# Table of Contents

# 1 Introduction

## 1.1 Motivation

As a result of increasing human population and habitat loss, human-wildlife conflicts have become increasingly common in recent decades [1]. According to the organisation The World Wide Fund for Nature (WWF), human-wildlife conflicts are defined as: "any interaction between humans and wild animals, that results in negative impacts on human social, economic or cultural life, on the conservation of wildlife populations, or on the environment" [2]. These conflicts range from mostly harmless, non-violent contacts, such as sightings of wildlife animals in urban areas, to the destruction of crops and infrastructure, killing of livestock, and, in the worst cases, loss of human lives. In more severe cases these conflicts end in defensive or retaliatory killings of wild animals, which can drive an already endangered species to extinction.

Polar bears, tigers, and elephants are generally considered to be the most problematic [1]. In the Arctic, as a consequence of the reduction of their natural habitat, polar bears are drawn to human settlements and food dumps while searching for food [3]. Unexpected encounters can become deadly for both sides. As wide-ranging animals, tigers need large areas where they can roam, hunt, and breed [4]. When their natural prey population is depleted, they often turn their attention to poorly protected livestock. Their attacks often have economic, social, and psychological consequences. According to WILDLABS, tigers killed 101 people between the years 2013 and 2016, in India alone [4].

As herbivores, elephants might be seen as less problematic when compared to polar bears or tigers, but this assumption could not be further from the truth.

Although exact numbers vary between sources, casualties from Human-Elephant Conflicts (HEC) are much higher compared to conflicts involving polar bears or tigers. According to WILDLABS, an average of 400 people and 100 elephants are killed every year in India [5]. The leading cause of death of elephants is electrocution (by electric fences, unprotected power lines), followed by train accidents, poaching, and poisoning [6]. Reasons for HEC are similar to the conflicts with polar bears and tigers. Their habitat is shrunk continuously and replaced with crop fields and plantations. As their food options decrease, surrounding agricultural landscapes become inviting. Various HECs can be seen in figure 1.1.

Figure 1.1: Various Human-Elephant Conflicts. Top left: two elephants running from a mob hurling flaming tar balls, top-right: palm platation owner inspecting damage done by elephants to the crops, bottom-left: an elephant crossing the protective barrier, bottom-right: An elephant that died because of HEC. Image sources: [5] [7] [8] [9]

One of the HEC hotspots is in the Sonitpur District, Assam Province, India. In a 5300 $km^2$ large area around 200,000 people and 200 elephants share the same space [5]. Elephants often venture into paddy fields which represent livelihood for local communities. A single elephant can quickly trample fields of rice crops in a few hours, causing big financial problems to already impoverished farmers [5].

Several measures have been taken to minimise HEC: Electrical fences, watch towers, trenches, chilli-based deterrents, regular patrols, usage of trained captive elephants and camera traps with motion sensors.

Although the above mentioned measures function to some degree, they are not effective enough, since they are unreliable, or come into effect when the damage has already been done [10].

## 1.2 Early detection system

One important component of minimizing Human-Elephant Conflicts is a reliable early detection system. A system capable of detecting the presence of nearby elephants would warn nearby communities and give them enough time to prepare and respond nonviolently. The same kind of system would also provide information about common elephant paths, thus giving farmers knowledge on how to construct and place their fences better to minimise HEC. The system (Figure 1.2) should consist of several small, deployed devices with mounted cameras that will detect elephants, and one server which will aggregate alerts and forward them to mobile phones and computers, where the local community will see them.

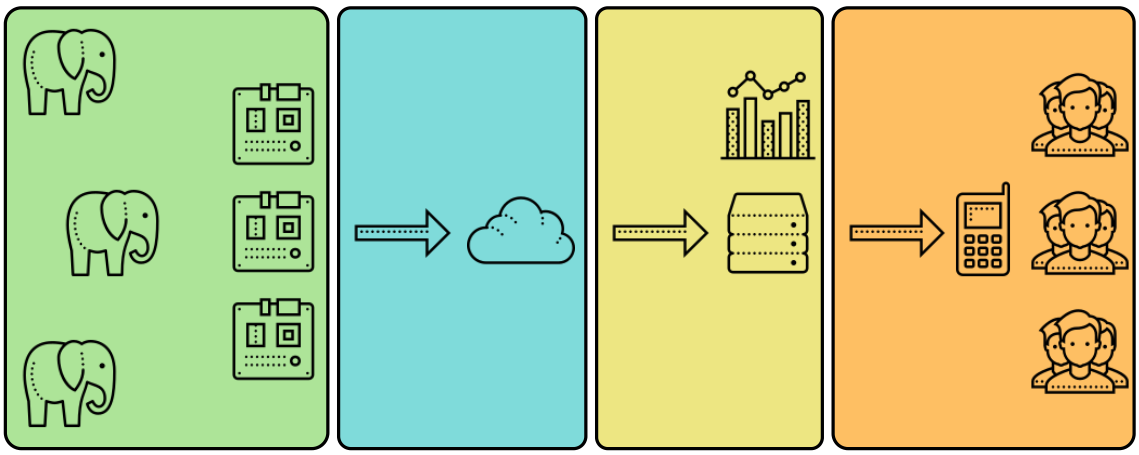


Figure 1.2: Diagram of early detection system. Icons source: [11]

Although most of the villages in the Sonitpur district have access to cell phones and the internet, the connectivity can be unreliable [5]. Furthermore, devices will be placed quite far from the main server, which makes sending a large amount of data a

problem. This limits the choice of wireless networks to long range, low bandwidth technologies. It is, therefore, preferable that elephant detection is done on deployed devices, and only results (which can be few bytes big) are sent over some radio network to the main server. Deployed devices will be placed in forests, fields, with no access to electricity, therefore, they need to be battery powered. Low maintenance of deployed devices is a desirable quality, which means that they should be functional for longer periods without any human interactions. To achieve that with a limited power source, they should be energy efficient, equipped with solar panels, and a low power radio. Devices should spend most of their time in sleep mode, conserving energy, only waking up to take a photo, processing it, and sending results to the main server. A thermal camera is needed, as most of the Human-Elephant Conflicts happen during the night [5].

