# Table of Contents


**1 Planning and design of early warning system** **3**

1.1 Hardware . . . . . 4

1.1.1 Nucleo-F767ZI . . . . . 5

1.1.2 Wisent Edge tracker . . . . . 5

1.1.3 Flir Lepton 2.5 camera module and Lepton breakout board . 7

1.1.4 PIR Sensor . . . . . 10

1.2 Firmware . . . . . 10

1.2.1 Tools and development environment . . . . . 10

1.2.1.1 Development environment for STM32f767ZI . . . . . 10

1.2.1.2 Development environment for NRF52840 . . . . . 10

1.2.2 Architecture design . . . . . 11

1.2.3 MicroML and build system . . . . . 12

1.2.4 Running inference on a microcontroller . . . . . 16

1.2.5 Wisent board control firmware . . . . . 17

1.3 Server side components and software . . . . . 17

**2 Measurements and results** **22**

2.1 Model comparisons . . . . . 22

2.1.1 Hyperparameter search space and results analysis . . . . . 22

2.1.2 Comparison of selected, re-trained models . . . . . 28

2.1.3 Comparison with Edge Impulse model . . . . . 30

2.2 On device performance testing . . . . . 30

2.2.1 Comparison of different optimization options . . . . . 31

2.2.2 Comparison of performance of selected models . . . . . 31

2.3 Power profiling of an embedded early warning system . . . . . 31
2.3.1 Battery life estimations . . . . . 31

# 1 Planning and design of early warning system

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

1. Several small embedded devices, deployed in the field. They capture images with thermal camera, process them and send results over wireless network.
2. One gateway, which is receiving results, and relays them to an application server over internet connection.

In this chapter we focus on the structure and design of deployed embedded system,both from hardware and firmware point of perspective. We also describe 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 on the Figure 1.1

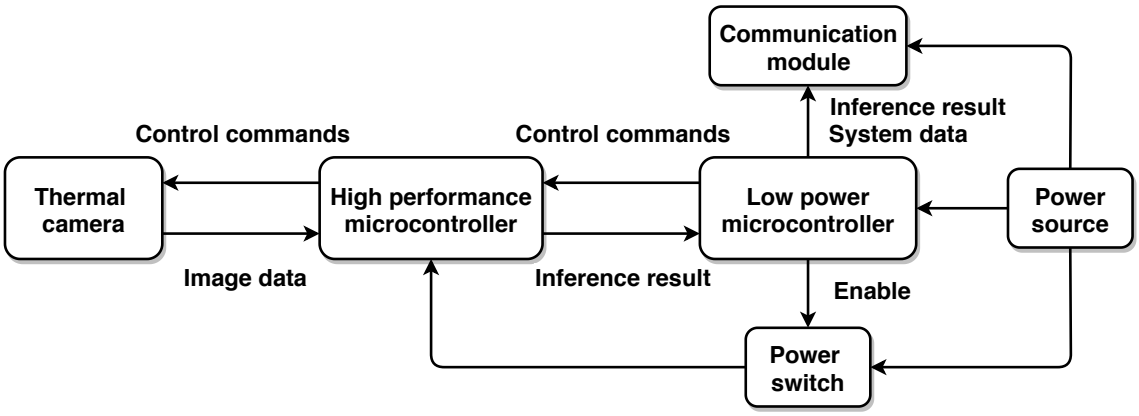



Figure 1.1: General block diagram of an embedded system

Embedded system will consist 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 turned off, to conserve battery life. A less capable, but low power microcontroller will spend most the time in sleep, waiting for a trigger from PIR sensor. PIR sensor will point in the same direction as the thermal camera and will detect any IR radiation of a passing object.

If an object passes PIR's field of vision, it triggers it, which in consequently wakes up a low power microcontroller. Microcontroller will then enable power supply to high performance microcontroller and thermal camera, and send a command request for image capture and processing.

Thermal camera only communicates with high performance microcontroller, which configures it and requests image data. That data is then inputted into neural network algorithm and an probability results are then returned to a low power microcontroller. low power microcontroller then packs the data and sends it over radio through wireless communication module. Power source to high performance microcontroller and thermal camera is then turned of to conserve power. Diagram of described procedure can also be seen on Figure 1.2.

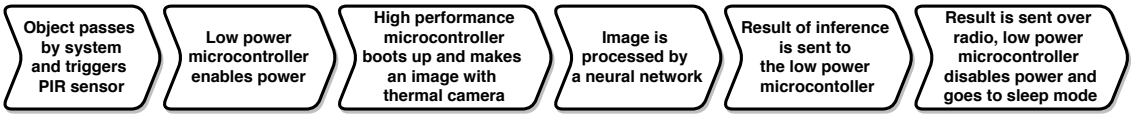



Figure 1.2: Diagram describing behavior of embedded early detection system

## 1.1 Hardware

In this section we present concrete components that we used to implement the embedded part of the early detection system. Hardware version of embedded system diagram is presented on the Figure 1.3. It should be noted that we did not include specific power source into the diagram. Wisent Edge tracker board is general enough

to work with different power sources, such as non-chargeable or chargeable batteries and or solar cells.

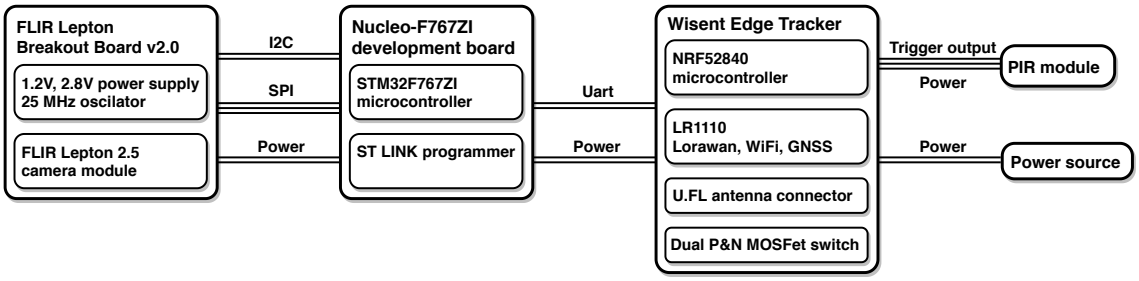



Figure 1.3: Hardware diagram of embedded early detection system

### 1.1.1 Nucleo-F767ZI

Nucleo-F767ZI (seen on Figure 1.4) is a development board made by STMicroelectronics. Board features STM32F767ZI 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 different memory caches and flash accelerator, which provide extra boost in performance. It is convenient to program it, as it includes on board ST-LINK programmer circuit.

