Zahvala

To magistrsko delo ne bi bilo možno brez pomoči nekaterih ljudi. Prvo bi se rad zahvalil mentorju Iztoku Krambergerju in somentorju Vojislavu Draganu Milivojeviču za njune neprecenljive nasvete in napotke. Posebej bi se rad zahvalil Luki Mustafi, mojemu šefu in nenapisanemu mentorju. Hvala ti za nasvete, za mnoge sestanke, da si skrbel da je napisano dosegalo visoke standarde in za vedno prižgan AWS server. Rad bi se zahvalil mojim sodelavcem na Irnasu za njihovo podporo, še posebej mojima prijateljema Njecu in Vidu, ker sta si vzela čas za pregled mojega dela.

Rad bi se zahvalil Arribadi Initiative, ki je priskrbela nabor termalnih slik in tako sploh omogočila nastanek tega dela. Hkrati bi se tudi rad zahvalil Danileu Situnayake-ju in Janu Jongboom-u, za odgovore na vsa moja vprašanja v zvezi s TinyML-om.

Rad bi se zahvalil mojim staršem za njihovo podporo in spodbudo skozi vsa leta mojega šolanja. Brez vaju dveh ne bi bil tukaj, kjer sem sedaj.

Nazadnje se želim zahvaliti moji punci Hristini, ker je verjela vame in me podpirala med pisanjem magistrskega dela.

Acknowledgement

This Master's thesis would not be possible without assistance of certain people. Firstly, I would like to thank mentor Iztok Kramberger and co-mentor Vojislav Dragan Milivojević for their invaluable guidance and feedback. Special thanks goes to Luka Mustafa, my boss and unwritten mentor. Thank your for your advices, for numerous feedback sessions, for making sure that the written is up to the high standards and for 24/7 AWS server. I would like to thank my colleagues at Irnas for their support, especially my friends Nejc and Vid, for taking the time to read and correct my work.

I want to thank Arribada Initiative for providing me with thermal image dataset, thus making this thesis possible in the first place. Additionally I would like to thank Daniel Situnayake and Jan Jongboom, for answering all my TinyMl related questions.

I want to thank my parents for supporting and encouraging me through the years of my eduction. Without you, I would not be here, where I am today.

Lastly I want to thank my girlfriend Hristina for believing in me and supporting me during thesis writing.

Table of Contents

List of Figures

List of Tables

1 Introduction

2 Design and implementation of the early warning system

General structure and tasks of an early warning system were already described in chapter **??**. As mentioned before, an early warning system consists of two different components:

1. Several small embedded devices, which are deployed in the field. They capture images with a thermal camera and process them. They sent messages over a wireless network.
2. One gateway, which is receiving messages and relaying them to an application server over the Internet connection.

In this chapter, we focus on the structure and design of the deployed embedded system, both from hardware and firmware perspective. We also describe the construction of an application server, how received data is processed, stored and presented.

The general block diagram of an embedded system with a thermal camera is presented in Figure 2.1

The embedded system consists of two different microcontrollers with two distinct tasks, a thermal camera, PIR sensor, wireless communication module, power switch and battery.

Powerful, high-performance microcontroller and thermal camera are usually turned off, to conserve battery life. A less capable, but low-power microcontroller spends most of the time in low-power mode, waiting for a wakeup trigger from the PIR sensor. The PIR sensor points in the same direction as the thermal camera and detects any IR radiation of a passing object.

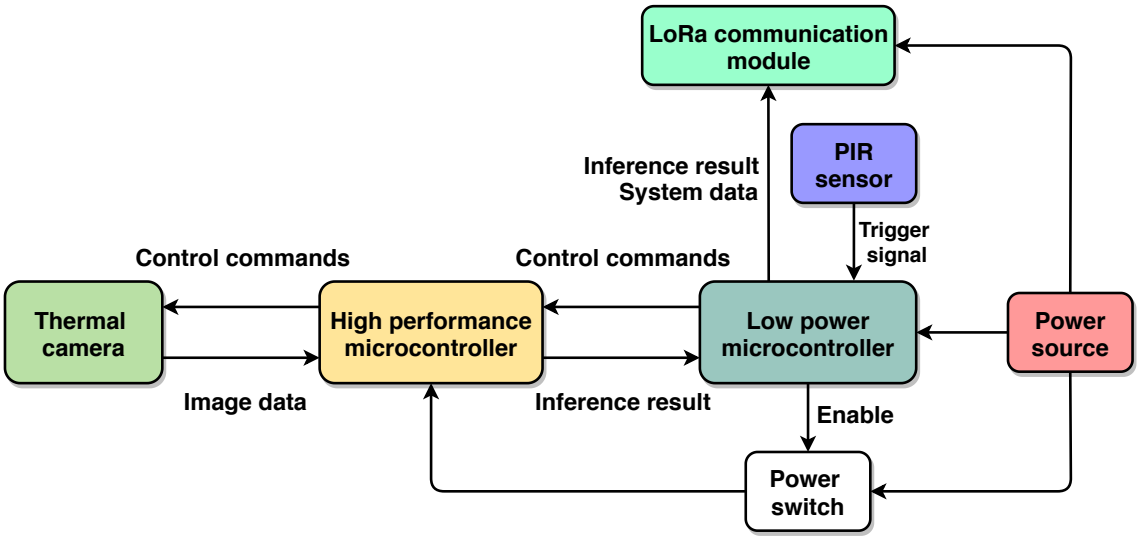

Figure 2.1: General block diagram of an embedded system

If an object passes PIR's field of vision, PIR sensor produces trigger signal, which consequently wakes up a low-power microcontroller. The microcontroller then enables the power supply to the high-performance microcontroller and thermal camera, and sends a command request for image capture and processing.

The thermal camera only communicates with a high-performance microcontroller, which configures it and requests image data. That data is then input into a Neural Network algorithm, which computes probability results that are sent back to a low-power microcontroller. Low-power microcontroller then packs the data and sends it over the radio through a wireless communication module. The power source to a high-performance microcontroller and thermal camera is then turned off to conserve power. Diagram of the described procedure can also be seen in Figure 2.2.

Figure 2.2: Diagram describing behavior of embedded early warning system

2.1 Hardware

In this section, we present concrete components that we used to implement the embedded part of the early warning system. The hardware version of the embedded

system diagram is presented in Figure 2.3. The system consists of various development and evaluation boards.

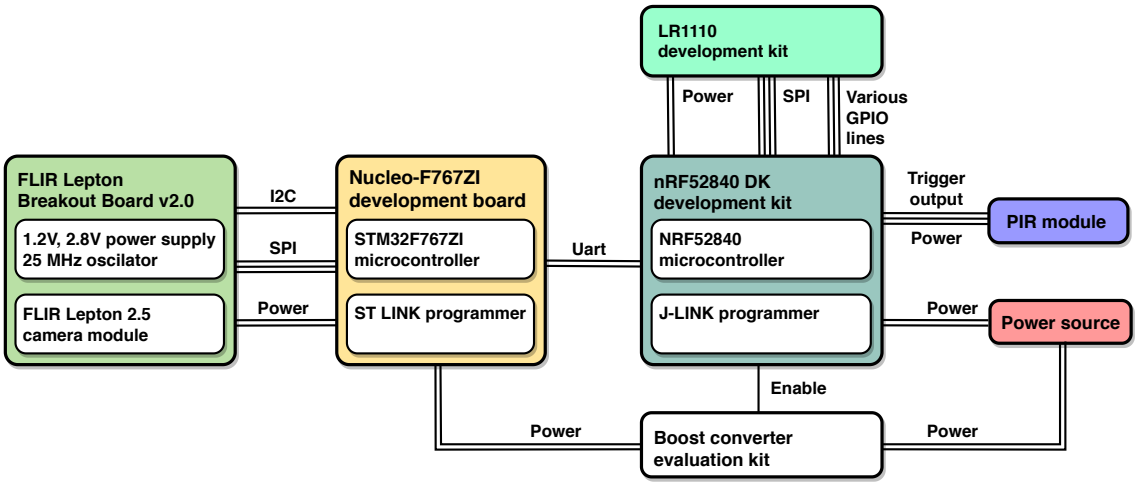

Figure 2.3: Hardware diagram of embedded early warning system

2.1.1 Nucleo-F767ZI

Nucleo-F767ZI (seen on Figure 2.4) is a development board made by STMicroelectronics. Board features STM32F767ZI (STM32) microcontroller with Cortex-M7 core, which has 2 MB of flash, 512 kB of SRAM and can operate at clock speed of 216 MHz. It also features memory caches and a flash accelerator, which provide an extra boost in performance. It is convenient to program it, as it includes onboard ST-LINK programmer circuit.

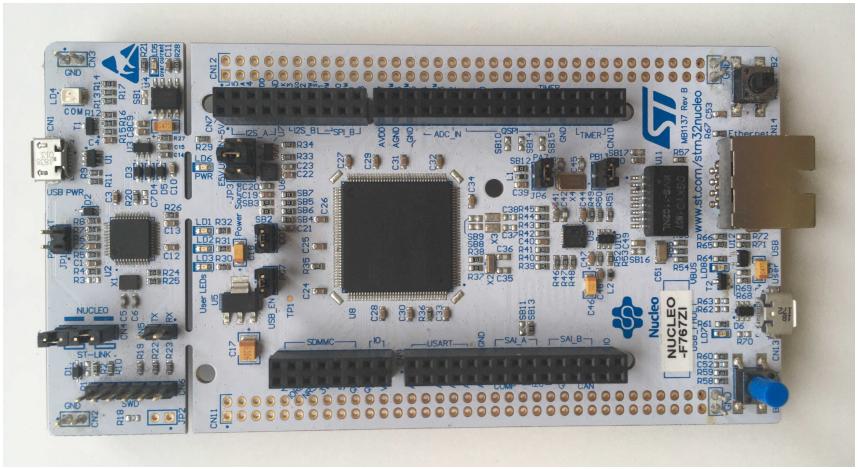

Figure 2.4: Nucleo-F767ZI development board

We chose this microcontroller simply because it is one of the more powerful general purpose microcontrollers on the market. As we knew that Neural Networks are computationally expensive to compute and that models can be quite large in terms of memory, we selected it knowing that we can always scale down, if we have to.

2.1.2 nRF52840 DK

For the part of the system, which had to contain a low-power microcontroller and would control the communication module and power control for the Nucleo-F767ZI board, we decided to use the nRF52840 DK development kit. The development kit, made by Nordic Semiconductor, can be seen in Figure 2.5

The main logic on the board is provided by an nRF52840 (nRF52) microcontroller with Cortex-M4 core, which has 1 MB of flash, 256 kB of RAM and a Bluetooth 5 support. nRF52 has a consumption of 0.5 µA in sleep mode, which makes it ideal for our purpose.