Elephant recognition can be done with the help of a convolutional neural network (CNN) running inference on a microcontroller. Making this possible and evaluating the solution is the focus of this master's thesis.

## 1.3 IRNAS and the Arribada Initiative

The system described above is currently in development at IRNAS in the collaboration with Arribada Initiative. The Slovenia based Institute IRNAS offers a complete development service, starting with an idea on paper and ending with a finished product. Its previous projects cover a wide range of different fields, from free space optical systems, bio-printing, to Internet of things (IoT) solutions that cover various industrial and nature conservative use cases. The Arribada Initiative is a London based team, that uses open source solutions for purposes of nature conservation. As the winner of WWF and WILDLABS Human Wildlife Conflict Tech Challenge [12], Arribada received funding to develop an early detection system. They spent some time in Assam, India, where they tested a proof-of-concept design [13]. They decided on devices with thermal cameras, as Human-Elephant conflicts often happen during the night. The sensor of choice was an FLIR Lepton 2.5 and/or 3.5. They also created a large dataset of elephant images while filming elephants in Whipsnade Zoo in the

United Kingdom. This dataset will be important for training the Neural Network, and it will be discussed in Section3.4. To create a final embedded system with on device Machine Learning, Arribada chose to work with IRNAS.

## 1.4 Reasoning for the Machine Learning approach

Today Machine Learning (ML) is present in many products that we use on a daily basis. It can be found in email spam detection, recommendation algorithms on Facebook and YouTube, speech recognition on smartphones and medical applications. ML can help us solve problems that are hard to solve by conventional methods. For example, to develop an email spam detector, a programmer would have write a program that would scan the content of an email while checking for the common words and phrases that appear in spam emails and flag the email as spam if they would be found. This would take several iterative cycles of writing the rules, evaluating the solution and analysing the possible mistakes. Even if a possible deterministic solution would be made, it would not stand the test of time, as new forms of spam emails would emerge, tricking the system.

Compare that to a Machine Learning approach. Given enough examples of spam and normal emails, we can train an ML algorithm and let it to discover by itself the rules that mandate what is a spam and what is a normal email. The program would be much smaller compared to the one made by the conventional approach. After the program is launched into the real world, we can use it to store new data and relearn, thus always adapting to new changes. This process can be automated.

Same parallels can be drawn to recognising elephants from thermal images. It is an impossible task to write a deterministic algorithm that could identify an elephant successfully from an image and not confuse it for a human or livestock. Using an ML approach we can train the algorithm on a image dataset and let it figure out the connections between the images and correct labels.

### 1.4.1 Implementation of Machine Learning algorithms

Since ML algorithms are at their core normal math operations, they can be implemented (although maybe not efficiently) in any programming language and on any hardware platform from scratch. However, to avoid reinventing the wheel and wasting the time on algorithm optimisation, it is more logical to use one of ML frameworks. Frameworks such as scikit-learn, Keras, Caffe and TensorFlow enable programmers to write application code in a high level language such as Python, which, at run time, translates to efficient C/C++ code. These frameworks take away low-level details of ML algorithms, enabling programmers to deal only with application code and not its underlying implementation.

In several past years there has been a growing desire to expand ML applications to embedded devices. Running ML algorithms directly on microcontrollers has benefits and challenges compared to running it on computers and servers. These comparisons are described further in Section 2.1.2. There are several frameworks, proprietary and open-sourced, that can be used to develop ML applications on microcontrollers. STMicroelectronics created X-CUBE-AI, a tool that converts the pre-trained model created by one of the various Deep Learning frameworks into an optimised library. X-CUBE-AI works only with STM32 microcontrollers, and is proprietary. Another framework, TensorFlow Lite for microcontrollers, was created as an extension of TensorFlow Lite, which was used to develop ML models for mobile applications. It provides converter tools and C++ implementations of common ML operations. In 2019 another framework, µTensor, was merged with TensorFlow Lite, providing it with support for an efficient CMSIS-NN library developed by ARM.

For this thesis we used TensorFlow Lite for microcontrollers. It can be used with any family of microcontrollers, and is open-source, so we can study its internal code.

### 1.4.2 Edge Impulse

Regardless of the many Ml frameworks on the market, companies that specialise in ML on embedded devices are scarce. One of them is Edge Impulse, which is a

recently founded company in San Jose, USA. They provide users with an end to end web solution for developing ML applications for embedded devices. Instead of designing and writing specific programs that deal with preprocessing of training data and creation and training of ML models, Edge Impulse does this automatically for their customers. After the ML model is created and converted into an optimised format for embedded systems, customers get the immediate first approximation of how much RAM and FLASH memory the model will take and how fast it will run. We can then run the model on the local machine, or deploy to a variety of different platforms, such as Arduino, STM32, OpenMV, Zeyphr and others.

Their solution will be used as a benchmark for our work with TensorFlow Lite.

## 1.5 Objective

The objective of this Master's thesis is to evaluate the feasibility of animals, especially elephants, from thermal images, with Machine Learning algorithms, running directly on a microcontroller.