We chose this microcontroller simply because it is one of 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.

### 1.1.2 Wisent Edge tracker

For the part of the system which had to contain low power microcontroller, communication module and power control for Nucleo-F767ZI board we decided to use Wisent Edge tracker board. Wisent Edge (seen on Figure TODO ADD IMAGE) is a tracker solution, specifically developed for conserving endangered wildlife animals. It is one of many tracker solutions that were a product of open-source[1]collaboration between Irnas and company Smart Parks, which provides modern solutions in anti-

[1]As a part of OpenCollar project, the design of Wisent Edge is open-source and available on GitHub [1], alongside other hardware and firmware tracker projects.

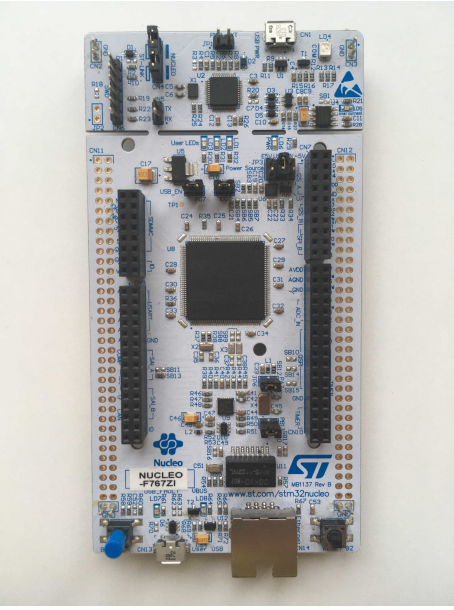



Figure 1.4: Nucleo-F767ZI development board

poaching and animal conservation areas.

The main logic on the board is provided by Nordic Semiconductor's NRF52840 microcontroller with Cortex-M4 core, which has 1 MB of flash, 256 kB of RAM and Bluetooth 5 support. NRF52840 has consumption of 0.5 µA in sleep mode, which makes it ideal for our purpose.

Wisent Edge also features Semtech's LR1110 chip (which acts as a LoRa transceiver, GNSS and WiFi location module) and another GPS module, U-blox's ZOE-M8G[2]. There is a ceramic GPS antenna on board and a U.FL connector to which a dual band Wifi, Bluetooth and LoRa antenna can be attached.

As geopositioning of system was not primary concern, GNSS functionalities were not used, however they might be usefull in future.

Power control of a Nucleo-F767ZI board and FLIR camera is provided by a dual

[2]Reason for two GNSS modules is that although LR1110 chip can provide extremely power efficient location information, it's accuracy is smaller when compared to ZOE-M8G and it can only be resolved after sending it to an application server [2].

channel p and n MOSFET, circuit can be seen on Figure 1.5. Circuit functions as a high side switch, with microcontroller pin driving enable line. When enable line is low, n MOSFET is closed, therefore p MOSFET is also closed, as it is pulled high by resistor R1. When enable line is high, n MOSFET is opened, therefore gate of p MOSFET is grounded, which opens the MOSFET.

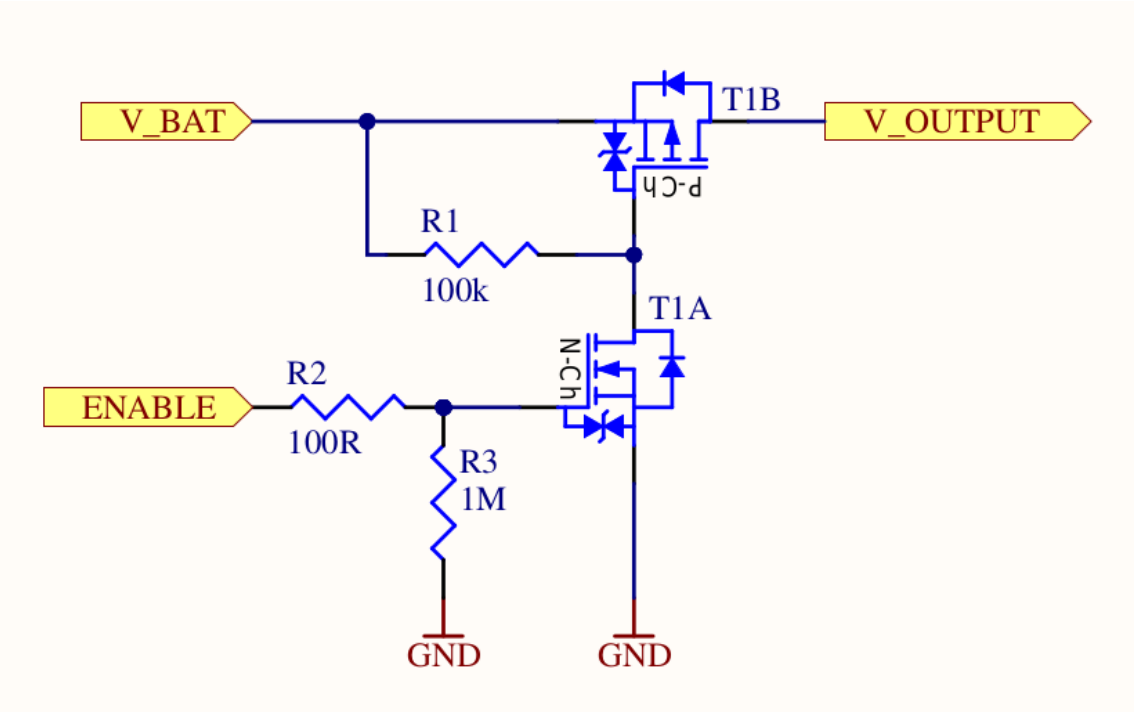



Figure 1.5: Dual P and N MOSFET configuration for power switching

### 1.1.3 Flir Lepton 2.5 camera module and Lepton breakout board

In section **??** it was described what kinds of thermal cameras exist and how do they work, and in section **??** it was 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 actually make images with it.

