

Name: Tanmay Khedekar

class: TY-15 (Batch A)

Roll Number: 2223122

Enrolment Number: MITU22BTCS0906

ECL Experiment 7

Introduction

Study of Classification learning block using a NN Classifier on Edge Devices

Objective: Build a project to detect the keywords using built-in sensor on Nano BLE Sense /
Mobile Phone

Tasks:

- Generate the dataset for keyword
- Configure BLE Sense / Mobile for Edge Impulse
- Building and Training a Model

Study of Confusion matrix

Introduction

Edge Impulse is a development platform for machine learning on edge devices, targeted at developers who want to create intelligent device solutions. The "classification block" 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

1. Dataset Image

The screenshot displays the Edge Impulse web interface for a project named 'Tanmay_Khedekar / Tanmay_Khedekar-project-1'. The interface is divided into a left sidebar with navigation options (Dashboard, Devices, Data acquisition, Experiments, EON Tuner, Impulse design) and a main content area. The main area has tabs for 'Dataset', 'Data explorer', 'Data sources', 'AI labeling', and 'CSV Wizard'. The 'Dataset' tab is active, showing a summary of data collected (3m 9s) and a train/test split (76% / 24%). Below this, a table lists the dataset samples, all labeled 'green'. A 'Collect data' button is visible, along with a prompt to 'Connect a device to start building your dataset.' The bottom of the image shows a Windows taskbar with the date 28-04-2025 and time 18:45.

Dataset

SAMPLE NAME	LABEL	ADDED	LENGTH
green.5o74tacv	green	Apr 09 2025, 1...	6s
green.5o74r6sc	green	Apr 09 2025, 1...	6s
green.5o74qpce	green	Apr 09 2025, 1...	6s
green.5o74mqrq	green	Apr 09 2025, 1...	6s
green.5o74n9m6	green	Apr 09 2025, 1...	6s
green.5o74mdcm	green	Apr 09 2025, 1...	6s

2. Feature Extraction Image

This screenshot shows the 'Create Impulse' interface in the Edge Impulse Studio. The interface is divided into several sections:

- Time series data:** A red panel on the left with settings for input axes (red, green, blue, brightness, proximity, gesture), window size (1,000 ms), window increase (stride) (1,000 ms), frequency (1 Hz), and zero-pad data (checked).
- Flatten:** A white panel in the center with a name field set to 'Flatten' and input axes (red, green, blue, brightness, proximity, gesture) where red, green, and blue are checked.
- Classification:** A blue panel on the right with a name field set to 'Classifier', input features (Flatten) checked, and output features (3 (blue, green, red)).
- Output features:** A green panel on the far right showing the output features (3 (blue, green, red)) and a 'Save Impulse' button.

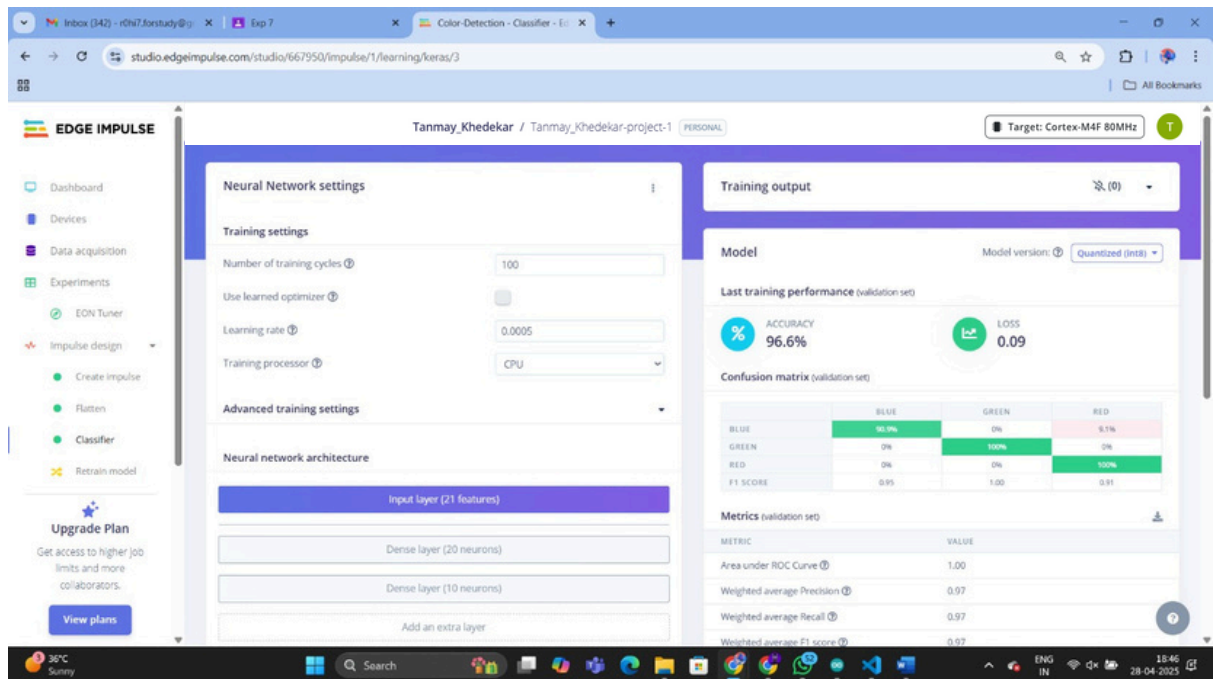
The top of the interface shows the user 'Tanmay_Khedekar' and the target 'Cortex-M4F 80MHz'. The bottom of the interface shows a Windows taskbar with various application icons and system information.