The objective of this Master's thesis is to design and build a system capable of detecting an elephant with the help of Machine Learning algorithms in the day or at night. Detection of an elephant needs to be reported over a wireless network to an application server.

For that we will:

- Train a Neural Network model capable of classifying elephants, humans and other animals from thermal images.
- Optimise Neural Network model for on device inference.
- Implement on device inference on an STM32 microcontroller using TensorFlow Lite.
- Compare the performance of our implementation against the Edge Impulse implementation.
- Build a system around the STM32 microcontroller with a thermal camera and

wireless network support.

- Profile power consumption of the built system.
- Establish system requirements for different ML applications.

## 1.6 Master's thesis outline

This chapter provided an overview of motivation and the companies involved, some reasoning for choosing the Machine Learning approach and the objectives of this thesis. Chapter 2 provides a theoretical description of the system building blocks. Machine Learning, Neural Networks, thermal cameras, TensorFlow Lite, and others topics are discussed there. Chapter 3 revolves around the design of an image classification Neural Network model. Chapter 4 describes the design and implementation of an early warning system, from the hardware, firmware and software points of view. In Chapter 5 we describe the measurement procedure and results. Chapter 6 presents our findings, describes the limitations of our project, and suggest paths for further research.

# 2 Theoretical description of system building blocks

## 2.1 Machine Learning

According to Arthur Samuel (qtd. in Geron [14]) Machine Learning is a field of study that gives computers the ability to learn without being programmed explicitly. This ability to learn is the property of various Machine Learning algorithms. We will be using the terms "Machine Learning" and "learning" interchangeably. To learn, these learning algorithms need to be trained on a collection of examples of some phenomenon [15]. These collections are called **datasets**, and can be generated artificially or collected in nature.

To understand how the ML approach can solve problems better, we can examine an example application. Let us say that we would like to build a system that can predict a type of animal movement based on accelerometer data. To train its learning algorithm, also known as a **model**, we need to train it on a dataset that contains accelerometer measurements of different types of movement, such as walking, running, jumping and standing still. Input to the system could be either raw measurements from all three axes, or components extracted from raw measurements such as RMS, spectral power, peak frequency and/or peak amplitude. These inputs are also known as **features**, they are values that describe the phenomenon being observed [15]. The output of the system would be a predicted type of movement. Although we would mark each example of measurement data with what type of movement it represents, we would not define the relationship between the two directly. Instead, we would let the model figure out the connection by itself, through the process of training. The trained model should be general enough so that it can predict the type of movement on unseen accelerometer data correctly.

There exists a large variety of different learning algorithms. We can categorise them broadly in several ways, and one of them depends on how much supervision the learning algorithm needs in the training process. Algorithms like K-nearest neighbours, linear and logistic regression, Support Vector Machines fall into the category of supervised learning algorithms. Training data that is fed into them includes solutions, also known as **labels** [14]. The above described example is an example of a supervised learning problem.

Algorithms like k-Means, Expectation Maximization, Principal Component Analysis, fall into the category of unsupervised learning algorithms. Here, training data is unlabelled, algorithms are trying to find similarities in data by themselves [14]. Other categories exist, such as semi-supervised learning which is a combination of the previous two and reinforcement learning, where the model acts inside the environment according to learned policies [14].

Neural Networks, algorithms inspired by neurons in human brains [14] [16], can fall into either of the categories. They are appropriate for solving complex problems like image classification, speech recognition, and autonomous driving, but they require a large amount of data and computing power for training. They fall into the field of Deep Learning, which is a sub-field of Machine Learning.

Training of ML models is computationally demanding, and is usually done on powerful servers or computers with dedicated Graphic Processing Units to speed up training time. After a model has been trained, data can be fed in and prediction is computed. This process is also known as **inference**. The inference is computationally less intensive compared to the training process, so, with properly optimised models, we can run inference on personal computers, smartphones, tablets, and even directly in internet browsers.

### 2.1.1 General Machine Learning workflow

There are several steps in ML workflow that need to happen to get from an idea to a working ML based system, and this is represented in Figure 2.1.

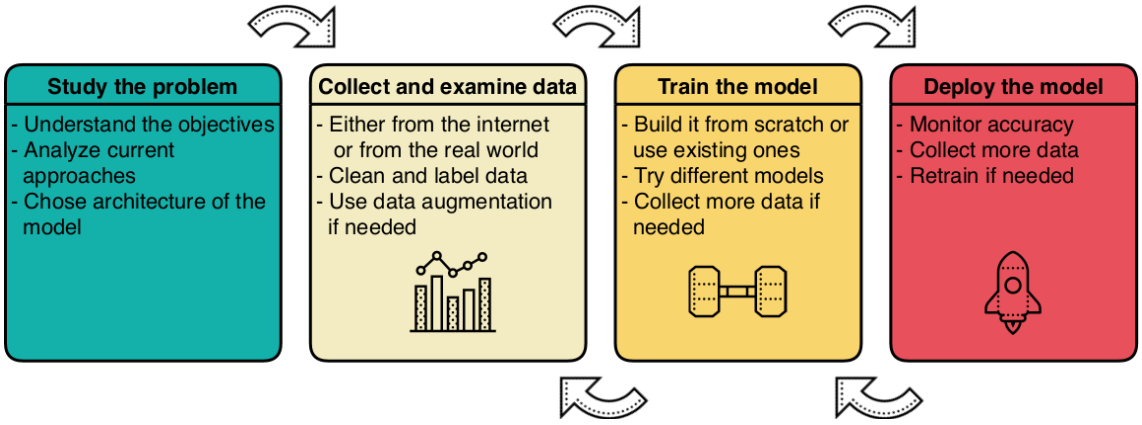



Figure 2.1: Workflow diagram of solving a generic Machine Learning problem. Icons source: [11]