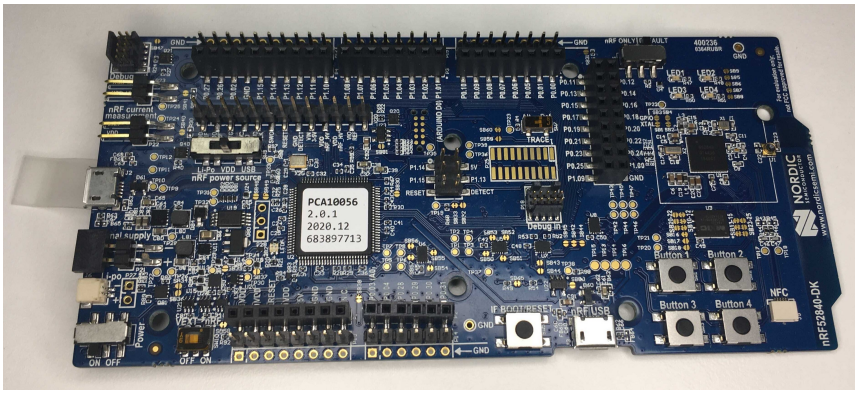

Figure 2.5: nRF52840DK development board

2.1.3 LR1110 development kit

For the role of the LoRa transceiver module, we decided to use Semtech's development kit which uses the LR1110 chip. LR1110 is a multi-functional solution as it contains LoRa transceiver, GNSS and WiFi geoposition scanning modules. Development kit seen in Figure 2.6 contains an LR1110 chip, three different antennas and their respective tuning networks. It comes in a convenient Arduino shield form factor, which means that we can directly attach it to nRF52 without any jumper wires.

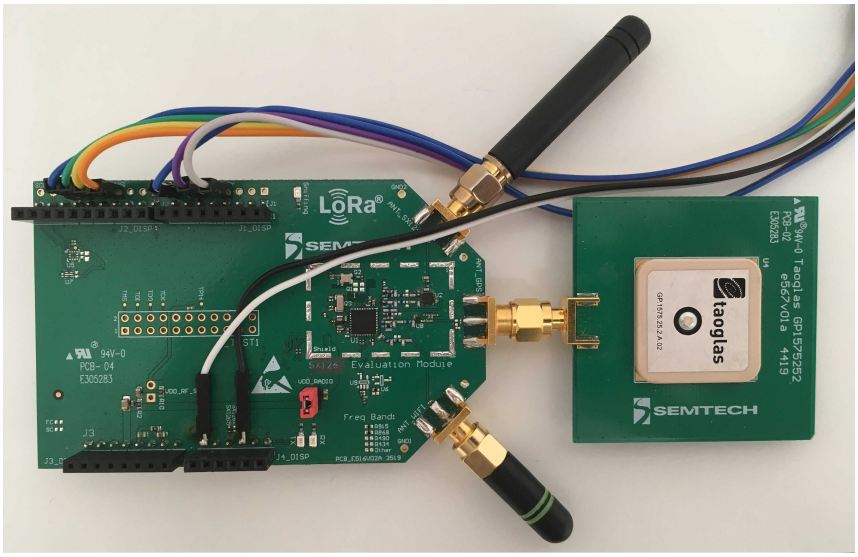

Figure 2.6: LR1110 development kit

2.1.4 Boost converter evaluation kit

Power of the Nucleo-F767ZI board and the FLIR camera is provided by the MAX17225ENT+T boost converter chip. Breakout board containing the chip is shown in Figure 2.7. Operating the boost converter chip is simple, its enable line can be directly connected to a microcontroller pin, driving it high will enables output and driving it low disables it. The output voltage is controlled by an external resistor.

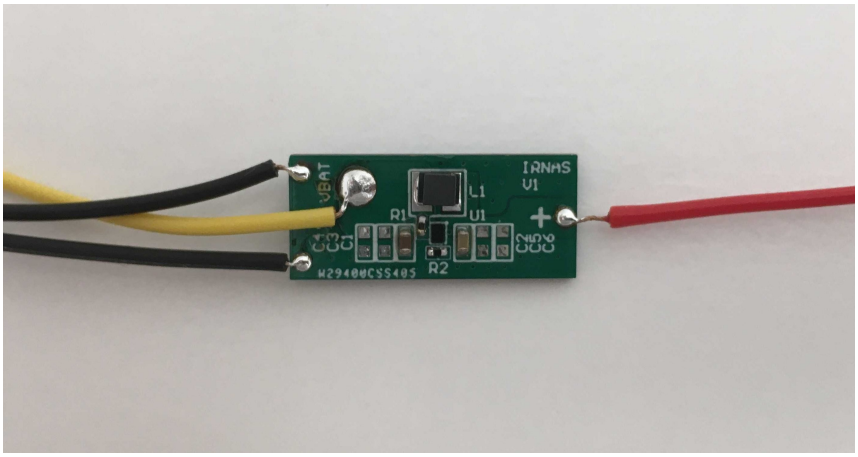

Figure 2.7: MAX17225ENT+T boost converter breakout board

2.1.5 FLIR Lepton 2.5 camera module and Lepton breakout board

Section ?? described what kinds of thermal cameras exist and how do they work, and Section ?? described why FLIR Lepton 2.5 was chosen. However, not much was said about what sort of support circuitry FLIR camera needs and how do we make images with it.

FLIR Lepton camera is powered from two different sources, 1.2 V and 2.8 V, and requires a reference clock of 25 MHz. All of this is provided by the Lepton breakout board, which can be seen in Figure 2.8. The front side of the breakout board contains a FLIR module socket and the backside contains two voltage regulators and an oscillator. Breakout board can be powered from 3.3 to 5 V and also conveniently breaks out all communication pins in form of header pins.

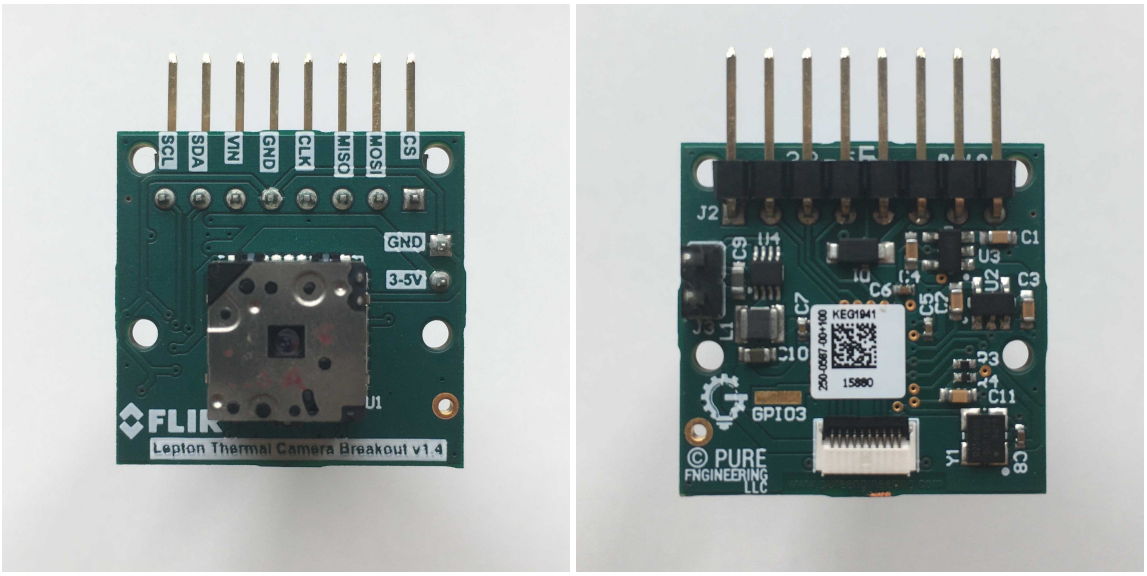

Figure 2.8: Front and back side of FLIR Lepton breakout board with thermal camera module inserted.

FLIR Lepton module itself contains five different subsystems that can be configured:

- AGC – Automatic Gain Control, affects image contrast and quality
- SYS – System Information
- VID – Video Processing Control

- OEM – Camera configuration for OEM customers
- RAD – Radiometry

Task of AGC subsystem is to convert a dynamic range of an IR sensor into a compact range that is more suitable for storing and displaying images. In the case of FLIR Lepton, this is a 14-bit to 8-bit conversion. For our purposes, the AGC subsystem was turned on, as the input to our image classification model were 8-bit values.

The microcontroller communicates with FLIR camera over two interfaces: two-wire interface (TWI) is used for control of the FLIR camera and Lepton's VoSPI protocol is used for image transfer.

2.1.6 PIR Sensor

We used a cheap, generic PIR sensor, that can be seen in Figure 2.9. It has two potentiometers, which are used to adjust sensor's sensitivity and detection delay. PIR sensor runs on 3.3 V, which enables us to directly power it from the nRF52.

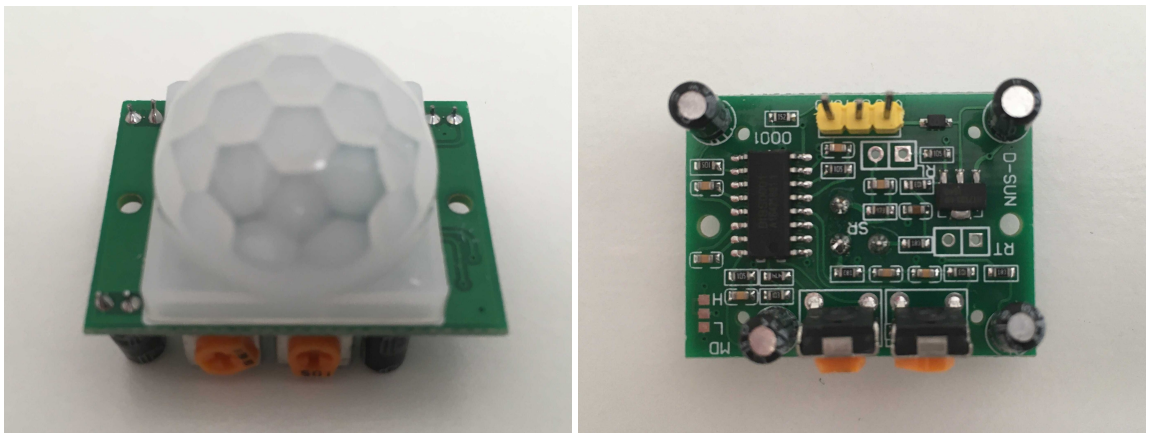

Figure 2.9: Front and back side of a PIR sensor.

2.2 Firmware

2.2.1 Tools and development environment

For our firmware development we did not choose any of the integrated development environments, provided by different vendors. Instead we used terminal text editor

Vim for writing and editing the code.

As we were programming two different microcontrollers, we were using different tools for each one.

2.2.1.1 Development environment for STM32f767ZI

For building our firmware programs we used GNU Make, build automation that builds software according to user written *Makefiles*. To compile code we used the Arm embedded version of GNU GCC. To program binaries into our microcontroller we used OpenOCD.

For hardware abstraction library we used libopencm3, which is an open-source low-level library that supports many of Arm's Cortex-M processors cores, which can be found in a variety of microcontroller families such as ST's STM32, Toshiba's TX03, Atmel's SAM3U, NXP's LPC1000, Silabs's EFM32 and others. Libopencm3 provided us with linker files, startup routines, thinly wrapped peripheral drivers and a starting template makefile, which served as a starting point for our project.

As libopencm3 does not provide `printf` functionality out of the box we used an excellent library by GitHub user mpaland [1].

2.2.1.2 Development environment for nRF52840

To develop firmware for nRF52 we decided to use The Zephyr OS, which is a small kernel, designed for IoT embedded systems. Besides usual RTOS functionalities such as tasks, mutexes, semaphores it also provides a common driver API for supported microcontrollers.

2.2.2 Architecture design

STM32 firmware was designed to be very efficient and lean, only truly necessary parts of firmware were implemented.

As seen in Figure 2.10 we split the firmware into two hardware and application modules.