FLIR Lepton camera needs to be powered from two different sources, 1.2 V and 2.8 V, as well it needs a reference clock of 25 MHz. All of this is provided by Lepton breakout board, which can be seen on the Figure 1.6. Front side of the breakout board contains a FLIR module socket and back side has two voltage regulators and a oscillator. Breakout board can be powered from 3.3 to 5 V and also conveniently

breaks out all communication pins in form of headers.

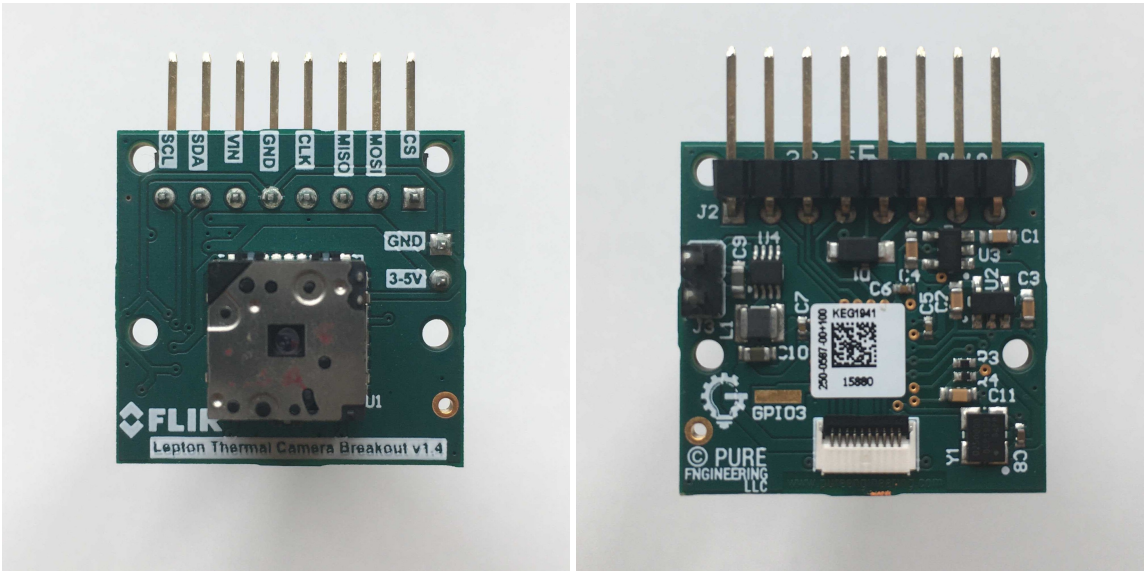



Figure 1.6: Front and back side of Flir Lepton breakout board with thermal camera module inserted.

FLIR Lepton module itself conatins five different subsystems that work together and 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

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

Microcontroller communicates with FLIR camera over two interfaces: two wire interface (TWI) is used for sending commands and controlling the FLIR camera and Lepton's VoSPI protocol is used for image transfer.

TWI is a variation of an I2C protocol, instead of 8 bits, all transfers are 16 bits.

Internal structure of Lepton's control block can be seen on the Figure 1.7. Whenever we are communicating with 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.

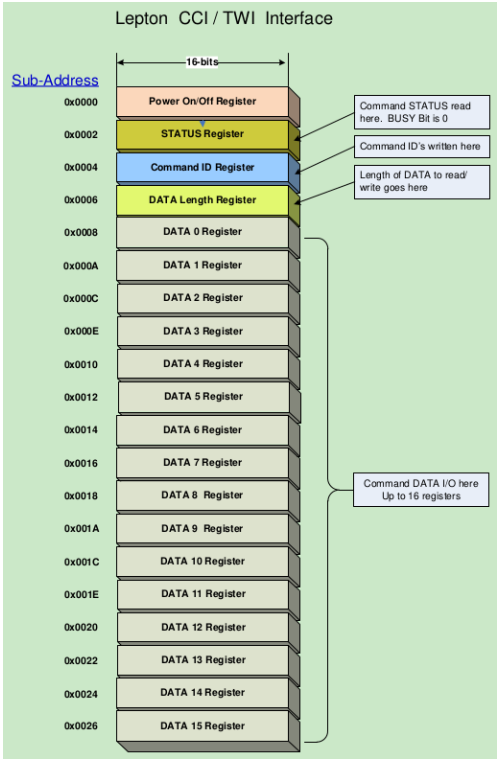



Figure 1.7: Command and control interface of FLIR Lepton camera.

Lepton's VoSPI protocol is used only to stream image data from camera module to the microcontroller, which means that MOSI line is not used. Each image is fits into one VoSPI frame and each frame consists of 60 VoSPI packets. One VoSPI packet contains an 2 bytes of an ID field, 2 bytes of an CRC field and 160 bytes of data[3], that represents one image line.

Refresh rate of VoSPI frames is 27 Hz, however only every third frame is unique from the last one. It is a job of the microcontroller to control the SPI clock speed and process each frame fast enough so that next unique frame is not discarded.

[3]Because images pixel values fit into 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 into 160 bytes. If AGC conversion is turned on, each pixel is then mapped into 8-bit range, however the size of one line in VoSPI packet still remains 160 bytes, 8 most significant bits are simply zeros.

### 1.1.4 PIR Sensor

## 1.2 Firmware

### 1.2.1 Tools and development environment

For our firmware development we did not chose any of various vendor provided integrated development environments. We instead used terminal text editor Vim for writing and editing the code.

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

#### 1.2.1.1 Development environment for STM32f767ZI

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

As a hardware abstraction library we used libopencm3, which is a open-source low level library that supports many of Arm's Cortex-M processors cores, which can be found in 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 excellent library by GitHub user mpaland [3]

#### 1.2.1.2 Development environment for NRF52840

To develop firmware for NRF52840 we decided to use The Zephyr OS, which is a small kernel, designed for IoT embedded systems. Besides usual RTOS functional-

ities such as tasks, mutexes, semaphores it also provides common driver API for supported microprocessors.

### 1.2.2 Architecture design

STM32F767ZI firmware was designed to be very efficient and lean, only truly necessary parts of firmware were implemented.

