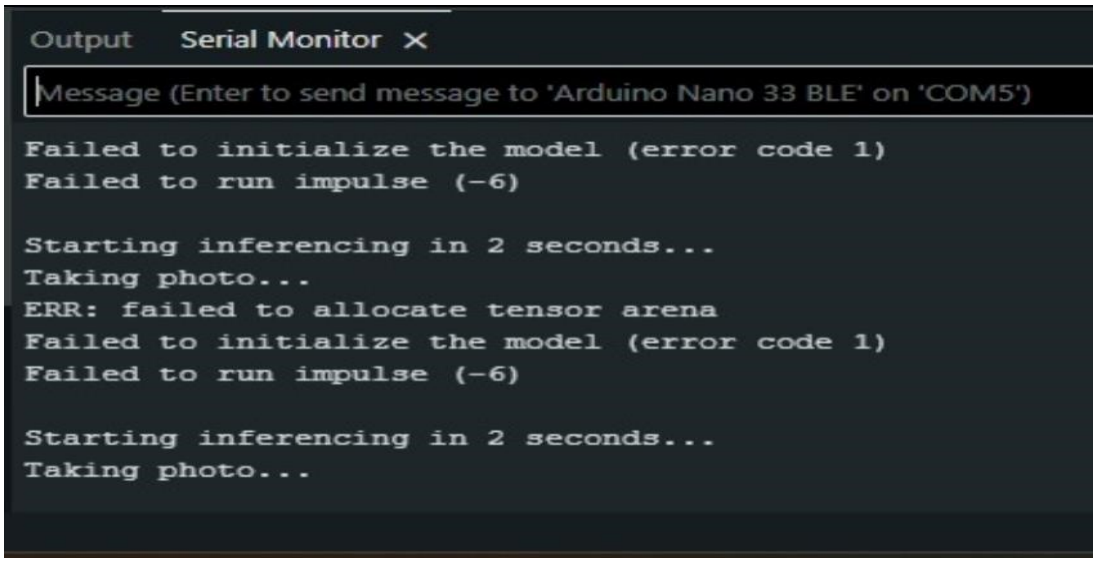
```
    Do_crop = true;
  }

  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 ½ 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 ½ 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_f
```

6. Output


Output
Serial Monitor
Message (Enter to send message to 'Arduino Nano 33 BLE' on 'COM5')
Failed to initialize the model (error code 1)
Failed to run impulse (-6)
Starting inferencing in 2 seconds...
Taking photo...
ERR: failed to allocate tensor arena
Failed to initialize the model (error code 1)
Failed to run impulse (-6)
Starting inferencing in 2 seconds...
Taking photo...