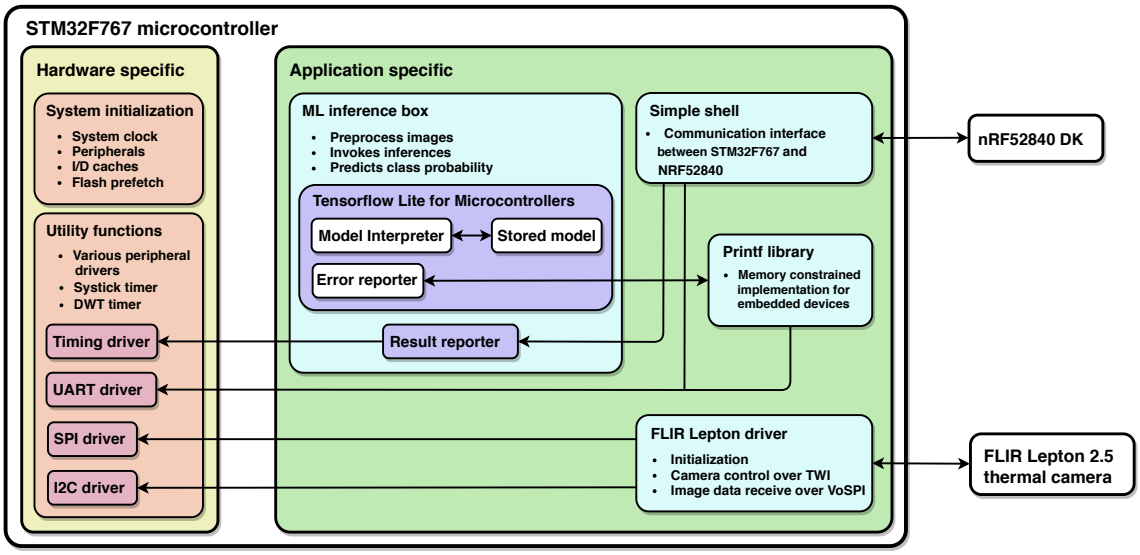

Figure 2.10: Architecture diagram of the firmware that is running on a STM32 microcontroller.

Hardware specific module is mostly using libopencm3 API to set the system clock and initialize peripherals. Small function wrappers had to be written to make use of various peripheral drivers easier.

FLIR Lepton libraries provided by the camera manufacturer or open-source communities were too complex and implemented way too many features that we did not need. We wrote FLIR Lepton driver from scratch, while reusing some concepts from official manufacturer's library.

Thanks to TFLite Micro API, the ML inference module could be written as a simple black box. Image data goes in, predictions come out.

The architecture diagram for nRF52 can be seen on Figure 2.11. For the nRF52 microcontroller we did not had to write any peripheral drivers, as they were provided by Zephyr itself.

Utmost importance was achieving low power consumption. That meant that nRF52 had to spend most of its time in low-power mode, only waking up for regular system checks and PIR trigger signals. When PIR trigger signal would be received, the interrupt would wake up the nRF52, which would then enable a boost converter, thus enabling power to STM32 and FLIR Lepton camera.

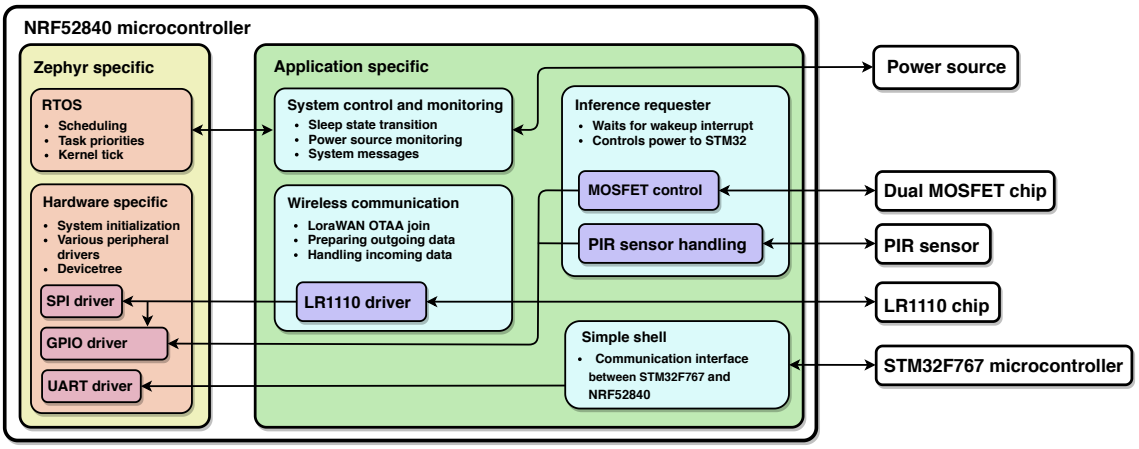

Figure 2.11: Architecture diagram of the firmware that is running on a nRF52840 microcontroller.

For the communication interface, we decided to implement a simple UART shell module. nRF52 microcontroller would act as a host and send commands to STM32. STM32 would execute commands and send back results. Once we knew that UART communication worked correctly, we could issue commands directly from the computer's serial port, which enabled us to develop and test firmware for STM32 separately from nRF52 firmware.

We also wrote a communication module, which took care of controlling the LR1110 chip, joining the LoRaWAN network, preparing outgoing messages and sending them over the LoRaWAN network.

2.2.3 FLIR Lepton driver

As mentioned before, FLIR Lepton driver had to implement two different protocols to control FLIR Lepton camera: TWI for general camera control and VoSPI for receiving images.

TWI is a variation of an I2C protocol, instead of 8 bits all transfers are 16 bits. The internal structure of the Lepton's control block can be seen in Figure 2.12. Whenever we are communicating with the FLIR camera we have to specify which subsystem are we addressing, what type of action we want to do (get, set or run), length of data and data itself.

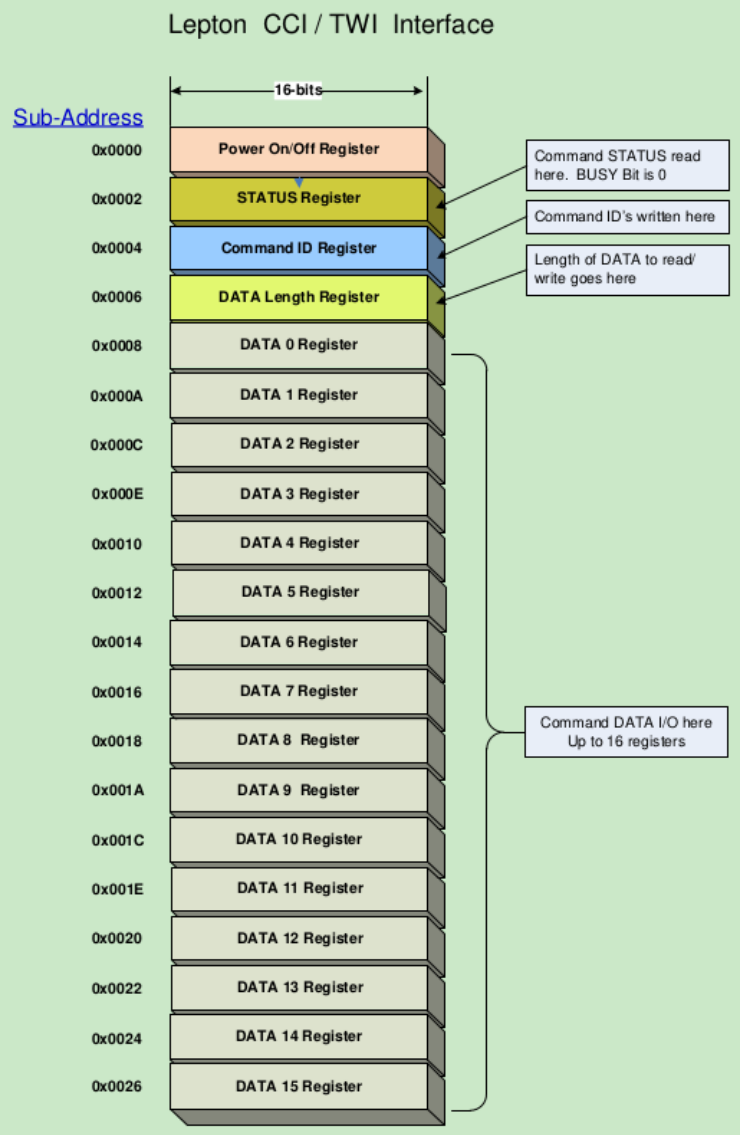

Figure 2.12: Command and control interface of FLIR Lepton camera.

We wrote the driver in such a way that API hid low-level details of exact data transfers. Two examples of such API can be seen in Figure 2.13.

Lepton's VoSPI protocol (which is variation of SPI protocol) is used only to stream image data from the camera module to the microcontroller, which means that the MOSI line is not used. Each image fits into one VoSPI frame and each frame consists of 60 VoSPI packets. One VoSPI packet contains 2 bytes of an ID field, 2 bytes of a CRC field and 160 bytes of data[1], which represents one image line. ID field of a valid VoSPI packet contains the number of equivalent frame row. The refresh rate of VoSPI frames is 27 Hz, however, only every third frame is unique from the last one.

```
/*!
 * @brief               Function sets position of shutter
 *
 * @param[in] position
 */
void set_flir_shutter_position(SHUTTER_POSITION position)
{
    if(!set_flir_command32(command_code(SHUTTER_POSITION,
                           LEP_I2C_COMMAND_TYPE_SET),
                           (uint32_t) position))
    {
        flir_print("Set shutter position : function failed!\n");
    }
}

/*!
 * @brief               Enable or disable AGC processing
 *
 * @param[in] position  If true AGC will be enabled
 */
void set_flir_agc(bool enable)
{
    if(!set_flir_command32(command_code(AGC_ENABLE_STATE,
                           LEP_I2C_COMMAND_TYPE_SET),
                           (uint32_t) enable))
    {
        flir_print("AGC mode: function failed!\n");
    }
}
```

Figure 2.13: Examples of FLIR Lepton driver API.

It is the job of the microcontroller to control the SPI clock speed and process each frame fast enough so that the each unique frame is not discarded.

Figure 2.14 shows our implementation of a `get_picture` function that was reading images from VoSPI protocol. To capture images from VoSPI stream we implemented finite state machine with three states: `INIT`, `OUT_OF_SYNC` and `READING_FRAME`. State `INIT` executes chip select sequence expected by the FLIR camera. After it, we start clocking out a stream of VoSPI frames over MISO line. Whenever we start communication, we did not know where exactly in stream are we, plus FLIR is also transmitting discard packets between valid frames. To solve this problem we have to check ID field of every VoSPI packet and look for a ID byte with a value `0x00`, while

[1]Because images pixel values fit into the 14-bit range by default, it means that one-pixel value needs two bytes of data (two most significant bytes are zero). That means that each image line (80 pixels) is stored in 160 bytes. If AGC conversion is turned on, each pixel is then mapped into an 8-bit range, however, the size of one line in the VoSPI packet remains 160 bytes, 8 most significant bits are simply zeros.

discarding packets with values `0xFF`. When first frame row is detected we simply start storing all incoming frame rows into `frame` variable, while checking that ID byte is correct. We continue doing until all VoSPI packets of a single frame are received. If we somehow miss the VoSPI packet, we returned early from the function.

2.2.4 Simple shell

2.2.5 MicroML and build system

A large part of this thesis was concerned with porting TFLite Micro to libopencm3, our platform of choice. To understand how this could be done, we first had to analyse how the code is built in TFLite Micro.

