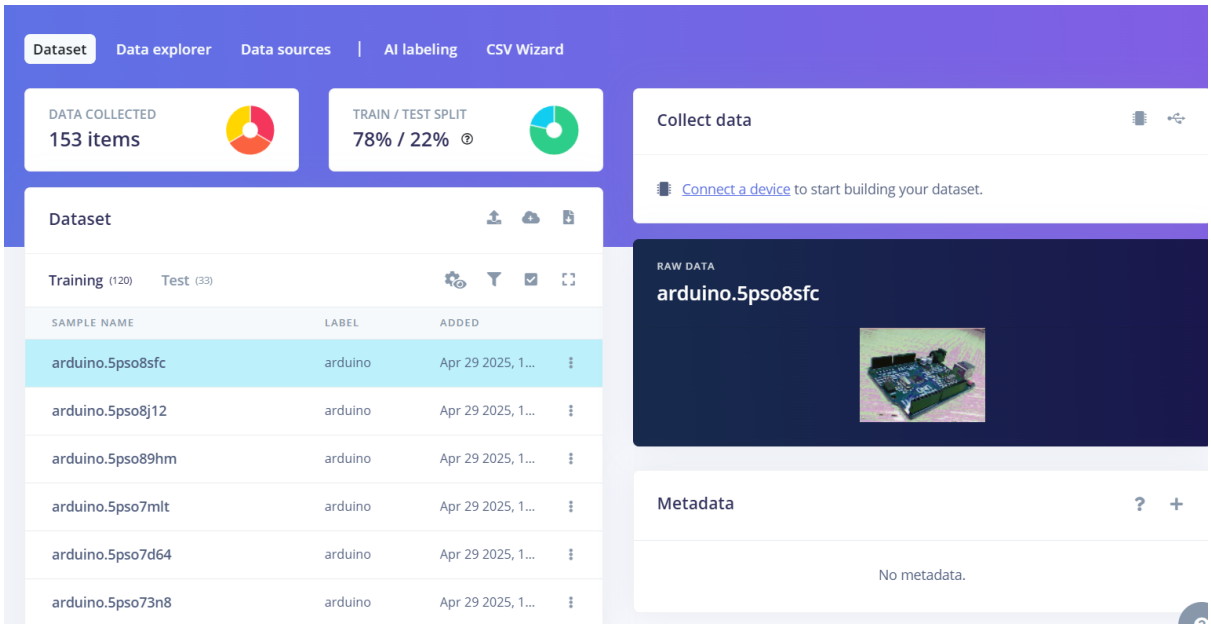
Name-Deep Gorle

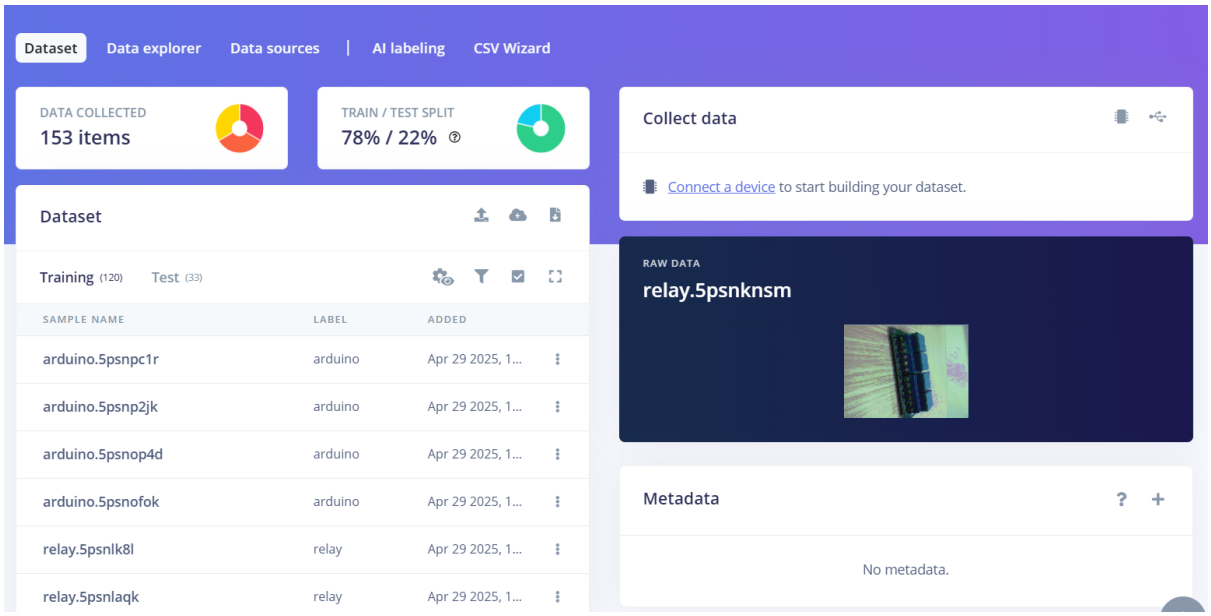
Roll no-22231068

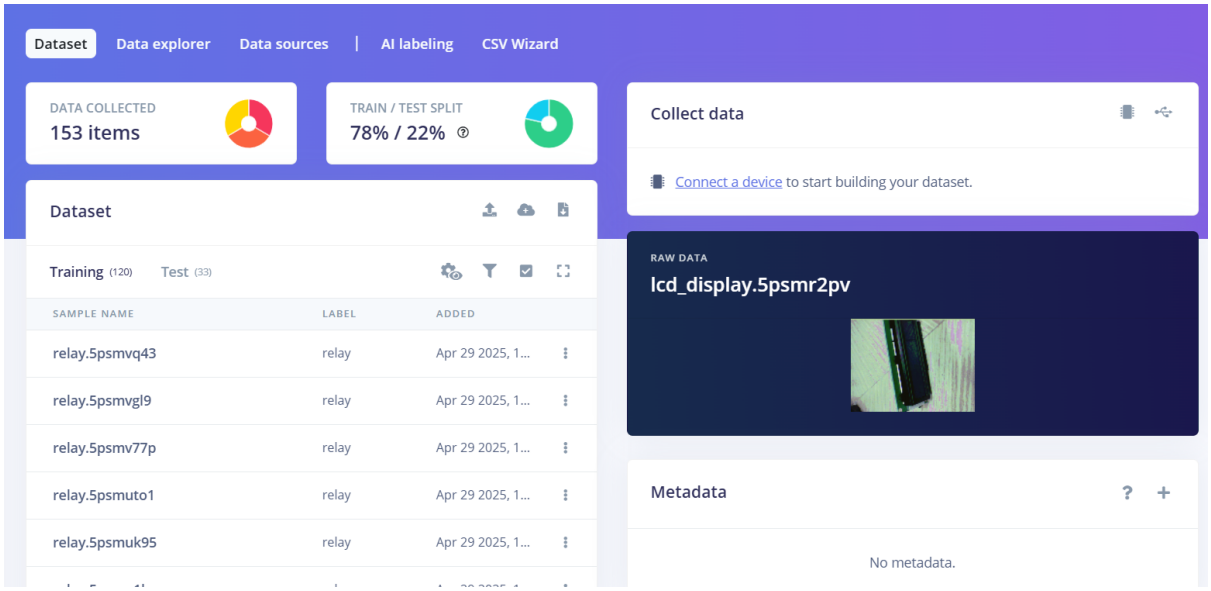
Class-TY-AIEC  Batch-B

## Experiment No-9

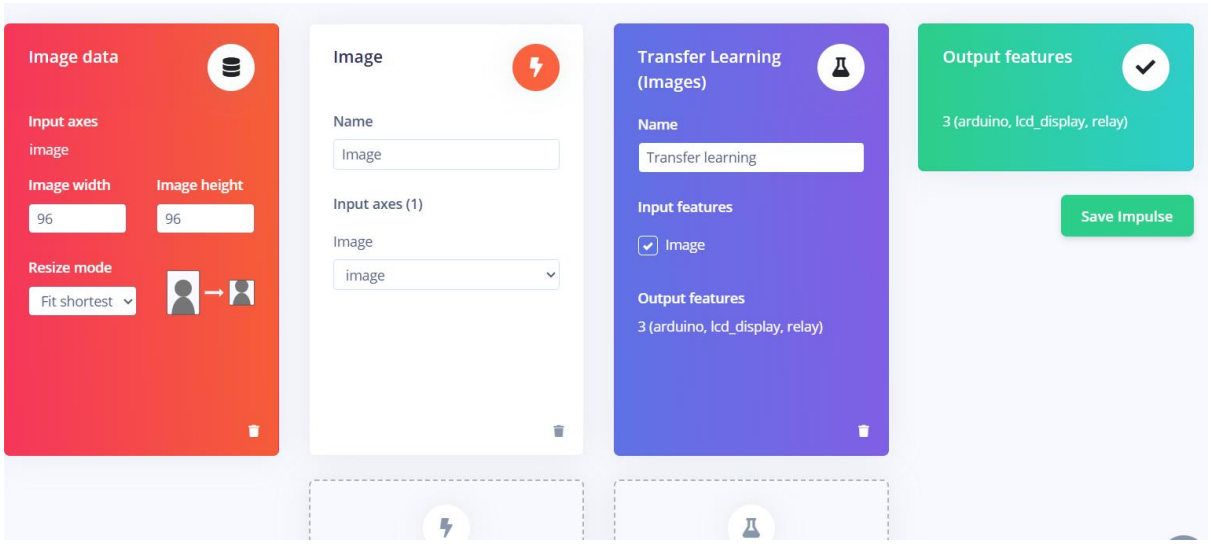
### 1 .DATASET-






## 2. Feature Extraction Image



## Raw features

0xbfa0a8, 0xbfa0a8, 0xbfa0a8, 0xbfa0a8, 0xbfa0a8, 0xbfa0a8, 0xbe9fa7, 0xbfa0a8, …

## Parameters

### Image

Color depth ⓘ Grayscale

Save parameters

## DSP result

### Image

### Processed features

0.6674, 0.6674, 0.6674, 0.6674, 0.6674, 0.6674, 0.6635, 0.6674, 0.6674, 0.6674, …

## On-device performance ⓘ

PROCESSING TIME
11 ms.