First, the problem has to be studied, it has to be understood what are the objectives, what are current solutions and which approach should be used. Here, we decide on the rough type of ML model that we will use, based on the problem. In the second step we collect and clean up data. We should always strive to collect a large amount of quality and diverse data that represent a real world phenomenon. Collecting that kind of data can be hard and expensive, but we can use various tools, such as data augmentation or data synthesis, thus increasing data size and variety. Sometimes data are not collected by us, in that case we should examine them and extract information that we need. Third, we train the ML model. We might create something from scratch or use an existing model. We can train several different types of models and choose the one that performs the best. To achieve the desired accuracy steps two and three can be repeated many times. In step four we deploy our model and monitor its accuracy. If accuracy drops we can always collect new data and retrain the model.

### 2.1.2 Machine Learning on embedded devices

Machine Learning on embedded devices is an emerging field, which coincides nicely with the Internet of Things. Resources about it are limited, especially when compared to the vast number of resources connected with Machine Learning on computers or servers. Most of the information about it can be found in the form of scientific papers, blog posts and Machine Learning framework documentation [17] [18] [19].

Running learning algorithms directly on embedded devices comes with many benefits. **Reduced power consumption** is one of them. In most IoT applications devices send raw sensor data over a wireless network to the server, which processes it either for visualisation or for making informed decisions about the system as a whole. Wireless communication is one of the more power hungry operations that embedded devices can do, while computation is one of more energy efficient [19]. For example, a Bluetooth communication might use up to 100 milliwatts, while a MobileNetV2 image classification network running 1 inference per second would use up to 110 microwatts [19]. As deployed devices are usually battery powered, it is important to keep any wireless communication to a minimum, so minimising the amount of data that we send is paramount. Instead of sending everything we capture, is much more efficient to process raw data on the devices and only send the results.

Another benefit of using ML on embedded devices is **decreased latency time**. If the devices can extract high-level information from raw data, they can act on it immediately, instead of sending it to the cloud and waiting for a response. Getting a result now takes milliseconds, instead of seconds.

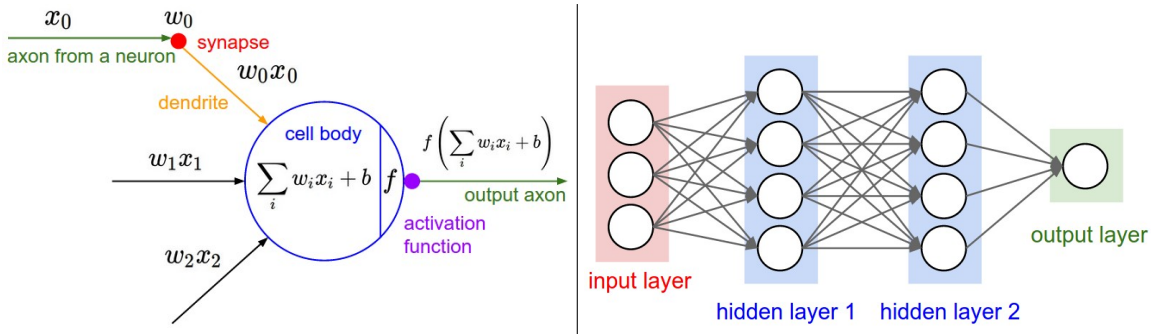
Such benefits do come with some drawbacks. Embedded devices are a more resource constrained environment when compared to personal computers or servers. Because of limited processing power, it is not feasible to train ML models directly on microcontrollers. Also it is not feasible to do online learning with microcontrollers, meaning that they would learn while being deployed. Models also need to be small enough to fit on a device. Most general purpose microcontrollers only offer several hundred kilobytes of flash, up to 2 megabytes. For comparison, the MobileNet v1 image classification model, optimised for mobile phones, is 16.9 MB in size [20]. To make it fit on a microcontroller and still have space for our application, it would have to be simplified.

The usual workflow while developing Machine Learning models for microcontrollers, is to train a model on training data on a computer. When we are satisfied with the accuracy of the model we quantize it, and convert it into a format understandable by our microcontroller. This is described further in Section 2.3.1.

## 2.2 Neural networks

Although the first models of Neural Networks (NN) were presented in 1943 (by McCulloch and Pitts) [14] and hailed as the starting markers of the Artificial Intelligence era, it had to pass several decades of research and technological progress before they could be applied to practical, everyday problems. Early models of NNs, such as the one proposed by McCulloch and Pitts, were inspired by how real biological neural systems work. They proved that a very simple model of an artificial neuron, with one or more binary inputs and one binary output, is capable of computing any logical proposition when used as a part of a larger network [14].

To learn how NNs work we can refer to Figure 2.2a, which shows a generic version of an artificial neuron.




(a) Artificial neuron (b) 3-layer neural network

Figure 2.2: (a) Mathematical model of an artificial neuron, similarities with biological neurons can be seen. (b) Fully connected 3-layer neural network. Image source: [16]

A Neuron takes several inputs, multiplies each input with its **weight** and sums them up. It adds to the sum the **bias** term, and then applies an activation function.