This screenshot shows the 'Generate Features' interface in the Edge Impulse Studio. The interface is divided into several sections:

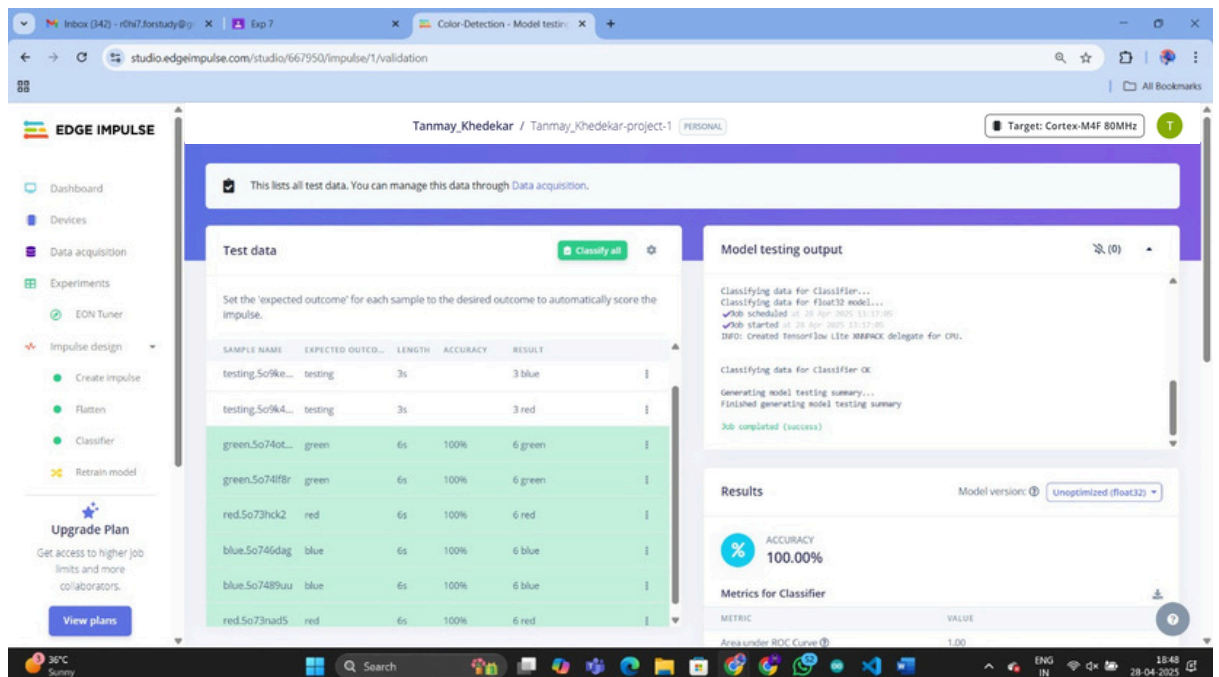
- Parameters:** A section on the left with settings for training set (Data in training set: 2m 24s, Classes: 3 (blue, green, red), Training windows: 144, Calculate feature importance: checked) and a 'Generate features' button.
- Feature explorer:** A section on the right with a 3D scatter plot showing data points for blue, green, and red classes. The X Axis is 'red Average', the Y Axis is 'green Average', and the Z Axis is 'blue Average'. The plot shows a clear separation between the three classes.