To compile source files and build binaries TFLite Micro uses GNU Make. The main makefile includes several platform specific makefiles, which dictate how firmware is built, and several bash scripts which download various dependencies. By providing command-line arguments users decide which example needs to be compiled and for which platform. The build system makes some assumptions about locations of the platform specific files, which in case of example projects are scattered over the whole TensorFlow GitHub repository.

We learned a useful principle while analysing the build process. Each time we would build an example for a new platform, Make would first compile all TensorFlow files, create a static library out of them, compile specific example source files and then link against the library in linking stage. If we wanted to build firmware for a different example, but for the same platform, Make would only have to compile source files of that example and link them with previously made library. As compilation of required TensorFlow files takes quite some time, this was an efficient option.

After analysing the TFLite Micro's build system we created a list of requirements that we wanted to fulfil on our platform.

1. We wanted to keep project-specific code, libopencm3 code and TFLite Micro

code separated.

2. We wanted a system, where it would be easy to change a microcontroller specific part of a building process.

3. We wanted to reuse static library principle that we saw in TFLite Micro build process.

Covering different platforms and use cases made main TFLite Micro makefile quite complex and hard to understand. This meant that it would be hard to reuse it while porting to a new platform and we needed a different approach or reuse something else.

To solve our problem we started developing a small project that we named MicroML[2]. MicroML enables users to develop ML applications on microcontrollers supported by libopencm3. Project's directory structure can be seen in Figure 2.15

Folders `tensorflow` and `libopencm` are directly cloned from their respective sources as Git submodules, which means that they are fixed at specific commits, usually at major release points. In folder `projects` users place all their specific projects. Besides source files, each project has to contain three specific files:

- **project.mk** - It contains information which files need to be compiled inside the project folder. In it is defined for which microcontroller the code needs to be compiled and what kind of optimisation flags should be used.
- **openocd.cfg** - Configuration file that tells OpenOCD which programmer interface needs to be used to flash a microcontroller and the location of the binary file that needs to be flashed.
- **Makefile** - Project's makefile that gathers source files inside the project folder. It makes it possible to call `make` directly from projects directory, which eases the development process. It does not specify any building rules, those are specified in included `rules.mk` file in the root directory of the project.

[2]Project is open-source and publicly available on GitHub [2].

Some initial commands need to be executed when the project is cloned from the GitHub for the first time. Figure 2.16 represents the complete build process.

In *submodules setup* stage we first compile both of the submodules, this step requires two makefiles that are already provided by each submodule. Compiling libopencm3 creates a group of archive files (static libraries), which contain all platform specific code. Compiling a TensorFlow Hello World example does not produce any archive files that we would need, however, it does execute several scripts which download several different third party files. TFLite Micro library depends on these files, which means that MicroML does as well. *Submodules setup* stage only has to be executed once.

Whenever we start with a new project that will use ML algorithms, we need to go through *project setup* stage. From main directory, we call `make` command with `archive_makefile` and define `PROJECT` variable with the name of our project. `Archive_makefile` looks into `project.mk` and extracts `DEVICE` variable. Libopencm3's `genlink-config.py` script then determines which microcontroller specific compilation flags[3]are needed. All needed TensorFlow source files and third party files are then compiled and a project specific `microlite.a` archive file is created in our project's folder.

Compile and flash stage is then continuously executed during the development process. By calling `make flash` directly in our project folder we compile all project files, `microlite.a` and libopencm3 archive files that were created early. Libopencm3 helper scripts (`genlink-config.mk` and `genlink-rules.mk`) provide us with the microcontroller specific flags and a linker script. After compilation a `firmvare.elf` is created, Make then automatically calls OpenOCD, which flashes a microcontroller.

As flashing a big binary to a microcontroller takes a long time, we also created a similar setup for testing inference directly on the host machine. That way we could test ML specific routines fast and quickly remove any mistakes found on the

[3]For example, to compile firmware for STM32 we need flags `-mcpu=cortex-m7`, `-mthumb`, `-mfloat-abi=hard` and `-mfpu=fpv5-sp-d16`.They tell GCC compiler that we are compiling for a cortex-m7 processor, that we want to use thumb instruction set and that we want to use hardware floating-point unit with single precision.

way.

2.2.6 Running inference on a microcontroller

TFLite Micro API is fairly simple to use and general enough that it can be copied from project to project without many modifications. Figure 2.17 shows a simplified inference code example, copied from our project.

```
bool get_picture(uint16_t frame[60][82])
{
    state_e state = INIT;
    uint8_t frame_row = 0;
    while(1)
    {
        switch(state)
        {
            case INIT:
                enable_flir_cs();
                disable_flir_cs();
                delay(185);
                enable_flir_cs();
                state = OUT_OF_SYNC;
            break;

            case OUT_OF_SYNC:
                spi_read16(frame[frame_row], 82);
                // Look for the start of the frame
                if ((frame[frame_row][0] & 0x00FF) == 0x0)
                {
                    // Start of frame detected
                    frame_row++;
                    state = READING_FRAME;
                }
            break;

            case READING_FRAME:
                spi_read16(frame[frame_row], 82);
                // Check each frame row
                if((frame[frame_row][0] & 0x00FF) == frame_row)
                {
                    // Frame row matches
                    frame_row++;
                    if (frame_row == 60)
                    {
                        // Full frame received, return to caller
                        disable_flir_cs();
                        return true
                    }
                }
                else
                {
                    // Error, end image reading
                    disable_flir_cs();
                    return false;
                }
            break;
        }
    }
}
```

Figure 2.14: Finite state machine implementation for reading FLIR images over SPI.

```
MicroML
├── tensorflow
├── libopencm3
├── projects
│   ├── hello_world_stm32f7
│   └── elephant_stm32f7
│       ├── test
│       ├── src
│       ├── Makefile
│       ├── project.mk
│       └── openocd.cfg
├── archive_makefile
└── rules.mk
```

Figure 2.15: Directory structure of MicroML project.

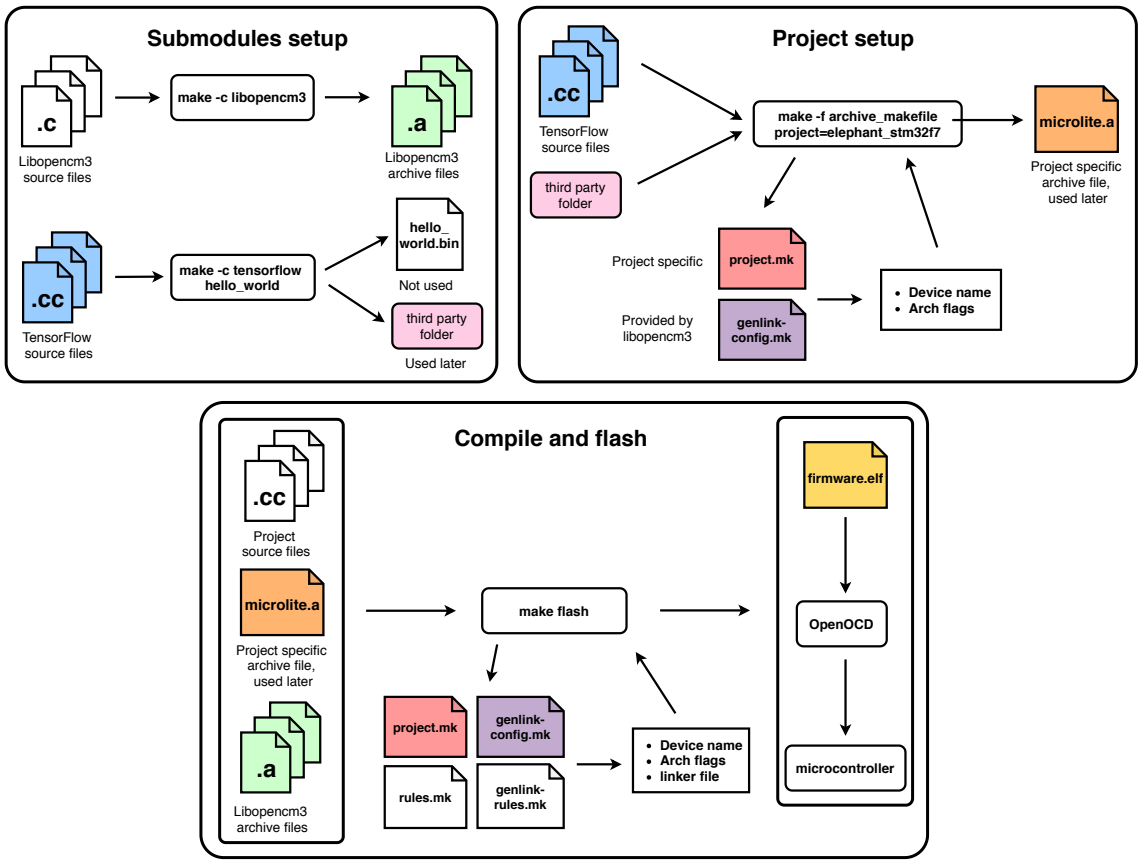

Figure 2.16: Build system of MicroML project.

```cpp
// An area of memory to use for input, output,
// and intermediate arrays.
const int kTensorArenaSize = 200 * 1024;
static uint8_t tensor_arena[kTensorArenaSize];

int main()
{
    // Debug print setup
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter *error_reporter = &micro_error_reporter;

    // Map the model into a usable data structure
    const tflite::Model* model = tflite::GetModel(full_quant_tflite
   );

    // Pull in needed operations
    static tflite::MicroMutableOpResolver<8> micro_op_resolver;
    micro_op_resolver.AddConv2D();
    micro_op_resolver.AddMaxPool2D();
    micro_op_resolver.AddReshape();
    micro_op_resolver.AddFullyConnected();
    micro_op_resolver.AddSoftmax();
    micro_op_resolver.AddDequantize();
    micro_op_resolver.AddMul();
    micro_op_resolver.AddAdd();

    // Build an interpreter to run the model with.
    static tflite::MicroInterpreter interpreter(model,
                                                micro_op_resolver,
                                                tensor_arena,
                                                kTensorArenaSize,
                                                error_reporter);
    // Allocate memory from the tensor_arena
    interpreter->AllocateTensors();

    // Get information about the memory area
    // to use for the model's input.
    TfLiteTensor* input  = interpreter->input(0);
    TfLiteTensor* output = interpreter->output(0);

    // Load data from image array
    for (int i = 0; i < input->bytes; ++i) {
        input->data.int8[i] = image_array[i];
    }

    // Run the model on this input and time it
    uint32_t start = dwt_read_cycle_counter();
    interpreter->Invoke();
    uint32_t end   = dwt_read_cycle_counter();

    // Print probabilites and time elapsed
    print_result(error_reporter, output, dwt_to_ms(end-start));
}

```

Figure 2.17: Example of TensorFlow Lite inference code in C++.

As a first step, we need to define the size of `tensor_arena` array, which holds memory of input, output, and intermediate arrays. Exact size of `tensor_arena` is determined by trail and error: we set it to some big value and then decrease it in steps, until the code does not work any more.

In lines 9 and 10 we create an instance of `ErrorReporter` object. This object serves as a thin wrapper around platform specific `printf` implementation. If some part of TensorFlow code crashes, `ErrorReporter` notifies us what went wrong.

In the line 13 we pull in our ML model in hex dump format that we created with xxd. `Full_quant_model` is defined in a different file, not seen in this example.