NNs consist of many neurons, which are organised into **layers**. Neurons inside the same layer do not share any connections, but they connect to the layers before and after them. The first layer is known as the **input** layer and last one is known as the **output** layer. Any layers between are said to be **hidden**. In Figure 2.2b we can see a neural network with an input layer with three inputs, two hidden layers with four neurons each, and an output layer with just one neuron. If all inputs of neurons in

one layer are connected to all outputs from the previous layer, we say that a layer is **fully connected** or **dense**, Figure 2.2b is an example of one. NNs with many hidden layers fall into the category of Deep Neural Networks (DNN).

### 2.2.1 Activation functions

Activation functions introduce non-linearity to a chain of otherwise linear transformations, which enables ANNs to approximate any continuous function [14]. There are many different kinds of activation functions, as seen on Figure 2.3, such as sigmoid function and rectified linear activation function (ReLu). A sigmoid function was used commonly in the past, as it was seen as a good model for a firing rate of a biological neuron: 0 when not firing at all, and 1 when fully saturated and firing at maximal frequency [16]. It takes a real number and squeezes it into a range between 0 and 1. It was later shown that training NNs with sigmoid activation function often hinders the training process, as saturated outputs cut off parts of networks, thus preventing the training algorithm from reaching all neurons and configuring the weights correctly [16]. It has since fallen out of practice, and is nowadays replaced by ReLu or some other activation function.

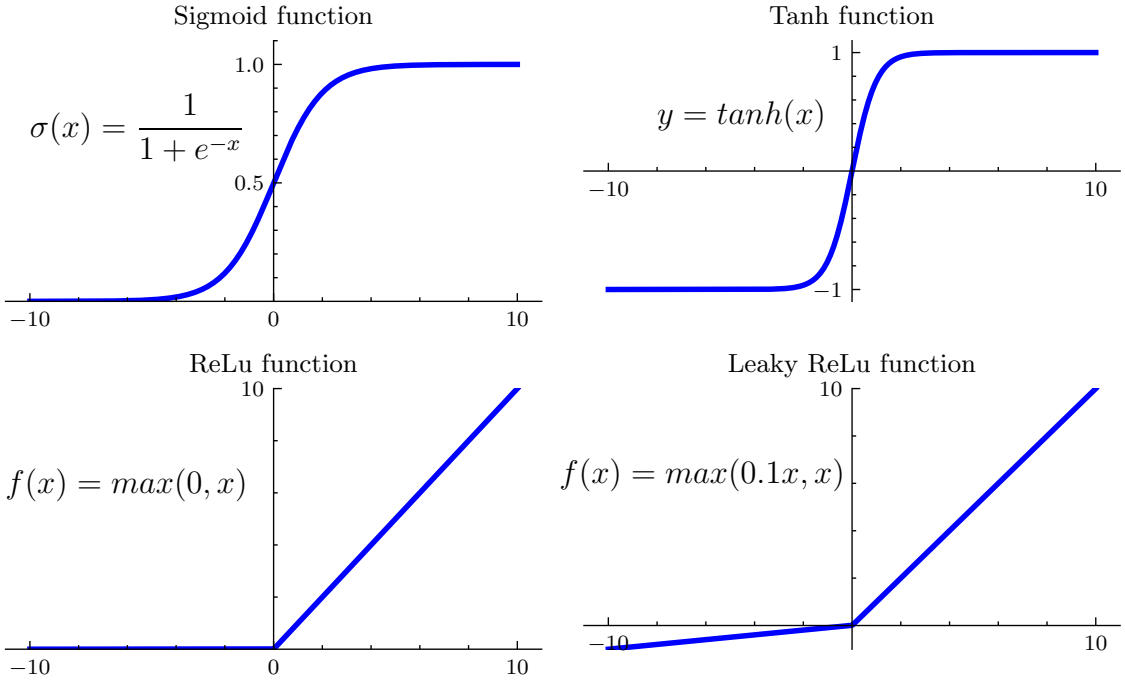



Figure 2.3: Different activation functions and their equations.

Another commonly used activation function is a softmax function (seen in 2.1, which is takes a vector as an input, computes an exponential of every element inside it and divides that with the sum of exponents of all elements [14] The end results is that softmax function transforms the vector of values into a vector of probabilities. Softmax is usually used as an activation in the last layer of a classifier network.

$$\sigma(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}} \quad \text{for } i = 1, \cdots, K \text{ and } y = (y_1, \cdots, y_k) \in \mathbb{R}^K \tag{2.1}$$

Where:

$y$ - Input vector

$K$ - Number of elements in the input vector

$\sigma(y_i)$ - Computed probability of the i-th element in the input vector

### 2.2.2 Backpropagation

Training of Neural Networks is done with a training algorithm, known as **backpropagation**. As mentioned before, we train the Neural Network by showing it a large amount of training data with labels. At the start of the training phase, all weights and biases are set to randomly small values. During each training step, a Neural Network is shown a small batch of training data. Each instance is fed into the NN and the final output label is calculated. This is known as **forward pass**, which is the same as making predictions, except that intermediate results are stored from each neuron from every layer. Calculated output is compared to an expected one using a **loss** (also known as **cost**) function. The loss function returns a single value, which tells us how badly our is NN performing: the higher it is, the worse is our NN performing. The goal is to minimise the loss function, thus increasing the accuracy of our NN. In the context of multivariable calculus, this means that we have to calculate the negative gradient of weights and biases, which will tell us in which direction we have to change each weight and bias so that the value of loss function decreases.