The top of the interface shows the user 'Tanmay_Khedekar' and the target 'Cortex-M4F 80MHz'. The bottom of the interface shows a Windows taskbar with various application icons and system information.

3. Accuracy / Loss Confusion Matrix Image



4. Validation Result



5. Copy of the Arduino Code

```
/* 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 <Color-Detection_inferencing.h>
#include <Arduino_LSM9DS1.h> //Click here to get the library:
https://www.arduino.cc/reference/en/libraries/arduino\_lsm9ds1/
#include <Arduino_LPS22HB.h> //Click here to get the library:
https://www.arduino.cc/reference/en/libraries/arduino\_lps22hb/
#include <Arduino_HTS221.h> //Click here to get the library:
https://www.arduino.cc/reference/en/libraries/arduino\_hts221/
#include <Arduino_APDS9960.h> //Click here to get the library:
https://www.arduino.cc/reference/en/libraries/arduino\_apds9960/

enum sensor_status {
    NOT_USED = -1,
    NOT_INIT,
    INIT,
    SAMPLED
};

/** Struct to link sensor axis name to sensor value function */
typedef struct{
    const char *name;
    float *value;
    uint8_t (*poll_sensor)(void);
    bool (*init_sensor)(void);
    sensor_status status;
} eiSensors;

/* Constant defines -----
 */
```

```

#define CONVERT_G_TO_MS2    9.80665f

/**
 * When data is collected by the Edge Impulse Arduino Nano 33 BLE Sense
 * firmware, it is limited to a 2G range. If the model was created with a
 * different sample range, modify this constant to match the input values.
 * See https://github.com/edgeimpulse/firmware-arduino-nano-33-ble-sense/blob/master/src/sensors/ei\_lsm9ds1.cpp
 * for more information.
 */
#define MAX_ACCEPTED_RANGE 2.0f

/** Number sensor axes used */
#define N_SENSORS    18

/* Forward declarations -----
- */
float ei_get_sign(float number);

bool init_IMU(void);
bool init_HTS(void);
bool init_BARO(void);
bool init_APDS(void);

uint8_t poll_acc(void);
uint8_t poll_gyr(void);
uint8_t poll_mag(void);
uint8_t poll_HTS(void);
uint8_t poll_BARO(void);
uint8_t poll_APDS_color(void);
uint8_t poll_APDS_proximity(void);
uint8_t poll_APDS_gesture(void);

/* Private variables -----
*/
static const bool debug_nn = false; // Set this to true to see e.g. features
generated from the raw signal

static float data[N_SENSORS];
static bool ei_connect_fusion_list(const char *input_list);

static int8_t fusion_sensors[N_SENSORS];
static int fusion_ix = 0;

/** Used sensors value function connected to label name */
eiSensors sensors[] =
{
    "accX", &data[0], &poll_acc, &init_IMU, NOT_USED,

```

```

"accY", &data[1], &poll_acc, &init_IMU, NOT_USED,
"accZ", &data[2], &poll_acc, &init_IMU, NOT_USED,
"gyrX", &data[3], &poll_gyr, &init_IMU, NOT_USED,
"gyrY", &data[4], &poll_gyr, &init_IMU, NOT_USED,
"gyrZ", &data[5], &poll_gyr, &init_IMU, NOT_USED,
"magX", &data[6], &poll_mag, &init_IMU, NOT_USED,
"magY", &data[7], &poll_mag, &init_IMU, NOT_USED,
"magZ", &data[8], &poll_mag, &init_IMU, NOT_USED,

"temperature", &data[9], &poll_HTS, &init_HTS, NOT_USED,
"humidity", &data[10], &poll_HTS, &init_HTS, NOT_USED,

"pressure", &data[11], &poll_BARO, &init_BARO, NOT_USED,

"red", &data[12], &poll_APDS_color, &init_APDS, NOT_USED,
"green", &data[13], &poll_APDS_color, &init_APDS, NOT_USED,
"blue", &data[14], &poll_APDS_color, &init_APDS, NOT_USED,
"brightness", &data[15], &poll_APDS_color, &init_APDS, NOT_USED,
"proximity", &data[16], &poll_APDS_proximity, &init_APDS, NOT_USED,
"gesture", &data[17], &poll_APDS_gesture, &init_APDS, NOT_USED,
};

/**
 * @brief      Arduino setup function
 */
void setup()
{
    /* Init serial */
    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 Sensor Fusion Inference\r\n");

    /* Connect used sensors */
    if(ei_connect_fusion_list(EI_CLASSIFIER_FUSION_AXES_STRING) == false) {
        ei_printf("ERR: Errors in sensor list detected\r\n");
        return;
    }

    /* Init & start sensors */

    for(int i = 0; i < fusion_ix; i++) {
        if (sensors[fusion_sensors[i]].status == NOT_INIT) {
            sensors[fusion_sensors[i]].status =
(sensor_status)sensors[fusion_sensors[i]].init_sensor();
            if (!sensors[fusion_sensors[i]].status) {