In lines 16 to 24 we create an operation resolver. One way to do it is to specify each required operation specifically (which is done in the example) or simply pull in all operations. Latter approach is not recommended, as it results in large binary size. To find out exactly which operations were required we used online tool Netron [3], which showed us a deconstructed view of a trained model.

In lines 27 and 33 we create an `MicroIinterpreter` instance and allocate memory to it that we specified with `tensor_arena` earlier. Lines 37 and 38 assign input and output of the interpreter to the new `TfLiteTensor` variables. This step enables us to do two things. Firstly, variables `input` and `output` now point to information about data format: We can found out how many dimensions are needed, what is the size of those dimensions and what is expected type of variable (`uint8_t`, `int8_t`, `float`...). In tests that we were running on the laptop, we tested exactly for these values to confirm that the model works as expected. Secondly, we now have a way to directly feed data into input, this is done in for loop on line 41. One of `TfLiteTensor` members is a union variable `data` which contains variables of all possible types. This type of structure enables us to load input with any kind of data, in our case `int8`.

In line 47 we finally invoke interpreter and run inference on input data. The whole expression is surrounded by the timing functions, which are used to keep track of time spent computing inference.

We finally call `print_results`, written by us, where we pass `error_reporter` for printing, `output` for extracting computed probabilities and elapsed time.

After the initial setup, we can load data, call invoke, and print results as many times we want.

2.3 Server-side components and software

In this section, we describe possible server-side construction of various frameworks which enable us to receive LoRaWAN messages, parse them, store them in a database and visualise them. We did not implement this specific setup as it was not required for testing purposes, however, at IRNAS we use this setup constantly for our IoT products and implementation of a such system would be trivial.

The system that we use consists of different tools, each one with a distinct task. These tools are The Things Network (TTN), Node-RED, InfluxDB and Grafana. The flow of information and tasks for each tool is presented in Figure 2.18.

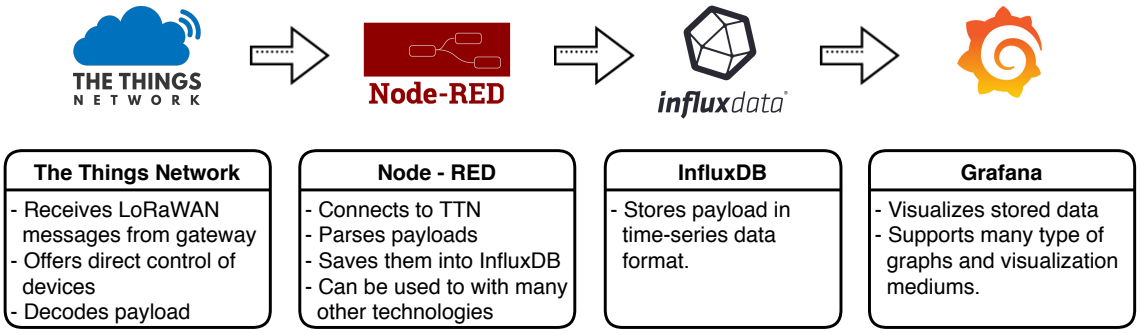

Figure 2.18: Server side flow of information. Icons source: [4]

TTN is responsible for routing packets that are captured by a gateway to the application server. Since it is open-source and free, anyone can register their gateway device into the network and thus helping to extend it. TNN is web-based, so we can see payload messages directly in the browser. Since data is usually encoded in binary format, we can provide a decoder-script written in JavaScript and TTN will automatically pass each message, thus decoding it.

Node-RED functions as a glue logic that parses packets and shapes them into a

format that is required by InfluxDB. Node-RED provides a browser-based flow editor, where actual programming can be done graphically. Logic is programmed by choosing different blocks called *nodes* and connecting them. This is convenient, as Node-RED provides different nodes for communicating with different technologies, such as MQTT, HTTP requests, emails, Twitter accounts and others. In our use case, we needed to use nodes seen in Figure 2.19. Node *Elephant Gateway* is connected to a specific application on TTN, which is used for the collection of packets from our devices in the field. Any packet that will appear in that TTN application will also appear in Node-RED. Node *Parse packet* extracts the information contained in each packet and stores it in a specific format, which is finally sent to the node *Elephant Database*.

Figure 2.19: Node-RED flow

Elephant Database is connected to InfluxDB, which acts as a time-series database. Any packet that is saved in it is automatically timestamped.

Data is then visualized in Grafana. Grafana is an open-source analytics and monitoring solution. Users define which database is set as a source and Grafana provides graphical controls which are at some point converted into SQL-like language, understandable to InfluxDB. Grafana provides different types of visualizations, such as graphs, gauges, heat maps, alert lists and others. In our use case, we could display information about various devices in the field, such as battery voltage, number of wakeup triggers, results of each inference and others.

Example of Grafana graph can be seen in Figure 2.20.

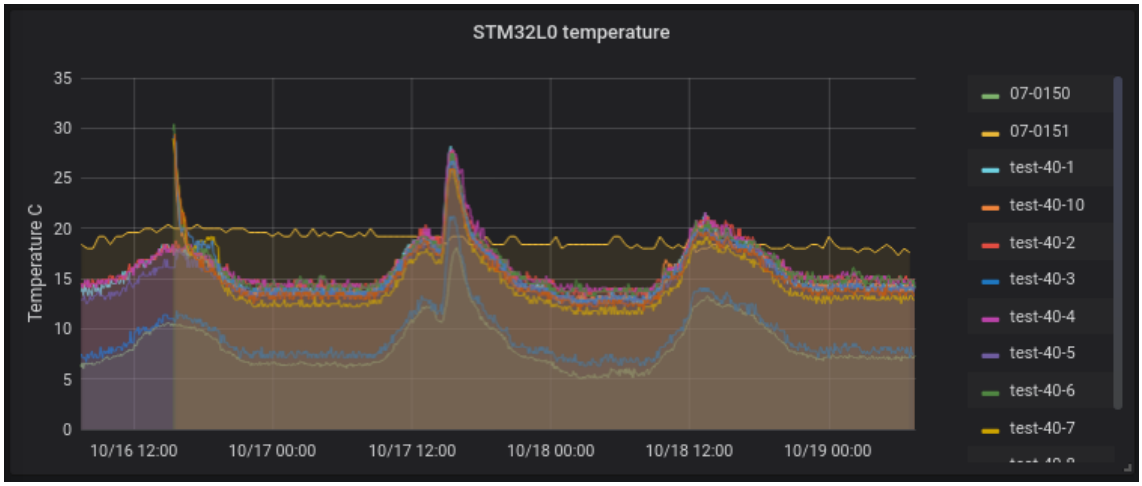

Figure 2.20: Example of Grafana graph.

One important quality of Node-RED, InfluxDB and Grafana is that they can run directly on an embedded Linux system, such as Raspberry Pi or BeagleBone, which greatly lowers the cost of hardware that is needed.

3 Measurements and results

3.1 Comparison of models

As mentioned in Section **??**, we used Keras Tuner model to find hyperparameters that would yield the highest accuracy. Instead of hard-coding hyperparameters when building a model with Keras API, we defined a search space of possible values with `HyperParameter` class and used that as a hyperparameter.

We passed the created model to a `RandomSearch` class, with few other parameters such as batch size, number of epochs and maximum number of trials. As we started the hyperparameter search, Keras Tuner started picking a randomly set of hyperparameters, which were used to train a model. This process was repeated for a trial number of times. Used hyperparameters and achieved accuracy on the validation set for each trained model were logged in a text file for later use.

After training several different models we picked a few and compared them. Comparison of models trained in Edge Impulse Studio was also done.

3.1.1 Hyperparameter search space and results analysis

General structure of CNN model was already described in Section **??** and in Figure **??**. We decided to search for the following hyperparameters:

- Number of filters in all three convolutional layers (can be different for each layer)
- Size of filters in all three convolutional layers (same for all layers)
- Size of the dense layer

- Dropout rate
- Learning rate

Possible values of hyperparameters (also known as hyperparameter search space) are specified in Table 3.1.

Table 3.1: First hyperparameter search space

Hyperparameter	**Set of values**
`FilterNum1`	From 16 to 80, with a step of 8
`FilterNum2`	From 16 to 80, with a step of 8
`FilterNum3`	From 16 to 80, with a step of 8
`FilterSize`	3 x 3 or 3 x 4
`DenseSize`	From 16 to 96, with a step of 8
`DropoutRate`	From 0.2 to 0.5, with a step of 0.05
`LearningRate`	0.0001 or 0.0003
Random search variable	**value**
`EPOCHS`	25
`BATCH_SIZE`	100
`MAX_TRIALS`	300

Search space of `FilterNumX`, `DenseSize` and `DropoutRate` hyperparameters were chosen based on initial training tests conducted on a thermal image dataset and other various models that were trained on similar data. Value of `FilterSize` is usually 3 x 3, however, most of example ML projects that we could find on the Internet were training on image data of the same dimensions. We wanted to test how would a filter with the same ratio of dimensions as image data (3 x 4 and 60 x 80 respectively) perform. Hyperparameter `learning_rate` was chosen heuristically, we saw that 10 times higher values, such as 0.001 or 0.003, would leave model's accuracy stuck at suboptimal optima, from where it could not be improved anymore.

We also had to set 3 variables that directly affected how long will random search last. From initial tests we saw that models usually reached maximum possible accuracy around 20^{th} epoch, to give some headroom we set the number of epochs to 25. We kept batch size relatively small, at 100, which meant that weights would get updated

regularly. Hyperparameter `MAX_TRIALS` had the biggest impact on the training time, we set it to 300.

The training lasted for about 12 hours. After it was done we compiled a list of all 300 trained models and their different hyperparameter values, number of parameters and achieved accuracies Part of it can be seen in Table 3.2.

Table 3.2: Partial results of first random search of hyperparameters

Model ID	FilterNum1	FilterNum2	FilterNum3	DenseSize	DropoutRate	FilterSize	LearningRate	Number of parameters	Accuracy[%]
0a	72	80	64	72	0.4	3x4	0.0003	1,514,400	98.35
1a	32	40	72	56	0.35	3x4	0.0001	1,260,332	98.31
2a	40	48	32	64	0.35	3x4	0.0001	656,797	98.31
3a	56	16	48	72	0.4	3x4	0.0001	1,057,924	98.28
4a	80	64	40	96	0.45	3x4	0.0003	1,245,788	98.28
96a	16	32	72	80	0.25	3x4	0.0001	1,762,508	98.00
97a	72	56	40	56	0.45	3x4	0.0003	748,580	98.00
98a	32	24	24	48	0.35	3x3	0.0001	358,308	98.00
99a	48	16	40	40	0.45	3x3	0.0003	493,412	98.00
100a	24	72	64	40	0.45	3x3	0.0003	844,684	98.00
191a	64	56	16	52	0.4	3x3	0.0001	386,996	97.76
192a	48	40	24	24	0.4	3x4	0.0001	208,172	97.73
193a	56	64	72	24	0.25	3x4	0.0003	617,692	97.73
194a	48	72	48	32	0.25	3x4	0.0003	544,652	97.73
295a	48	32	64	16	0.5	3x4	0.0001	351,012	95.87
296a	40	24	56	24	0.5	3x4	0.0001	431,572	95.77
297a	56	16	80	16	0.2	3x4	0.0001	411,020	95.63
298a	24	16	48	24	0.5	3x4	0.0001	359,924	94.46
299a	40	48	56	16	0.35	3x3	0.0003	310,860	82.86

After analyzing results we came to several conclusions:

1. We saw that almost all trained models, except for the last one, achieved accuracy above 90 %. This proved that the general architecture of the model was appropriate for the problem.