As seen on Figure 1.8 we split the firmware into two hardware and application modules.

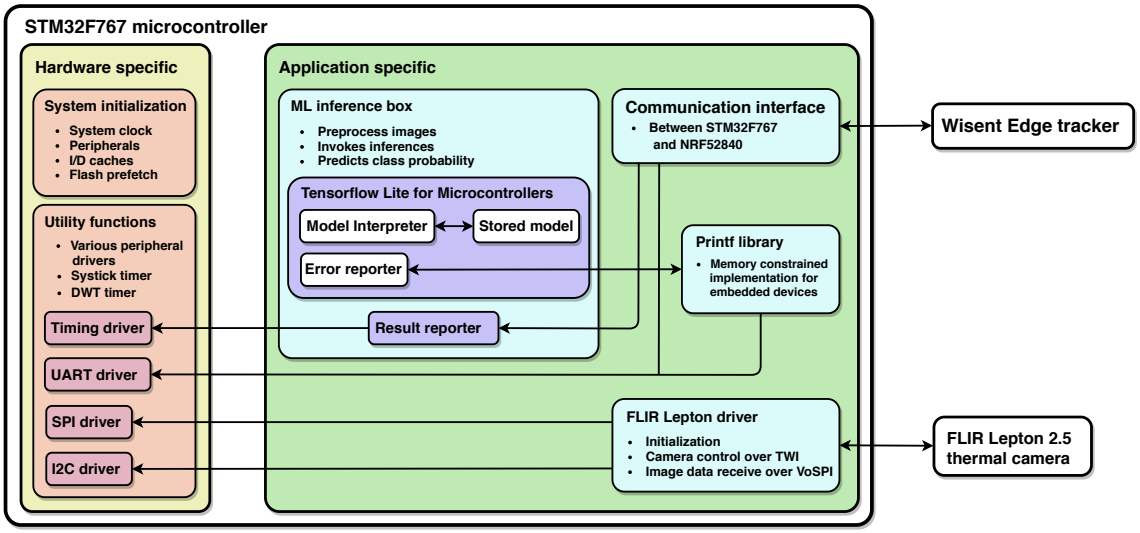



Figure 1.8: Architecture diagram of the firmware that is running on a STM32F767ZI microcontroller.

Both modules are lightly coupled, which enables us reuse of the application module on a different hardware sometime in the future. 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 more abstract.

FLIR Lepton driver was written from scratch, as many libraries provided either by camera manufacturer or open source communities were too complex and implemented many features which we did not require.

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

TODO describe communication interface between THIS AND WISENT EDGE.

The architecture diagram for NRF52840 can be seen on Figure 1.9. For NRF52840 microcontroller, we did not had to write any peripheral drivers, as they were provided by Zephyr itself. Priority was to achieve low power consumption, for which NRF52840 had to spend most of its time in sleep mode, such behavior was easily configurable in Zephyr.

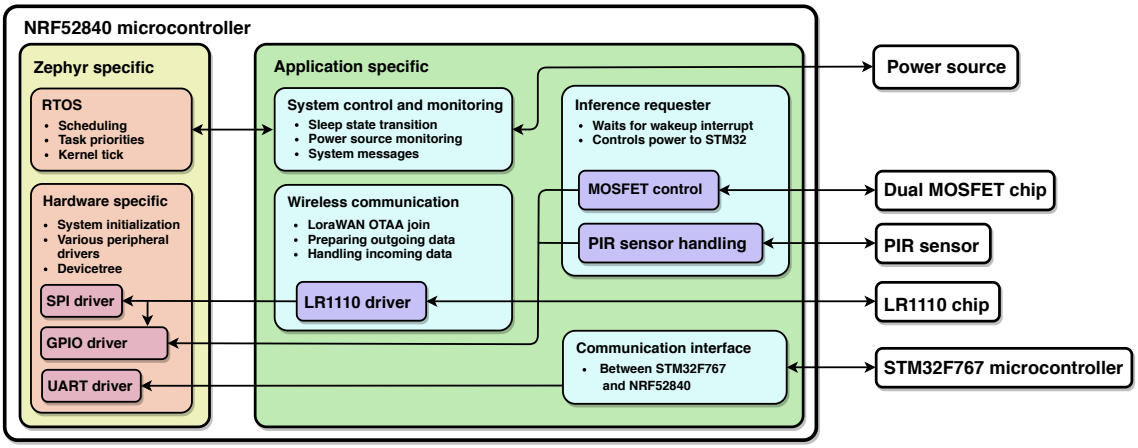



Figure 1.9: Architecture diagram of the firmware that is running on a NRF52840 microcontroller.

This functionality is encapsulated in inference requester module, which is also waking up the microcontroller when PIR trigger signal is received and controlling the MOSFET.

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

TODO describe communication interface between THIS AND WISENT EDGE.

### 1.2.3 MicroML and build system

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 analyze how the code is build in TFLite Micro.

To compile source files and build binaries TFLite Micro uses GNU Make. Main

makefile that includes several platform specific makefiles dictates how firmware is built and several scripts which download various dependencies. By providing command line arguments users decide which example has 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 observing 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 library in linking stage. If we wanted to build firmware for a different example, Make would only had to compile source files of that example and it could reuse previously made library. As compiling of static library took quite some time, this was an efficient option.

After analyzing 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 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 called MicroML[4]. MicroML enables users to develop ML applications on libopencm3 supported microcontrollers. Project's directory structure can be seen on Figure 1.10

[4]Project is open-source and publicly available on GitHub [4].

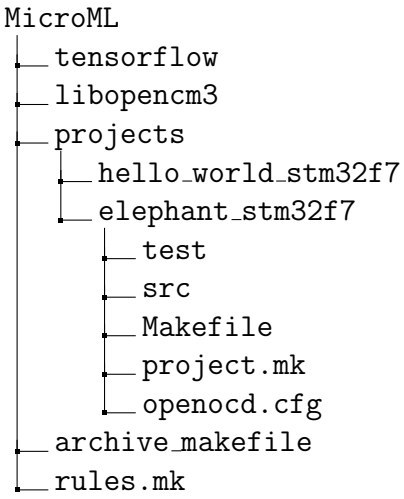



Figure 1.10: Directory structure of MicroML project.