```

```

        ei_printf("%s axis sensor initialization failed.\r\n",
sensors[fusion_sensors[i]].name);
    }
    else {
        ei_printf("%s axis sensor initialization successful.\r\n",
sensors[fusion_sensors[i]].name);
    }
}
}

/**
 * @brief      Get data and run inferencing
 */
void loop()
{
    ei_printf("\nStarting inferencing in 2 seconds...\r\n");

    delay(2000);

    if (EI_CLASSIFIER_RAW_SAMPLES_PER_FRAME != fusion_ix) {
        ei_printf("ERR: Sensors don't match the sensors required in the
model\r\n"
        "Following sensors are required: %s\r\n",
EI_CLASSIFIER_FUSION_AXES_STRING);
        return;
    }

    ei_printf("Sampling...\r\n");

    // Allocate a buffer here for the values we'll read from the sensor
    float buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE] = { 0 };

    for (size_t ix = 0; ix < EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE; ix +=
EI_CLASSIFIER_RAW_SAMPLES_PER_FRAME) {
        // Determine the next tick (and then sleep later)
        int64_t next_tick = (int64_t)micros() +
((int64_t)EI_CLASSIFIER_INTERVAL_MS * 1000);

        for(int i = 0; i < fusion_ix; i++) {
            if (sensors[fusion_sensors[i]].status == INIT) {
                sensors[fusion_sensors[i]].poll_sensor();
                sensors[fusion_sensors[i]].status = SAMPLED;
            }
            if (sensors[fusion_sensors[i]].status == SAMPLED) {
                buffer[ix + i] = *sensors[fusion_sensors[i]].value;
                sensors[fusion_sensors[i]].status = INIT;
            }

```



```

    }

    int64_t wait_time = next_tick - (int64_t)micros();

    if(wait_time > 0) {
        delayMicroseconds(wait_time);
    }
}

// Turn the raw buffer in a signal which we can the classify
signal_t signal;
int err = numpy::signal_from_buffer(buffer,
EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE, &signal);
if (err != 0) {
    ei_printf("ERR:(%d)\r\n", err);
    return;
}

// Run the classifier
ei_impulse_result_t result = { 0 };

err = run_classifier(&signal, &result, debug_nn);
if (err != EI_IMPULSE_OK) {
    ei_printf("ERR:(%d)\r\n", err);
    return;
}

// print the predictions
ei_printf("Predictions (DSP: %d ms., Classification: %d ms., Anomaly: %d
ms.):\r\n",
    result.timing.dsp, result.timing.classification,
result.timing.anomaly);
    for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
        ei_printf("%s: %.5f\r\n", result.classification[ix].label,
result.classification[ix].value);
    }
#if EI_CLASSIFIER_HAS_ANOMALY == 1
    ei_printf("    anomaly score: %.3f\r\n", result.anomaly);
#endif
}

#if !defined(EI_CLASSIFIER_SENSOR) || (EI_CLASSIFIER_SENSOR !=
EI_CLASSIFIER_SENSOR_FUSION && EI_CLASSIFIER_SENSOR !=
EI_CLASSIFIER_SENSOR_ACCELEROMETER)
#error "Invalid model for current sensor"
#endif

/**