2. We could not see any visible correlation between a specific choice of a certain

hyperparameter and accuracy. This showed that selection of hyperparameters is a non-heuristic task, at least for our particular problem.

3. Filter of size 3 x 4 did not perform significantly better compared to one with size 3 x 3.

4. There is a weak correlation between the number of parameters (model's complexity) and accuracy, however, models with small size and great accuracy exist, model *2a* is an example of that.

5. First 200 models cover an accuracy range of 0.62 %. However inside of this range model number of parameters varies hugely, for example, model *192a* has more than 8 times fewer parameters than model *96a*, although the difference in accuracy (0.27 %) is negligible.

It is apparent from results that large models are not necessary to achieve high accuracy on our training data, so we decided to run the random search of hyperparameters again.

This time we lowered the maximum and the minimum numbers of filters and size of the dense layer. We decreased all steps from 8 to 2, thus increasing the number of possible configurations. We decided to lower the bottom boundary of `DropoutRate` from 0.2 to 0.0, which means that some models will not be using dropout layer at all. We expected that training without dropout layer would produce suboptimal results, however, we wanted to test it. Redefined search space for second random search can be seen in Table 3.3 We increased the number of `MAX_TRIALS` from 300 to 500, as we were expecting that more models will end up underfitting and also because there would be more possible options because of smaller step size. Partial table of results of random hyperparameter search can be seen in Table 3.4.

Table 3.3: Second hyperparameter search space

Hyperparameter	**Set of values**
`FilterNum1`	From 4 to 48, with a step of 2
`FilterNum2`	From 4 to 48, with a step of 2
`FilterNum3`	From 4 to 48, with a step of 2
`FilterSize`	3 x 3 or 3 x 4
`DenseSize`	From 4 to 48, with a step of 2
`DropoutRate`	From 0.0 to 0.5, with a step of 0.05
`LearningRate`	0.0001 or 0.0003
Random search variable	**value**
`EPOCHS`	25
`BATCH_SIZE`	100
`MAX_TRIALS`	500

Table 3.4: Partial results of second random search of hyperparameters

Model ID	FilterNum1	FilterNum2	FilterNum3	DenseSize	DropoutRate	FilterSize	LearningRate	Number of parameters	Accuracy[%]
0b	40	20	20	48	0.25	3x4	0.0001	304,216	98.14
1b	44	10	28	42	0.2	3x4	0.0003	362,264	98.14
2b	18	38	26	38	0.1	3x4	0.0003	316,956	98.11
95b	20	16	34	40	0.3	3x3	0.0003	416,230	97.62
96b	46	42	28	32	0.4	3x3	0.0003	297,466	97.62
97b	30	26	30	34	0.2	3x3	0.0001	320,570	97.59
195b	28	16	40	24	0.1	3x3	0.0001	298,252	97.31
196b	44	30	32	20	0.3	3x4	0.0003	220,098	97.31
197b	46	40	10	40	0.1	3x3	0.0001	140,874	97.31
295b	20	8	34	26	0.3	3x3	0.0003	269,464	96.90
296b	18	16	10	20	0.3	3x4	0.0003	65,740	96.87
297b	8	22	28	16	0.1	3x3	0.0001	141,742	96.87
395b	10	20	12	30	0.0	3x3	0.0001	112,246	96.87
396b	24	24	46	18	0.2	3x3	0.0003	263,924	96.14
397b	6	18	12	24	0.4	3x4	0.0001	90,520	96.11
497b	42	30	22	6	0.4	3x3	0.0003	57,386	82.86
498b	4	4	20	12	0.4	3x3	0.0003	72,992	82.86
499b	32	36	36	4	0.15	3x3	0.0001	65,648	82.86

Some observations:

1. We can see that the accuracy of the best model *0b* compared to the best model *0a* from the previous search is only 0.21 % lower, although it has about 5 times fewer parameters.

2. Although that it might seem that `FilterSize` of 3 x 4 yields best results, we did not saw a strong tendency towards 3 x 3 or 3 x 4 filter size after manually analyzing best 30 models.

3. We can see that the worst three models have the same accuracy of 82.86 %, same as the worst-performing model from the first random search. There are 82.86 % images of elephants in the validation class, which means that the model probably assigned all validation images to elephant class and was satisfied with achieved accuracy.

4. We can see that the model *296b* has a quite low number of parameters, only 65,740 when compared to its neighbours.

3.1.2 Comparison of selected, re-trained models

Two random searches gave us a large number of different models to choose from. In every other ML application where the execution time would not be a constraint, we could simply take the best performing model and be done with it. In our case, we had to make a trade-off between model's accuracy and execution speed.

For comparison and later on device performance testing we decided to pick and retrain[1]6 models: *0a*, *2a*, *0b*, *172b*, *338b* and *460b*, their properties are listed in Table 3.5.

Chosen models vary greatly in the number of parameters. Models *0a*, *2a*, *0b* have high number of parameters but their accuracy is high. Models *172b*, *338b* and *460b* were chosen because of their small size and reasonably good accuracy.

[1]Retraining was required as Keras Tuner module only saved hyperparameter settings during search and not each trained model. As the weights are initially randomized, the accuracy of retrained models is going to be similar but not exact when compared to the accuracy returned by random search.

Table 3.5: Properties of selected models

Model ID	FilterNum1	FilterNum2	FilterNum3	DenseSize	DropoutRate	FilterSize	LearningRate	Number of parameters	Accuracy[%]
0a	72	80	64	72	0.4	3x4	0.0003	1,514,400	98.35
2a	40	48	32	64	0.35	3x4	0.0001	656,797	98.31
0b	40	20	20	48	0.25	3x4	0.0001	304,216	98.14
172b	42	44	8	14	0.1	3x4	0.0001	60,672	97.38
338b	4	18	6	10	0.05	3x4	0.0003	20,290	96.63
460b	6	28	4	8	0.1	3x4	0.0003	13,114	93.60

As we are dealing with an imbalanced dataset, where 82.86 % of our validation data consists of elephant images, accuracy is not the best metric to use when comparing models. Simply classifying all images into elephant class would yield an accuracy of 82.86 %, which sounds high, although it would not actually do any classification.

When analysing the performance of a model on an imbalanced dataset it is more appropriate to use precision and recall metrics[2]. They can give us a better idea of how well the model is performing on data of specific classes. Calculated metrics can be seen in Table 3.6, we abbreviated precision to P and recall to R for clarity.

Table 3.6: Precision and recall metrics of trained models

Model ID	0a	2a	0b	172b	338b	460b
Metrics						
accuracy[%]	98.18	98.04	98.04	96.80	96.28	93.4
Number of parameters	1,515K	657K	304K	61K	20K	13K
P of elephant class[%]	99.22	99.46	99.25	99.29	98.80	97.80
P of human class[%]	96.92	95.38	95.38	92.00	91.69	80.31
P of cow class[%]	90.99	93.69	90.09	84.68	75.68	69.37
P of nature/random class[%]	77.42	64.52	79.03	46.77	59.68	40.32
R of elephant class[%]	99.29	98.80	98.84	97.87	98.43	97.39
R of human class[%]	93.20	94.51	95.09	91.44	89.22	85.57
R of cow class[%]	94.39	92.04	96.15	89.52	84.00	81.91
R of nature/random class[%]	87.27	97.56	84.48	93.55	67.27	28.09

[2]Precision tells us what percentage of data points in a specific predicted class fall into that class. Recall tells us what percentage of data points inside a certain class were actually predicted correctly [5].

As we can see all six models are generally classifying elephants correctly, both precision and recall of elephant class are high, above 97 %, which is important. Precision and recall values of other classes are generally lower, especially for nature/random. We can see that top three models *0a*, *2a* and *0b* are quite similar in terms of precision and recall, which means that we can easily prefer *0b*, without sacrificing accuracy. Models *172b* and *338b* perform a bit worse when compared to top three models, however, they have a low number of parameters which should translate to lower inference time. Last model, *460b*, performs the worst and it should generally not be used.

Another way to compare models performance is to look at a confusion matrix. Figure 3.1 shows comparison between confusion matrices of *0a* model on the left and *460b* model on the right. In the case of *0a*, 19 elephant images were not classified correctly, and 17 images were wrongly classified as elephants. This is not ideal, however is much better compared to performance of *460b*, where 53 elephants were wrongly classified and 63 of images classified as elephants were not actually elephants.

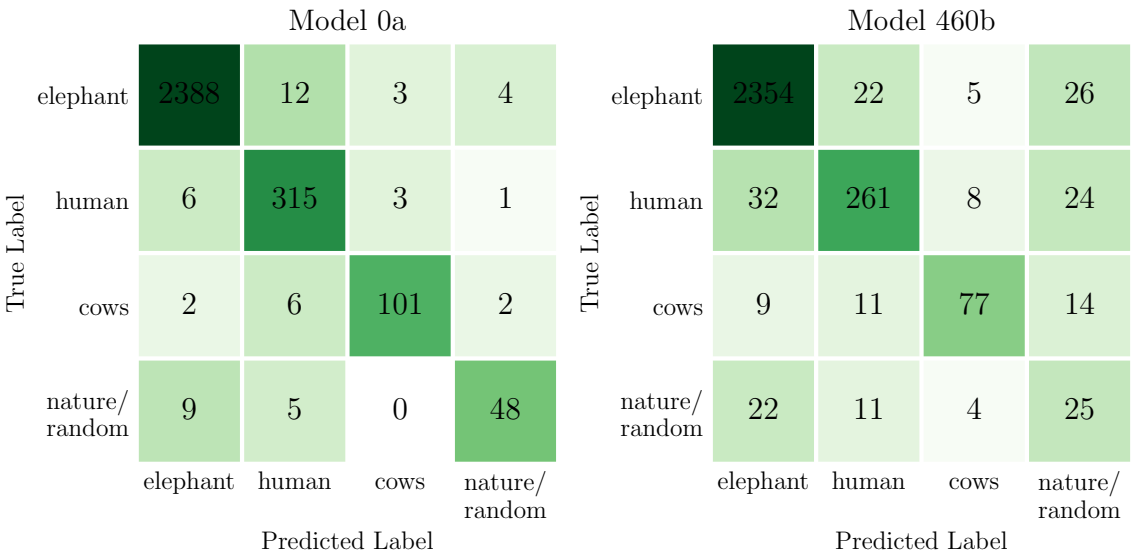

Figure 3.1: Confusion matrices of *0a* model (left) and *460b* model (right).

3.1.3 Comparison of Edge Impulse models

We wanted to take our 6 models and compare them against 6 Edge Impulse models that were created by using the same hyperparameters. However, at the time of writing Edge Impulse supported only model training on images of the same dimensions. Images with different dimensions could either be cropped or scaled to fit 1:1 ratio. Using the same hyperparameters in Edge Impulse Studio, that were used in our models, would always create a model with a smaller number of parameters. The smaller image creates a smaller network when compared to a bigger image, given that the rest of architecture does not change. That meant that we could not make a direct comparison between our models and models trained in Edge Impulse Studio. We also could not perform a random search of hyperparameters in Edge Impulse Studio, as this feature was not fully supported at the time of writing this thesis.