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. It is a place where we also define which microcontroller are we using and what kind of compiler optimization we would like to set.
- **openocd.cfg** - Configuration file which tells OpenOCD which programmer is used to flash which microcontroller and location of the binary file that has to be flashed.
- **Makefile** - Project's makefile which is copied from project to project. It makes it possible to call `make` directly from projects directory, which eases development process. It does not include any building rules, those are specified in included `rules.mk` file in root directory of the project.

Some initial commands need to be executed when the project is cloned from the GitHub for the first time. Figure 1.11 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

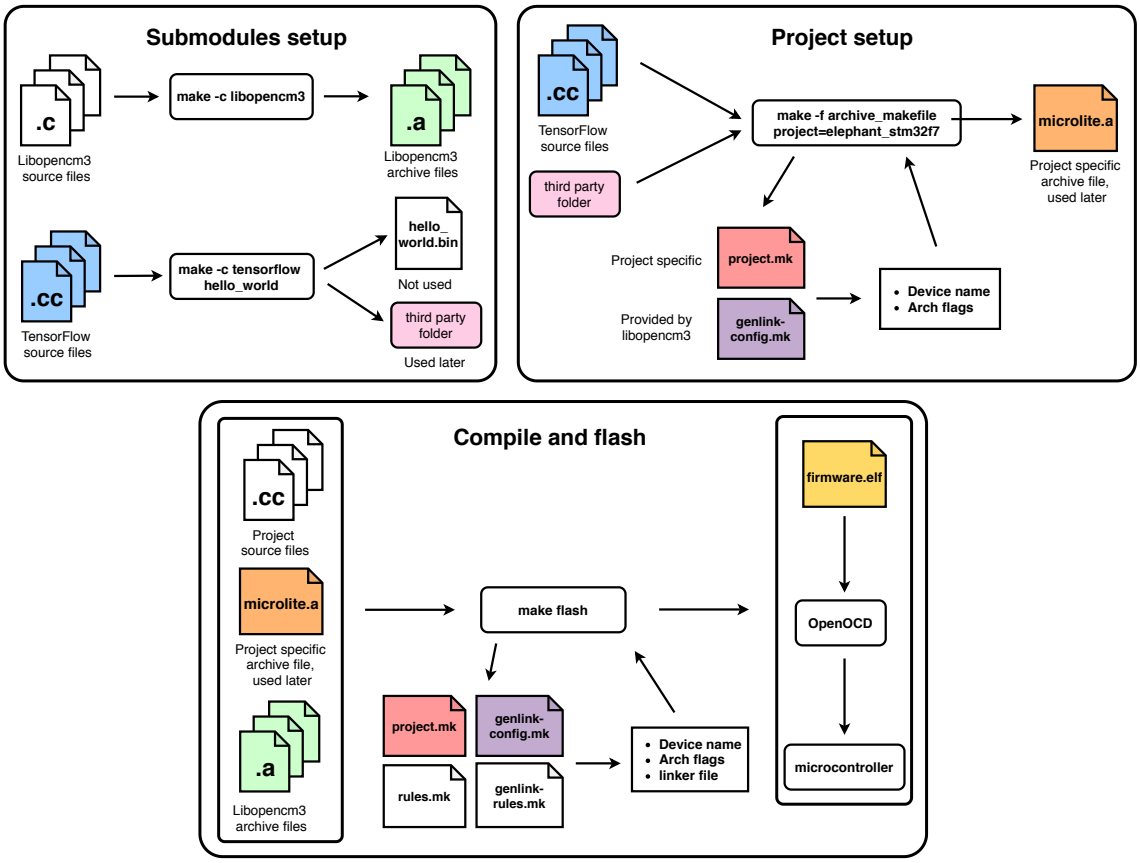



Figure 1.11: Build system of MicroML project.

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 this files, so does MicroML. *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 with the help of `DEVICE` variable determines which compile flags[5]are needed. Afterwards all needed TensorFlow source files and third

[5]For example, flags `-mcpu=cortex-m7`, `-mthumb`, `-mfloat-abi=hard` and `-mfpu=fpv5-sp-d16` tell gcc compiler that we are compiling for cortex-m7 proccesor, that we want to use thumb instruction set and that we want to use hardware floating point unit with single precision. This flags were generated for STM32F767ZI microcontroller by libopencm3.

party files are compiled with this flags and a project specific `microlite.a` archive file is created in our project's folder.

*Compile and flash* stage is then continuously performed during development period. 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 microcontroller specific flags and 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 can take 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 removed any mistakes found on the way.

### 1.2.4 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 1.12 shows a simplified inference code example, copied from our project. As a first step, we need to define size of `tensor_arena` array, which holds memory 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 anymore.

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

In line 13 we pull in our ML model in hex array 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 needed operation specifically (which is done in the example) or simply pull in

all operation implementations. Latter approach is however not recommended, as it results in a large binary size. To find out exactly which operations are needed we used online tool Netron [5].

In lines 27 and 33 we create an `MicroIinterpreter` instance and allocate memory are to it that we specified with `tensor_arena` earlier. Lines 37 and 38 assign input and output of the interpreter to new `TfLiteTensor` variables. This step eases later steps as we can access to input and output more directly. It also enables us to do two things. Firstly, variables input and output now actually point to information about specific format of the data: we can found out how many dimensions are needed, what is size of those dimensions and what is expected type of variable (`uint8_t`, `int8_t`, `float`...). In our test that we ran directly on our laptop we test exactly for these values to confirm that the model will work 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 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. Whole expression is surrounded with timing functions, which are used to keep track of time spend computing inference.

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

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

### 1.2.5 Wisent board control firmware

flow diagram cli interface (this should be put somewhere, or not?)

## 1.3 Server side components and software

In this section we describe possible server side construction of various frameworks which enable us to receive LoRaWAN message, parse it, store it in a database and

present it. For this thesis we did not implement this specific setup as it was not required for testing purposes, however at IRNAS we use this setup all the time for our IoT products and implementing such system would be trivial.

System that we use consists of different components, each one with a distinct task. Tools that we use are The Things Network (TTN), Node-RED, InfluxDB and Grafana. Flow of information and tasks of each tool are presented on Figure 1.13.