Doing this for all weights and biases at the same time would be complicated, so the backpropagation algorithm does this in steps. After computing the loss function, the

algorithm calculates analytically how much each output connection contributed to the loss function (essentially the local gradient) with the help of previously stored intermediate values. This step is done recursively for each layer until the first input layer is reached. At that moment the algorithm knows in which direction each weight and bias should change, so that the value of the loss function lowers. A procedure is then performed, known as a **Gradient Descent**. All local gradients are multiplied with a small number known as a **learning rate**, and then subtracted from all weights and biases. This way, in each step we change weights and biases slowly in the right direction, while minimising the loss function. Gradient Descent is not only used when training neural networks but also when training other ML algorithms.

We do not have to execute a backpropagation algorithm for each training instance, instead, we can calculate predictions for a small set of training data, calculate the average loss function and then apply backpropagation.

### 2.2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a kind of Neural Networks that work especially well with image data. Like NNs they have found inspiration in nature, in their case the visual cortex of the brain.[1]

In Figure 2.4 we can see an example of CNN which takes an image of a car as an input and outputs probability results in five different classes.

Specific to CNNs are two different types of layers, **convolutional** layers and **pooling** layers. Each convolutional layer detects some sort of shapes: the first ones detect different kinds of edges, while later ones detect more complex shapes and objects, like wheels, legs, eyes, ears. Pooling layers downsample the data in the spatial dimension, thus decreasing the number of parameters and operations needed in CNN. After a few alternating pairs of convolutional and pooling layers, the output of the last pooling layer is flattened out into one dimensional vector and fed into a fully connected NN

[1]Scientists Weisel and Hubel showed that different cells in the primary visual cortex of a cat responded differently to different visual stimuli [16]. Some were activated when shown a horizontal line in a specific location, some were activated by vertical lines. More complex cells responded

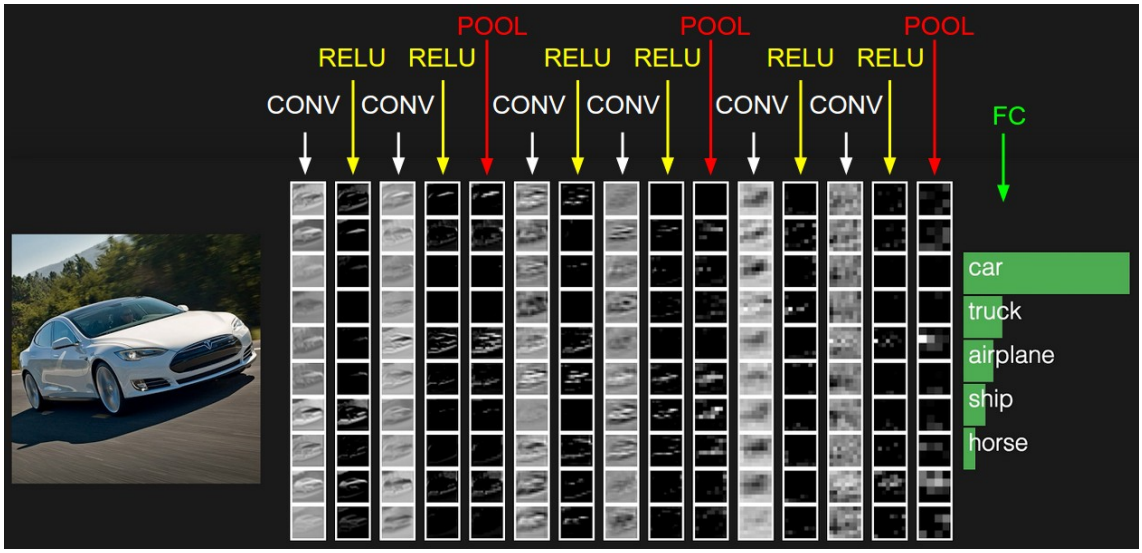



Figure 2.4: Structure of a Convolutional Neural Network. Image source: [16]

which produces probability results in given classes.

It makes sense to explain how convolutional and pooling layers work in greater detail, as this will be important later when we will be designing our CNN models in Section 3.6.

#### 2.2.3.1 Convolutional layers

Data that CNNs operate on are 3 dimensional matrices, where width and height correspond to an image resolution, and depth corresponds to the number of colour channels, 3 for colourful images (red, green, blue) and 1 for greyscale. When speaking about these matrices we will refer to them as volumes.

Convolution layers perform dot products between input volume and several **filters** or **kernels** to produce output volume. In these layers, filters are configured through the training phase. We can see a concrete example in Figure 2.5. 2D filter with size 2 x 2 covers a part of the input volume, over which element-wise multiplication is computed, elements are summed and the result is written into the first element of output volume.

The filter then moves a fixed distance or **stride** and the process is repeated. It is

to boxes, circles and so on. CNNs also detect simpler shapes first and use them to detect more complex ones later.

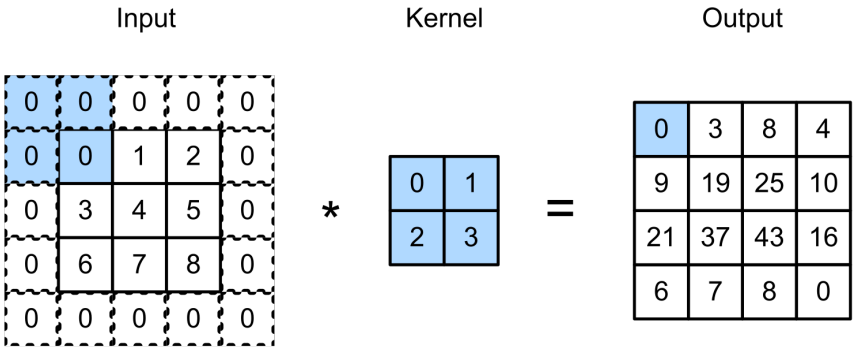



Figure 2.5: Dot product operation between filter and zero-padded input matrix. Image sources: [21]

important to note that, although we can choose the width and height of the filter, the depth of the filter is always equal to the depth of the input volume. If the depth is larger than one, then dot products are done for each 2D matrix in the depth dimension separately, and then an element-wise sum operation is performed between these matrices. To avoid losing information from the image pixels that are on the edges (as they would be included in dot products fewer times compared to central ones) we often pad input images with zeros.