We decided to train a few differently sized models, using the same general CNN architecture as before, but with some minor changes in hyperparameter values. We also trained a few models with Transfer Learning technique. Edge Impulse offers scaled-down versions of pre-trained MobileNetV2[3]NN architecture, which we used.

Tables 3.7 and 3.8 show properties of Edge Impulse models using CNN architecture and Transfer Learning technique, respectively. Table 3.9 shows calculated precision and recall values of Edge Impulse models using both approaches.

We used only two different versions of MobileNetV2, 0.35, and 0.1, as we saw an accuracy drop in the reduction of width multiplier hyperparameter. In all cases the pre-trained model was followed by a one or two dense layers, with dropout layers in between.

[3]MobileNetV2 is a efficient, lightweight NN architecture, designed for image recognition tasks, suitable for mobile applications [5]. MobileNetV2 contains width multiplier hyperparameter, which scales up or down the total number of parameters, thus providing a trade-off between accuracy and computation complexity. Edge Impulse offers three different width multiplier options: 0.35, 0.1 and 0.05.

Table 3.7: Properties of Edge Impulse models using CNN architecture.

Model ID	FilterNum1	FilterNum2	FilterNum3	DenseSize	DropoutRate	FilterSize	LearningRate	Number of parameters	Accuracy[%]
0ei	72	80	64	72	0.4	3x4	0.0003	1,168,804	97.7
1ei	40	48	32	64	0.35	3x4	0.0001	503,196	97.5
2ei	40	20	20	48	0.25	3x4	0.0001	231,204	97.3
3ei	42	44	8	14	0.1	3x4	0.0003	52,272	96.6

Table 3.8: Properties of Edge Impulse models using Transfer Learning technique

Model ID	Width Multiplier	DenseSize1	DenseSize2	DropoutRate	LearningRate	Number of parameters	Accuracy[%]
0tl	0.35	16	N/A	0.1	0.0005	430,676	98.5
1tl	0.35	16	16	0.1	0.0005	430,948	98.4
2tl	0.35	32	32	0.1	0.0005	452,484	98.7
3tl	0.1	32	32	0.1	0.0005	135,732	95.7

Table 3.9: Precision and recall metrics of trained Edge Impulse models

Model ID	0e	1e	2e	3e	0tl	1tl	2tl	3tl
Metrics								
Accuracy[%]	97.7	97.5	97.3	96.6	98.5	98.4	98.7	95.7
Number of parameters	1,169K	503K	231K	52K	430K	431K	452K	136K
P of elephant class[%]	99.69	99.53	99.42	99.27	99.27	99.42	99.48	98.65
P of human class[%]	95.05	95.05	91.87	91.52	97.17	95.41	96.47	85.16
P of cow class[%]	82.22	78.89	82.22	79.57	92.22	91.01	92.22	75.56
P of nature/random class[%]	63.04	65.22	73.91	50.0	86.96	89.13	91.3	78.85
R of elephant class[%]	99.86	98.91	98.44	98.65	99.48	99.27	99.37	98.03
R of human class[%]	93.4	92.44	94.2	88.4	94.5	95.74	95.79	90.26
R of cow class[%]	90.24	86.59	84.09	79.57	92.22	89.01	94.32	67.33
R of nature/random class[%]	87.88	88.24	94.44	95.83	95.24	97.62	95.45	91.11

Some observations:

- Models using CNN architecture did not out perform our models in terms of accuracy. Models *2a* and *0b* both had accuracy of 98.04 %, while none of Edge Impulse models with CNN architecture did not pass 98 %.
- Most of the models trained with Transfer Learning technique outperformed our models.
- Model *2tl* performed exceptionally well, reaching an accuracy of 98.7 % while having a relatively small number of parameters.
- We saw that by decreasing width multiplier we did not benefit much in accuracy as much as we lost in model size. Even increasing the sizes of dense layers did not solve the problem.

3.2 On-device performance testing

Performance testing of all models was done on an STM32F767ZI microcontroller, running at 216 MHz. Testing of our models was done by using MicroML framework that we wrote, which directly called TFLite Micro API. Testing of Edge Impulse models was done on Mbed OS, as this platform is supported by Edge Impulse and they already provide an example for it. We could not test model *0a* on device as TFLite converter failed to produce a compilable model.

To time the execution of our code we used Data Watchpoint Trigger (DWT) which contains a counter that is directly incremented by the system clock. DWT does not use interrupts, therefore it does not introduce the overhead of calling interrupt routines as systick timer does.

Edge Impulse provides examples for testing out of the box, so not much work is needed to get the first-order approximation of performance. For profiling, the code execution Mbed API was used, which uses timer interrupts to track elapsed time. Figure 3.2 shows models ranked from the fastest to the slowest.

We can see that all models perform inference in less than 1 second, which was a

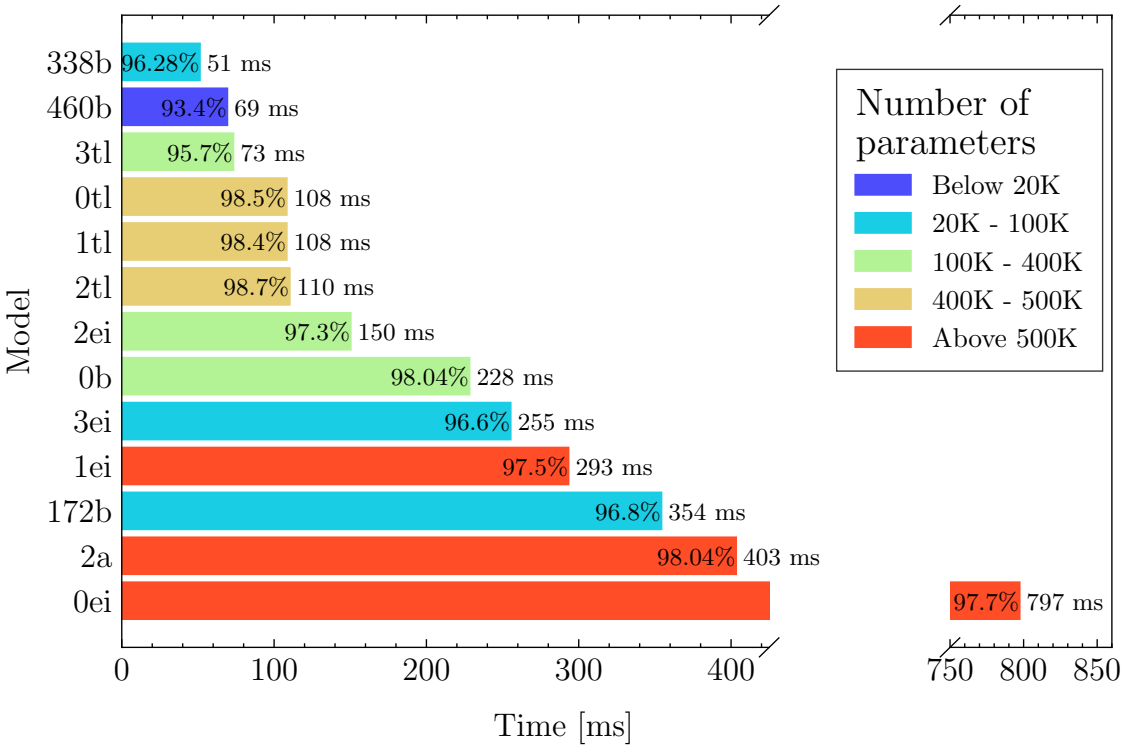

Figure 3.2: Comparison of time of inference of different models.

constraint that we set earlier in Section **??**. Best time wise performing model was *338b* with inference time of 51 ms, there are also many models that perform inference under 300 ms.

We also discovered some unexpected trend in results. We assumed that inference time is proportional to the number of parameters if the general structure of the model remains the same. As can be seen from the Figure 3.2 there are few exceptions to this rule, model *338b* executed inference faster than *460b* although it has more parameters (20K versus 13K). Model *172b* was slower than *0b*, although it has five times less parameters. This behaviour is not exclusive only to our models, but it can be seen in Edge Impulse models as well, for example, models *2ei* and *1ei*.

Edge Impulse models trained with Transfer Learning technique *0tl*, *1tl*, *2tl* and *3tl* should not be compared to other models in this sense, as the architecture of MobileNetV2 contains additional different operations.

We can only speculate about the reason for this behaviour, since it is present both in our models and Edge Impulse models, we can assume this to be a TFLite bug.

3.2.1 Comparison of code sizes

We also wanted to inspect Flash and RAM sizes of binaries, that we compiled for on-device testing. For this task we used `arm-none-eabi-size` command line tool which returns sizes of `text`, `data`, `bss` sections in bytes, example of output can be seen in Figure 3.3. To compute used Flash we need simply add bytes from `text` and `data` sections, and to compute used RAM we add together `data` and `bss` sections[4].

```
$ arm-none-eabi-size firmware.elf
   text    data     bss     dec     hex filename
 149124     388   47064  196576   2ffe0 firmware.elf
```

Figure 3.3: Example output of arm-none-eabi-size command.

Code sizes for all models are presented in Figure 3.4, models are ordered the same way as they have been in Figure 3.2.

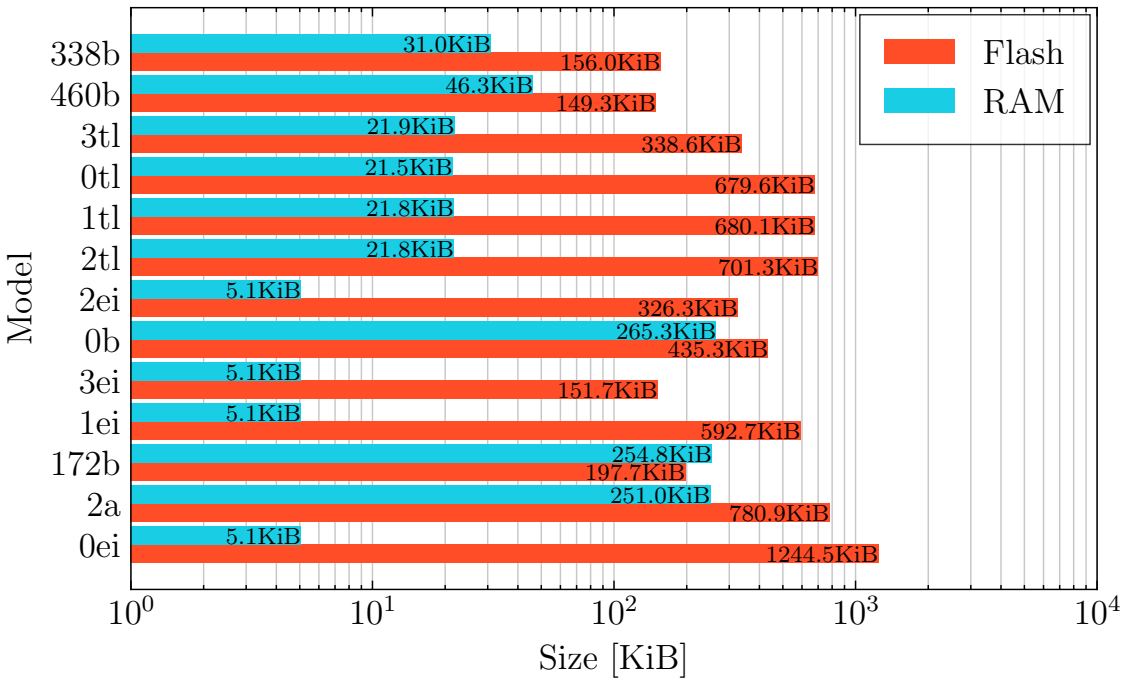

Figure 3.4: Comparison of Flash and RAM size of compiled example models.