TTN is responsible for routing packets that are captured by a gateways to the application server. Since it is open-source and free, anyone can register their gateway device into the network and thus helps 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 decode each message by it.

Node-RED functions as a glue logic that parses packets and shapes them into 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 together. 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 need to use a combination of nodes seen on Figure 1.14 Node *Elephant Gateway* is connected to a specific application on TTN, which is used for collection of packets from our device in field. Any packet that will appear in that TTN application will also appear in Node-RED. Node *Parse packet* extracts information contained in each packet and stores it in a specific format, which is finally send to node *Elephant Database*.

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

Finally data is visualized in Grafana. Grafana is a open source analytics and monitoring solution. Users define which database is set as source and Grafana provides

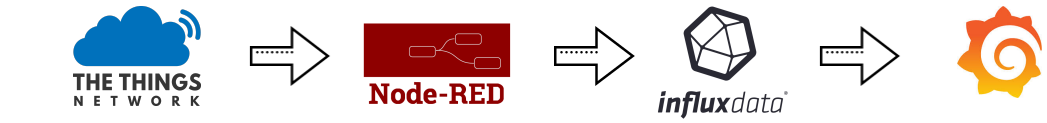
graphical controls which are at some point converted into SQL like language understandable to InfluxDB. Grafana provides different types of visualizations, such as graphs, gauges, heat maps, alert lists and others. In our use case we could display information about various devices in the field, such as battery voltage, number of wakeup triggers, results of each inference and others.

Example of Grafana graph can be seen on Figure 1.15.