PEAK RAM USAGE
4 KB

## 3. Accuracy / Loss Confusion Matrix Image

### Model

Model version: Quantized (int8)

### Last training performance (validation set)

ACCURACY
100.0%

LOSS
0.00

### Confusion matrix (validation set)

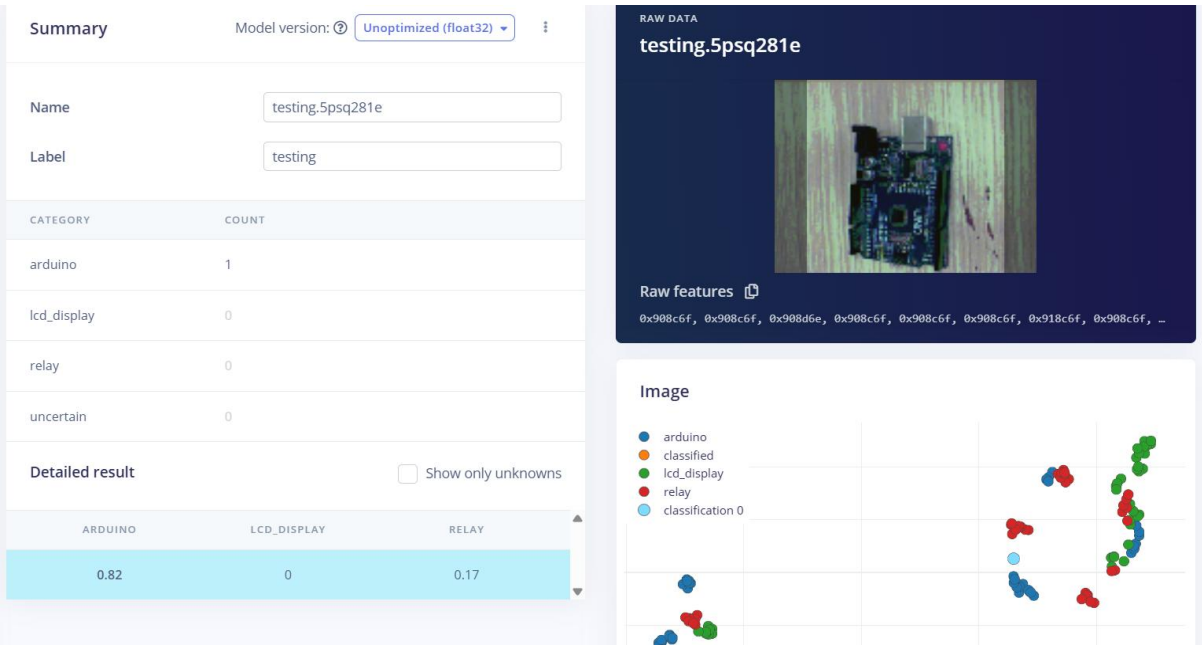
| | ARDUINO | LCD_DISPLAY | RELAY |
|---|---|---|---|
| ARDUINO | 100% | 0% | 0% |
| LCD_DISPLAY | 0% | 100% | 0% |
| RELAY | 0% | 0% | 100% |
| F1 SCORE | 1.00 | 1.00 | 1.00 |

### Metrics (validation set)

| METRIC | VALUE |
|---|---|
| Area under ROC Curve | 1.00 |
| Weighted average Precision | 1.00 |
| Weighted average Recall | 1.00 |
| Weighted average F1 score | 1.00 |

### Data explorer (full training set)

# 4. Validation Result



# 5. Copy of the Arduino Code-

```
/* 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-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;
```

```
    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;
```

```
} 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

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()
{
```

```ino
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;
            }
```

```
                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;
                }
```

```
                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_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
 */
```

```
//
// 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() */
```

## 6. Output

```
12:08:33.252 -> Taking photo...
12:08:36.032 -> ERR: failed to allocate tensor arena
12:08:36.032 -> Failed to initialize the model (error code 1)
12:08:36.032 -> Failed to run impulse (-6)
12:08:36.032 ->
12:08:36.032 -> Starting inferencing in 2 seconds...
12:08:38.035 -> Taking photo...
12:08:40.821 -> ERR: failed to allocate tensor arena
12:08:40.821 -> Failed to initialize the model (error code 1)
12:08:40.821 -> Failed to run impulse (-6)
12:08:40.821 ->
```