We can see that all of our models generally use more RAM then edge Impulse models. This is due to how the inference is executed. TFLite Micro uses a generic interpreter

[4]Data section which contains initialized static variables is first placed into Flash memory and is copied to RAM before program enters main function. That is why we have to account for additional `data` section in Flash memory.

approach, where the model is loaded at runtime. Edge Impulse uses a compiled approach, which they named The EON™ Compiler [6]. The EON™ Compiler still uses TFLite Micro, however, it does not use its interpreter, but directly calls operation kernels. This means that linker knows exactly which operations are used and more data can be moved into Flash, thus eliminating unneeded code space [6].

3.2.2 Comparison of different optimisation options

To be able to run the ML inference at maximum possible efficiency under MicroML some extra amount of work and research was required. Figure 3.5 shows reductions in inference time of *0b* model while using different optimisation methods.

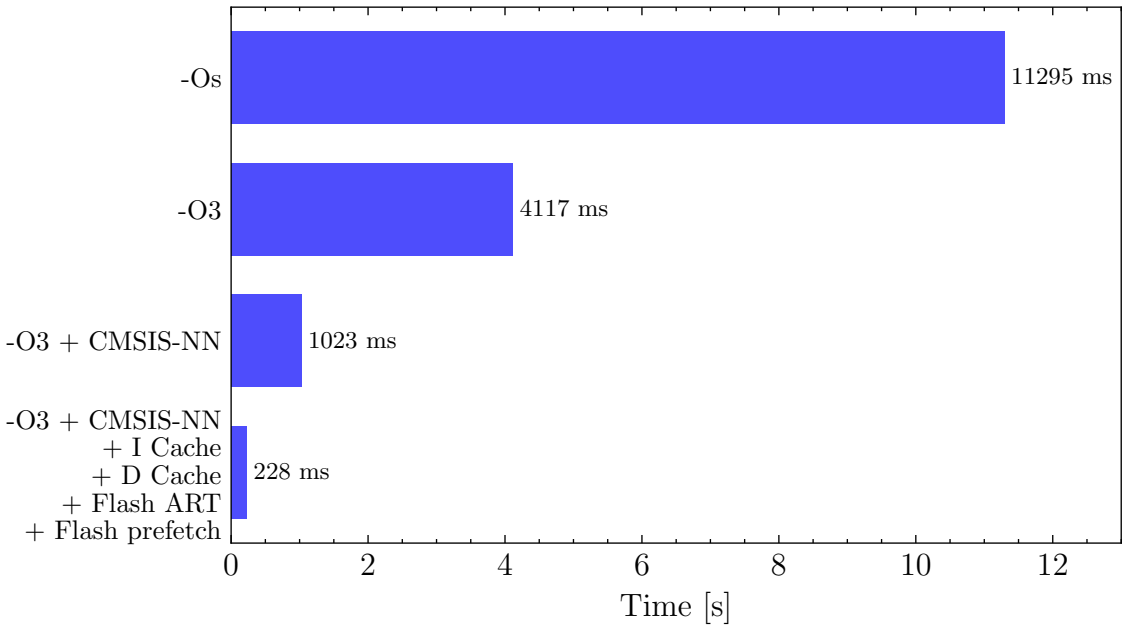

Figure 3.5: Inference time of *0b* model using different optimisations.

We started with no optimisations at all, while using only `-Os` compiler flag. `-Os` flag generally optimises for minimal size, it enables all `-O2` optimisations, except those that increase size. This optimisation level is often used, however, we found out that inference time of more than 11 seconds was too long.

Changing optimisation level to `-O3` drastically decreased the time of inference, down to 4117 ms. `-O3` turns on all `-O2` optimisations plus additional ones and completely

disregards any code size optimisations.

Changing compiler optimisations flags could not lower time of inference any further, so other approaches were needed. While reading through TensorFlow documentation we saw that it supports CMSIS-NN library for ARM microcontrollers. CMSIS-NN is a collection of efficient Neural Network kernels, that intends to maximise performance and lower code size of NN models implemented on ARM microcontrollers. TensorFlow provides wrappers for some of these kernels, such as convolutional, fully connected, pooling layers and others. Not much work was needed to use these highly efficient kernels, we only needed to specify in our Makefile that we want to compile CMSIS-NN kernels and not compile generic TensorFlow kernels. Time of inference dropped for about 3 seconds, down to 1023 ms.

As we saw that similarly sized Edge Impulse models were running much faster on Mbed platform compared to our MicroML code, while using the same microcontroller, we knew that there was another step left. The final performance increase was reached by using features only fully found in Cortex-M7 microcontrollers and partly in Cortex M3/4 microcontrollers. To achieve it we had to enable I and D caches, flash prefetch and flash ART.

ART stands for Adaptive real-time memory accelerator, which encompasses I/D caches and flash prefetch buffer. I and D caches are small, efficient portions of memory, which are located in the CPU of the microcontroller. They hold instructions and data respectively, if those are requested by next microcontroller instruction, they can be read much faster compared to reading them from flash memory. By enabling flash prefetch microcontroller reads additional sequential instructions into prefetch buffer, thus enabling execution without any wait states (if instruction flow is sequential). Elimination of wait states greatly improved the time of inference, it was decreased to 228 ms.

3.2.3 Scoring trained models

Choosing the best model for on-field deployment is a hard task due to many different metrics: precision and recall values, time of inference, and code size. To make this

job easier we devised a scoring system: each metric is going to be normalized and multiplied with some weight value. All products would then be summed up and the result would represent the final score. The Sum of the weights is equal to 100, which means that the possible maximum score is also 100. We decided to allocate 50 weight points to all precision and recall values, 30 points to the time of inference value, 5 points for Flash size and 15 points for RAM size. As we care more about precision and recall values of elephant, human and cow classes we gave them 7 weight points each, while nature/random class received only 4. We value Flash size less than RAM, as most of the microcontrollers have much less RAM than they do Flash, thus we gave 15 weight points to RAM size and only 5 to Flash size.

Since the time of inference, Flash and RAM sizes are properties, which should give a larger score, the smaller they are, we mapped them into a range between 0 and 1. The smallest value inside the set would be assigned 1, the biggest 0, the values in between are mapped linearly.

Scoring is described mathematically in 3.1, while the final results can be seen in Figure 3.6

$$
\begin{aligned}
Score[i] = {} & 7K(P_{elephant}, i) + 7K(P_{human}, i) + 7K(P_{human}, i) + 4K(P_{ntr/rnd}, i) \\
& + 7K(R_{elephant}, i) + 7K(R_{human}, i) + 7K(R_{human}, i) + 4K(R_{ntr/rnd}, i) \\
& + 30\, F(ToI, i) + 5\, F(Flash, i) + 15\, F(RAM, i)
\end{aligned}
$$

$$
K(X, i) = \frac{(X[i] - MIN(X))}{MAX(X) - MIN(X)}
$$

$$
F(X, i) = \frac{(X[i] - MAX(X))}{MIN(X) - MAX(X)} \tag{3.1}
$$

Where:

$Score$ - Vector of calculated scores

i - ith model

P_j - Vector of precision values of jth class

R_j - Vector of recall values of jth class

ToI - Vector of Time of Inference values

$Flash$ - Vector of Flash sizes

RAM - Vector of RAM sizes

$K(X, i)$ - Normalizing function with vector X and element index i as arguments

$F(X, i)$ - Normalizing, inverting, function with vector X and element index i as arguments

$MAX(X)$ - Function that finds maximum element in vector X

$MIN(X)$ - Function that finds minimum element in vector X

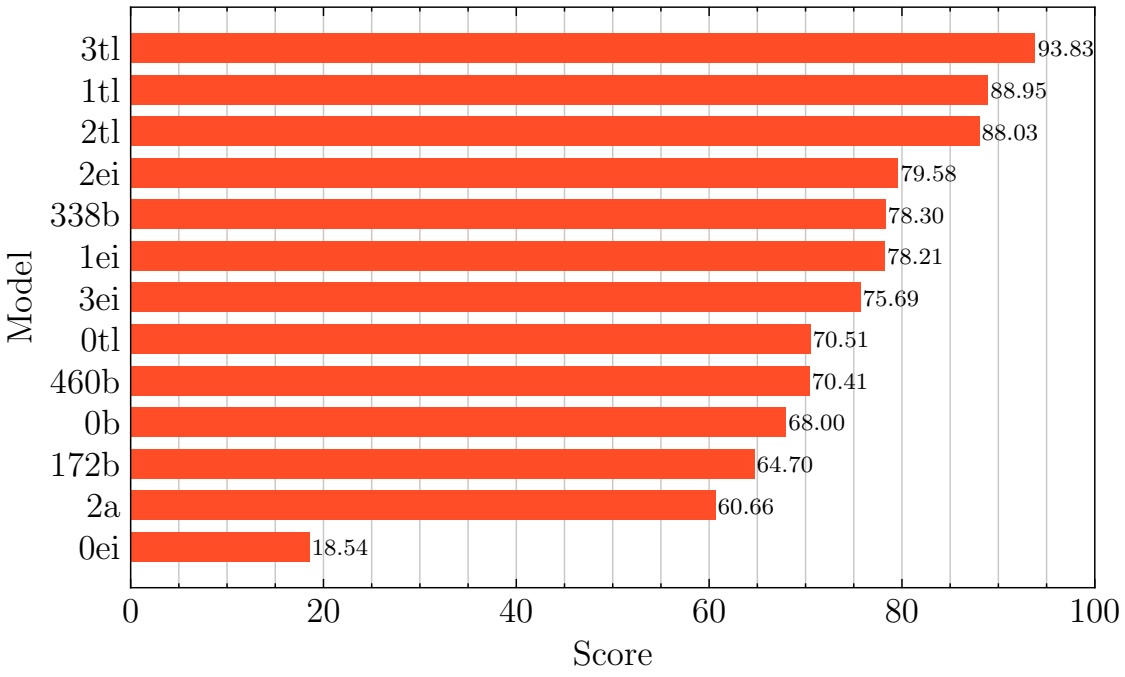

Figure 3.6: Score comparison of different models

Final verdict: model *3tl* received the highest score, models *1tl*, *2tl* follow. This should not be a surprise, models trained with Transfer Learning achieve high accuracies and execute inference in about 100 ms or less while compiled approach for computing Neural Networks keeps used Flash and RAM sizes to a minimum.

Bibliography

[1] Mpaland, A printf / sprintf Implementation for Embedded Systems, GitHub. Available on: `https://github.com/mpaland/printf`, [27.10.2020].

[2] Sagadin M., MicroML, Quick-start machine learning projects on microcontrollers with help of TensorFlow Lite for Microcontrollers and libopencm3, GitHub repository. Available on: `https://github.com/MarkoSagadin/MicroML`, [27.10.2020].

[3] Roeder, L., Netron, Visualizer for neural network, deep learning, and machine learning models. Available on: `https://netron.app/`, [30.10.2020].

[4] Icons8 - Icons used in various figures. Available on: `https://icons8.com/`, [21.9.2020].

[5] Geron, A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems, 2nd edition.* O'Reilly Media, Sebastopol, CA, 2019.

[6] Jongboom J., Introducing EON: neural networks in up to 55less ROM. Available on: `https://www.edgeimpulse.com/blog/introducing-eon`, [20.11.2020].