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

```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 1.12: Example of TensorFlow Lite inference code in C++.




| The Things Network | Node - RED | InfluxDB | Grafana |
|---|---|---|---|
| - Receives LoRaWAN messages from gateway<br>- Offers direct control of devices<br>- Decodes payload | - Connects to TTN<br>- Parses payloads<br>- Saves them into InfluxDB<br>- Can be used to with many other technologies | - Stores payload in time-series data format. | - Visualizes stored data<br>- Supports many type of graphs and visualization mediums. |

Figure 1.13: Server side flow of information. Icons source: [6]


Figure 1.14: Node-RED flow.

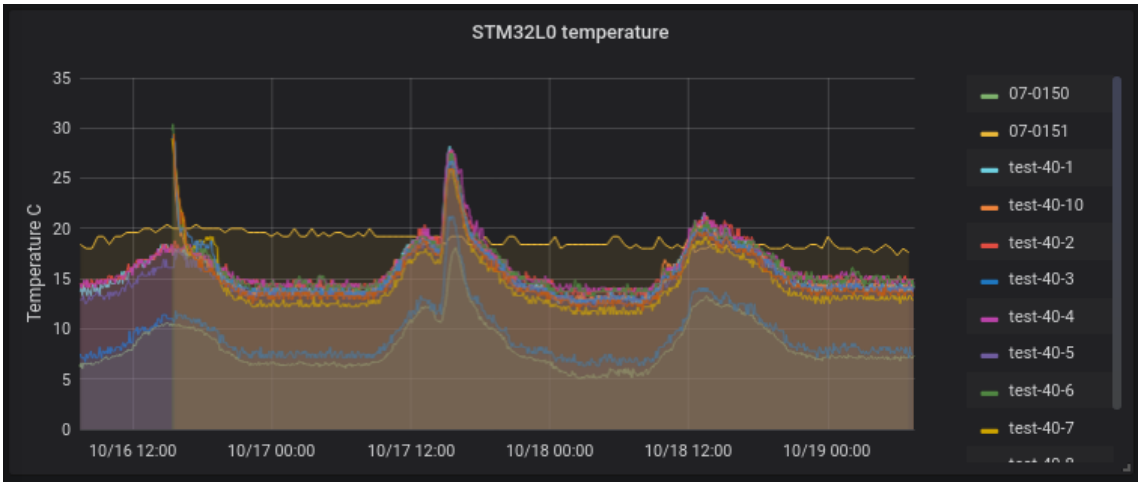



Figure 1.15: Example of Grafana graph.

# 2 Measurements and results

## 2.1 Model comparisons

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 then passed the created model to a `RandomSearch` class, with few other parameters such as batch size, number of epochs and maximum number of trials. We then started hyperparameter search, which means that Keras Tuner was randomly picking a set of hyperparameters and training a model with them. This process was repeated for a trial number of times. Accuracy and hyperparameters that were used while training every module were saved to a log file for later analysis.

After training a number of different models we handpicked a few of them and compared them. Comparison of equivalent model trained in Edge Impulse studio was also done.

### 2.1.1 Hyperparameter search space and results analysis

General structure of CNN model was already described in section **??** and in Figure **??**. We decided to search for following hyperparameters: number of filters in all three convolutional layers (can be different for each layer), filter size in all three convolutional layers (same for all layers), size of dense layer, dropout rate and learning rate. Possible values of hyperparameters (also known as hyperparameter search space) are specified in table 2.1.

Table 2.1: 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 `filter_numX`, `dense_size` and `dropout_rate` hyperparameters was chosen based on initial training tests and various models that were trained on similar data. Value of `filter_size` is usually 3 x 3, however all example ML projects were training on image date of same dimensions. We wanted to test how would a filter with 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 higher values, such as 0.001 or 0.003, would leave model's accuracy stuck at suboptimal optima, from where it could not be improve 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 $20^{\text{th}}$ 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.

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 accuracies, part of it can be seen in Table 2.2.

It is important to keep in mind that we are dealing with imbalanced dataset, where

Table 2.2: Partial results of first random search of hyperparameters

| **Hyperparameter** | FilterNum1 | FilterNum2 | FilterNum3 | DenseSize | DropoutRate | FilterSize | LearningRate | Number of parameters | Accuracy[%] |
|---|---|---|---|---|---|---|---|---|---|
| **Model ID** | | | | | | | | | |
| 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 |
| 5a | 64 | 24 | 72 | 88 | 0.45 | 3x3 | 0.0001 | 1,931,356 | 98.28 |
| 6a | 64 | 56 | 40 | 80 | 0.35 | 3x3 | 0.0001 | 1,013,556 | 98.24 |
| 7a | 16 | 40 | 64 | 88 | 0.35 | 3x3 | 0.0003 | 1,719,108 | 98.24 |
| 8a | 48 | 64 | 32 | 64 | 0.35 | 3x4 | 0.0003 | 676,884 | 98.24 |
| 9a | 72 | 48 | 56 | 80 | 0.3 | 3x4 | 0.0001 | 1,419,172 | 98.24 |
| 91a | 72 | 48 | 56 | 40 | 0.35 | 3x3 | 0.0003 | 728,324 | 98.00 |
| 92a | 48 | 24 | 64 | 64 | 0.35 | 3x4 | 0.0001 | 1,262,092 | 98.00 |
| 93a | 24 | 48 | 32 | 72 | 0.4 | 3x3 | 0.0001 | 716,076 | 98.00 |
| 94a | 32 | 48 | 72 | 32 | 0.25 | 3x3 | 0.0001 | 736,732 | 98.00 |
| 95a | 64 | 24 | 64 | 48 | 0.3 | 3x4 | 0.0001 | 959,628 | 98.00 |
| 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 |
| 195a | 72 | 56 | 24 | 56 | 0.25 | 3x4 | 0.0003 | 469,012 | 97.73 |
| 196a | 72 | 48 | 72 | 40 | 0.3 | 3x3 | 0.0003 | 927,252 | 97.73 |
| 197a | 80 | 16 | 32 | 80 | 0.25 | 3x3 | 0.0001 | 785,380 | 97.73 |
| 198a | 56 | 24 | 16 | 88 | 0.25 | 3x3 | 0.0001 | 438,996 | 97.73 |
| 199a | 56 | 24 | 16 | 88 | 0.25 | 3x3 | 0.0001 | 438,996 | 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 |

82.86 % of our validation data are elephant images. Simply classifying all images as elephant class would yield 82.86 % accuracy, which sounds high, although it is actually not helpful.

After analyzing results we came to several conclusions:

1. We saw that almost all trained models, except of 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 shows that selection of hyperparameters is really a non-heuristic task.

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 number of parameters (model's complexity) and accuracy. Although eight of top ten models have more then 1 million parameters, models with IDs 2 and 8 have almost half of the parameters, but still perform well.

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 with ID *192a* has more than 8 times less parameters than model than model with ID *96a*, although the difference in accuracy (0.27 %) is negligible.

As we realized that we do not need complex models to achieve high accuracy on our training data, we decided to run the random search of hyperparameters again. We decided to lower the maximum and minimum numbers of filters and size of dense layer. We also decreased the step from 8 to 2. 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 2.3 We increased the number of `MAX_TRIALS` from 300 to 500, as we were expecting that more models will end

Table 2.3: 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 |

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 2.5.

Some observations:

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

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

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

4. We can see that the model with ID *296b* has quite low number of parameters,

Table 2.4: Partial results of second random search of hyperparameters

| **Hyperparameter** | FilterNum1 | FilterNum2 | FilterNum3 | DenseSize | DropoutRate | FilterSize | LearningRate | Number of parameters | Accuracy[%] |
|---|---|---|---|---|---|---|---|---|---|
| **Model ID** | | | | | | | | | |
| 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 |
| 3b | 46 | 34 | 28 | 40 | 0.35 | 3x4 | 0.0003 | 367,056 | 98.07 |
| 4b | 26 | 36 | 36 | 34 | 0.15 | 3x4 | 0.0003 | 394,568 | 98.04 |
| 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 |