The size of output volume depends on several factors as seen in 2.2.

$$V_o = (V_i - F + 2P)/S + 1 \tag{2.2}$$

Where:

$V_i$ - Input volume size, only width or height

$V_o$ - Output volume size, only width or height

$F$ - Filter or receptive field size

$P$ - Amount of zero padding used on the border

$S$ - Stride length

If we examine the example in Figure 2.5 we can see that input with a size 3 x 3, stride 1, padding 1 and filter with a size 2 x 2, produces an output with size 4 x 4.

The depth of output volume is equal to the number of filters used in the convolutional

layer as seen in Figure 2.6, it is a norm that a single convolutional layer uses a large number of filters to produce a deep output volume [16]. It is also common to set padding, stride and filter size so that the width and height of the input volume are preserved. This prevents the information at the edges from being lost too quickly [16].

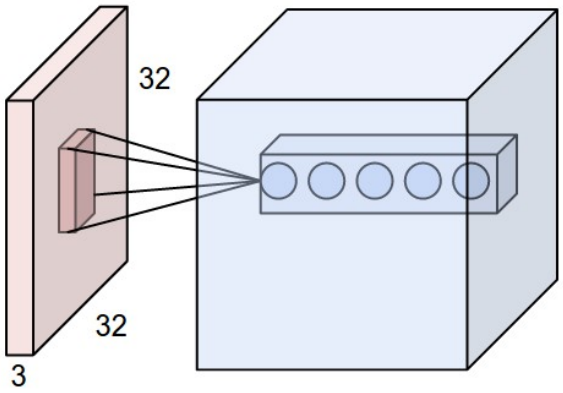


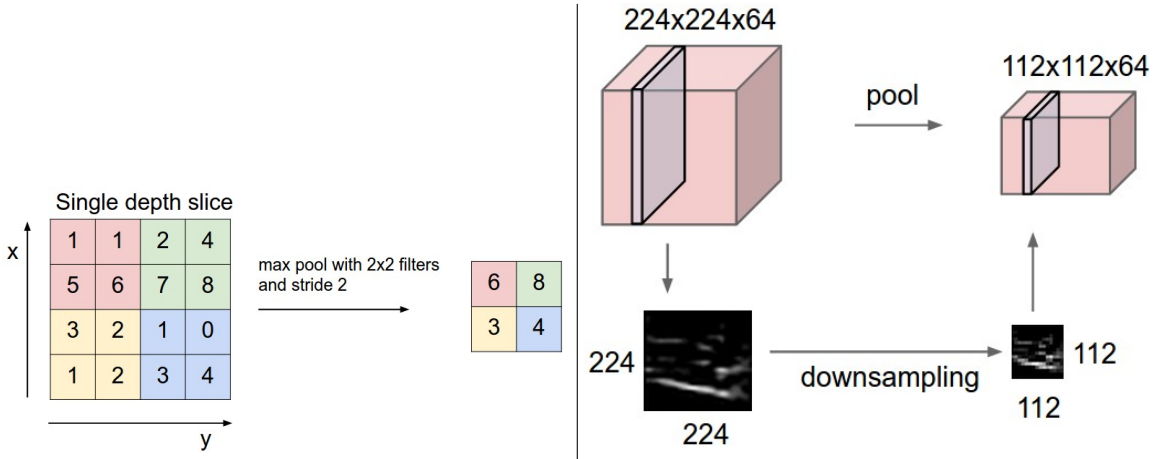

Figure 2.6: Convolutional layer with five different filters. Image sources: [21] [16]

At the end of the convolutional layer the output volume is fed into neurons similar to the one described in Section 2.2. All elements in the same depth are affected by the same bias term and fed into the activation function. The size of the volume is preserved in this step.

#### 2.2.3.2 Pooling layers

Pooling layers perform the downsampling of input volumes in both width and height dimensions. This is done by sliding a filter of fixed size over the input and doing a MAX operation on elements that the filter covers, and only the largest value element is copied into the output (Figure 2.7). Pooling is done on each depth slice separately from other slices, so depth size is preserved through the layer.

It is common to select pool size 2 x 2 and stride 2. Like this, inputs are downsampled by two in height and width dimensions, discarding 75 % of activations. Pooling layers therefore reduce the number of activations, and prepare them to be flattened out and fed into a fully connected layer.




(a) Max pooling operation (b) Effect of polling on input dimensions

Figure 2.7: Pooling layer. Image source: [16]

## 2.3 TensorFlow

TensorFlow is a free and open-source framework for numerical computation. It is particularly suited for large-scale Machine Learning applications [14]. It started as a proprietary project developed by a Google Brain team at Google in 2011, and became open-source in late 2015. It is used in many of Google's products such as Gmail, Google Cloud Speech and Google Search.

TensorFlow gives programmers tools for creating and training ML models, without needlessly diving into the specifics of computing Neural Networks. Programmers can write high level code in Python API, which calls a highly efficient C++ code. When using TensorFlow, the hardest part of an ML project is usually data preparation. After that is done, the creation of an ML model, its training and evaluation can be done in a few lines of Python code.