```

```

* @brief Go through sensor list to find matching axis name
*
* @param axis_name
* @return int8_t index in sensor list, -1 if axis name is not found
*/
static int8_t ei_find_axis(char *axis_name)
{
    int ix;
    for(ix = 0; ix < N_SENSORS; ix++) {
        if(strstr(axis_name, sensors[ix].name)) {
            return ix;
        }
    }
    return -1;
}

/**
* @brief Check if requested input list is valid sensor fusion, create sensor
buffer
*
* @param[in] input_list      Axes list to sample (ie. "accX + gyrY + magZ")
* @retval false if invalid sensor_list
*/
static bool ei_connect_fusion_list(const char *input_list)
{
    char *buff;
    bool is_fusion = false;

    /* Copy const string in heap mem */
    char *input_string = (char *)ei_malloc(strlen(input_list) + 1);
    if (input_string == NULL) {
        return false;
    }
    memset(input_string, 0, strlen(input_list) + 1);
    strncpy(input_string, input_list, strlen(input_list));

    /* Clear fusion sensor list */
    memset(fusion_sensors, 0, N_SENSORS);
    fusion_ix = 0;

    buff = strtok(input_string, "+");

    while (buff != NULL) { /* Run through buffer */
        int8_t found_axis = 0;

        is_fusion = false;
        found_axis = ei_find_axis(buff);
    }
}

```

```

        if(found_axis >= 0) {
            if(fusion_ix < N_SENSORS) {
                fusion_sensors[fusion_ix++] = found_axis;
                sensors[found_axis].status = NOT_INIT;
            }
            is_fusion = true;
        }

        buff = strtok(NULL, "+ ");
    }

    ei_free(input_string);

    return is_fusion;
}

/**
 * @brief Return the sign of the number
 *
 * @param number
 * @return int 1 if positive (or 0) -1 if negative
 */
float ei_get_sign(float number) {
    return (number >= 0.0) ? 1.0 : -1.0;
}

bool init_IMU(void) {
    static bool init_status = false;
    if (!init_status) {
        init_status = IMU.begin();
    }
    return init_status;
}

bool init-HTS(void) {
    static bool init_status = false;
    if (!init_status) {
        init_status = HTS.begin();
    }
    return init_status;
}

bool init_BARO(void) {
    static bool init_status = false;
    if (!init_status) {
        init_status = BARO.begin();
    }
    return init_status;
}

```

```

}

bool init_APDS(void) {
    static bool init_status = false;
    if (!init_status) {
        init_status = APDS.begin();
    }
    return init_status;
}

uint8_t poll_acc(void) {

    if (IMU.accelerationAvailable()) {

        IMU.readAcceleration(data[0], data[1], data[2]);

        for (int i = 0; i < 3; i++) {
            if (fabs(data[i]) > MAX_ACCEPTED_RANGE) {
                data[i] = ei_get_sign(data[i]) * MAX_ACCEPTED_RANGE;
            }
        }

        data[0] *= CONVERT_G_TO_MS2;
        data[1] *= CONVERT_G_TO_MS2;
        data[2] *= CONVERT_G_TO_MS2;
    }

    return 0;
}

uint8_t poll_gyr(void) {

    if (IMU.gyroscopeAvailable()) {
        IMU.readGyroscope(data[3], data[4], data[5]);
    }
    return 0;
}

uint8_t poll_mag(void) {

    if (IMU.magneticFieldAvailable()) {
        IMU.readMagneticField(data[6], data[7], data[8]);
    }
    return 0;
}

uint8_t poll_HTS(void) {

```

```

    data[9] = HTS.readTemperature();
    data[10] = HTS.readHumidity();
    return 0;
}

uint8_t poll_BARO(void) {

    data[11] = BARO.readPressure(); // (PSI/MILLIBAR/KILOPASCAL) default kPa
    return 0;
}

uint8_t poll_APDS_color(void) {

    int temp_data[4];
    if (APDS.colorAvailable()) {
        APDS.readColor(temp_data[0], temp_data[1], temp_data[2],
temp_data[3]);

        data[12] = temp_data[0];
        data[13] = temp_data[1];
        data[14] = temp_data[2];
        data[15] = temp_data[3];
    }
}

uint8_t poll_APDS_proximity(void) {

    if (APDS.proximityAvailable()) {
        data[16] = (float)APDS.readProximity();
    }
    return 0;
}

uint8_t poll_APDS_gesture(void) {
    if (APDS.gestureAvailable()) {
        data[17] = (float)APDS.readGesture();
    }
    return 0;
}

```

6. Output

Starting Nano BLE Sense Classification...

Sensor data collected.

Running inference...

Predicted Class: Green

Confidence: 86.3%

Raw Output:

- Red: 10.2%
- Green: 86.3%
- Blue: 3.5%

Waiting for next sensor input...

Predicted Class: Red

Confidence: 92.8%

Raw Output:

- Red: 92.8%
- Green: 5.1%
- Blue: 2.1%

Waiting for next sensor input...