| 98b | 20 | 10 | 16 | 46 | 0.45 | 3x3 | 0.0003 | 224,500 | 97.59 |
| 99b | 32 | 20 | 32 | 26 | 0.25 | 3x4 | 0.0003 | 265,562 | 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 |
| 198b | 12 | 28 | 40 | 32 | 0.4 | 3x4 | 0.0003 | 401,860 | 97.31 |
| 199b | 36 | 38 | 6 | 40 | 0.4 | 3x3 | 0.0003 | 86,972 | 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 |
| 298b | 18 | 28 | 14 | 32 | 0.25 | 3x4 | 0.0001 | 145,592 | 96.87 |
| 299b | 28 | 40 | 20 | 12 | 0.25 | 3x3 | 0.0001 | 89,684 | 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 |
| 398b | 44 | 22 | 6 | 32 | 0.45 | 3x4 | 0.0001 | 71,564 | 96.11 |
| 399b | 12 | 16 | 10 | 28 | 0.25 | 3x4 | 0.0001 | 88,550 | 96.08 |
| 495b | 32 | 14 | 40 | 8 | 0.5 | 3x4 | 0.0003 | 109,334 | 82.86 |
| 496b | 36 | 38 | 22 | 6 | 0.2 | 3x3 | 0.0003 | 59,890 | 82.86 |
| 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 |

Table 2.5: Partial results of second random search of hyperparameters

| **Hyperparameter** | FilterNum1 | FilterNum2 | FilterNum3 | DenseSize | DropoutRate | FilterSize | LearningRate | Number of parameters | Accuracy[%] |
|---|---|---|---|---|---|---|---|---|---|
| **Model ID** | | | | | | | | | |
| 172b | 42 | 44 | 8 | 14 | 0.1 | 3x4 | 0.0001 | 60,672 | 97.38 |
| 230b | 34 | 6 | 4 | 40 | 0.2 | 3x3 | 0.0003 | 50,606 | 97.18 |
| 265b | 6 | 46 | 6 | 24 | 0.3 | 3x3 | 0.0003 | 48,404 | 97.01 |
| 270b | 36 | 20 | 12 | 10 | 0.15 | 3x3 | 0.0003 | 45,086 | 97.01 |
| 304b | 46 | 6 | 10 | 12 | 0.2 | 3x4 | 0.0003 | 40,710 | 96.87 |
| 338b | 4 | 18 | 6 | 10 | 0.05 | 3x4 | 0.0003 | 20,290 | 96.63 |
| 454b | 10 | 14 | 8 | 6 | 0.05 | 3x4 | 0.0001 | 17,610 | 94.18 |
| 460b | 6 | 28 | 4 | 8 | 0.1 | 3x4 | 0.0003 | 13,114 | 93.60 |

only 65,740.

After making a skim analysis of results, we decided to go through the whole list of trained results and mark down those with extremely low number of parameters, which can be seen in Table **??**.

### 2.1.2 Comparison of selected, re-trained models

Two random searches gave us a large amount 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 need 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 different models: *0a*, *2a*, *0b*, *172b*, *338b* and *460b*. We chose the models on purpose to vary greatly in number of parameters, as we expected that this will have important effect on the inference time.

[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 accuracy of retrained models is going to be similar but not exact when compared to the accuracy returned by random

Table 2.6: Second hyperparameter search space

| **Models** | 0a | 2a | 0b | 172b | 338b | 460b |
|---|---|---|---|---|---|---|
| **Metrics** | | | | | | |
| accuracy[%] | 98.18 | 98.04 | 98.04 | 96.80 | 96.28 | 93.4 |
| number of parameters | 1,515,404 | 656,797 | 304,216 | 60,672 | 20,290 | 13,114 |
| Precision of elephant class | 99.22 | 99.46 | 99.25 | 99.29 | 98.8 | 97.8 |
| Precision of human class | 96.92 | 95.38 | 95.38 | 92.0 | 91.69 | 80.31 |
| Precision of cow class | 90.99 | 93.69 | 90.09 | 84.68 | 75.68 | 69.37 |
| Precision of nature/random class | 77.42 | 64.52 | 79.03 | 46.77 | 59.68 | 40.32 |
| Recall of elephant class | 99.29 | 98.8 | 98.84 | 97.87 | 98.43 | 97.39 |
| Recall of human class | 93.2 | 94.51 | 95.09 | 91.44 | 89.22 | 85.57 |
| Recall of cow class | 94.39 | 92.04 | 96.15 | 89.52 | 84.0 | 81.91 |
| Recall of nature/random class | 87.27 | 97.56 | 84.48 | 93.55 | 67.27 | 28.09 |

As we are dealing with imbalanced dataset, where 82.86 % of our validation data consists of elephant images, accuracy is not the best metric to use when comparing models. Precision and recall metrics[2]can give us better idea how well the models are performing, results can bee seen in Table 2.6.

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 of 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 choose *0b*, without penalising accuracy. Models *172b* and *338b* perform a bit worse when compared to top three models however they have low number of parameters which translates to lower inference time. Last model, *460b*, performs the worst and it should not generally be used.

Another way to compare models performance is to look at confusion matrix graphs. Figure 2.1 shows comparison between confusion matrices of *0a* model on the left and *460b* model on the right. In case of *0a* 19 elephant images were not classified correctly, and 17 images were wrongly classified as elephants. This is not ideal, however

search.

[2]Precision tells us what percentage of data points in a specific predicted class actually fall into that class. Recall tells us what percentage of data points inside a certain class were actually predicted correctly [7].

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.

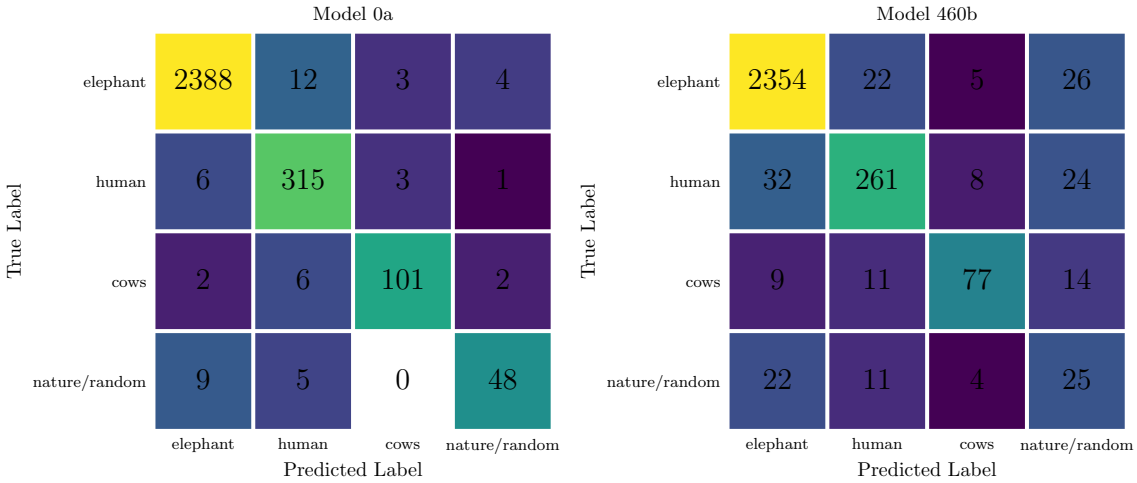



Figure 2.1: Confusion matrices of *0a* model (left) and *460b* model (right)

### 2.1.3 Comparison with Edge Impulse model

Choose one, or few model configurations and make a model in EI.

Describe how did you build Edge Impulse model, maybe mention transfer learning (might be extra work, you did not write about theory)

## 2.2 On device performance testing

Describe setup

To profile execution of our code we first wrote a timer driver based on a Arm's systick timer, however we later decided to use data watch trigger (DWT). DWT does not use interrupts, therefore it does not introduce overhead of calling interrupt routines like systick timer does.

how are you timing,

### 2.2.1 Comparison of different optimization options

### 2.2.2 Comparison of performance of selected models

Transfer learning could be interesting here, bigger number of parameters and takes less time.

## 2.3 Power profiling of an embedded early warning system

- [ ] Power consumption test of whole setup, PIR wakes up wisent, wisent turns on stm32f7 and flir, which makes a picture, does inference, reports result and wisent sends the result. Otii image of consumption with marked sections.(shouldnt be hard)

### 2.3.1 Battery life estimations

Based on numbers and different scenarios estimate how long would this last with different batteries.

# Bibliography


[1] OpenCollar, Collection of open-source conservation solutions, GitHub repositories. Available on: `https://github.com/opencollar-io`, [26.10.2020].

[2] Mustafa L., Sagadin M., LR1110 chip: one solution for LoRa and GNSS tracking. Available on: `https://www.irnas.eu/lr1110-chip-one-solution-for-lora-and-gnss-tracking/`, [26.10.2020].

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

[4] 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].

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

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

[7] 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.