TensorFlow also supports Keras high level API for building ML models. Keras is a Python library that functions as a wrapper for TensorFlow. When building ML models developers can use Keras Sequential API, where each layer in a model is represented as one line of code. Users do not need to care about connections between the layers, they only need to choose the type of layer (convolutional, max pool, fully connected), its size and a few other specific parameters. Sequential API is used most of the time, but if a finer level of control is needed TensorFlow provides low level

math operations as well.

Finally, TensorFlow's trained output model is portable [14]. Models can be trained in one environment and executed in another. This means that we can train our model by writing Python code on a Linux machine and execute it with Java on an Android device. This last functionality is important for running ML models on microcontrollers.

### 2.3.1 TensorFlow Lite for Microcontrollers

TensorFlow Lite (TFLite) is a set of tools and libraries that enable running ML inferences on constrained devices [22]. It provides support for Android and iOS devices, and embedded Linux. TensorFlow Lite for Microcontrollers (TFLite Micro) is a recent port of TFLite (as of mid 2019), dedicated to running ML models on microcontrollers. TFLite itself provides API in different languages, such as Java, Swift, Python and C++. TFLite Micro uses C++ API, specifically C++11, which reuses a large part of the general TensorFlow codebase.

TFLite Micro library does not require any specific hardware peripherals, which means that the same C++ code can be compiled to run on a microcontroller or a personal computer with minimal changes. Users are only expected to implement their version of `printf()` function. As microcontroller binaries are usually quite big, flashing firmware to a microcontroller is a time consuming procedure. It makes sense first to test and debug a program that includes only ML inference specific code on a personal computer, before moving on to a microcontroller, to save time. Implementation of the test setup is described in 4.2.3.

The TFLite Micro library is available publicly as a part of a much larger TensorFlow project on GitHub [22]. To use the library for embedded development the whole project has to be cloned from the GitHub. The TensorFlow team provides users with several example projects that have been ported to several different platforms, such as Mbed, Arduino, OpenMV and ESP32. Example projects show how to use TFLite API, while showcasing different ML applications: Motion detection, wake word detection and person detection.

Is important to know that TFLite is just an extension of the existing TensorFlow project. General steps for creating a trained ML model are still the same as seen in Figure 2.1, although we have to be aware of some details. Figure 2.8 shows all steps that are needed to prepare an ML model for running on a microcontroller.

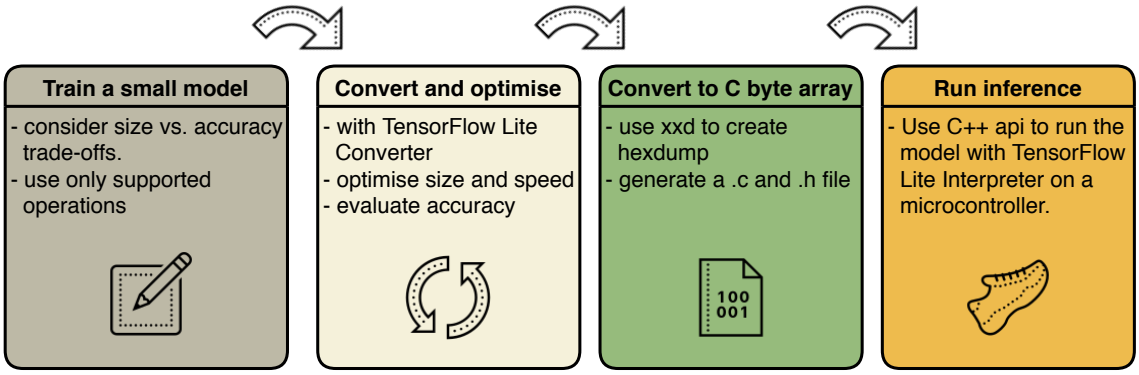



Figure 2.8: Workflow of preparing a ML model for an inference on a microcontroller. Icons source: [11]

We start with a small but inaccurate model that can still accomplish the basic criteria that our objective demands. When the end of this workflow process is reached and we made sure that the model can fit into a flash memory area of our target microcontroller, we can start training more complex models to increase accuracy. We are allowed to use only operations that have supported implementations on microcontrollers. This is usually not a restriction, as many of them are supported.

The model that is created is usually quite big, and needs to be converted with the TensorFlow Lite Converter tool. This tool provides a non-optimised conversion and several different optimised conversions.

To import and use the optimised model, we need to convert it into binary format, which is done with the command line tool xxd. The model is then ready to be executed on a microcontroller, we can run it and process the results. Accuracy will be the same as compared to running the same .tflite model on a personal computer, but execution time will naturally be different. If needed, we can tweak the model parameters, train a new model and repeat the described workflow again.

#### 2.3.1.1 Post-training quantization

By using quantization optimisation we approximate floating-point numbers in a different format, usually with 8-bit integers. When computing Neural Networks we can quantize weights, biases and intermediate values output by separate neurons. Quantization has a dramatic effect on the size of the model and its execution speed. By changing 32-bit floating-point numbers with 8-bit integers size decreases by a factor of 4. Floating-point math is by nature slow to compute, many microcontrollers do not even have a floating-point unit. In comparison integer math is faster to compute, therefore quantized models are executed faster. Model accuracy decreases after using quantization, but usually by less than a percent.

## 2.4 IoT and wireless technologies

The Internet of Things, or IoT, is a system of uniquely identifiable devices, which communicate with each other or other systems over wireless networks [23]. A device or a thing is a battery powered embedded system such as smart watch, heart monitor, or animal tracker which would transmit collected sensor data to an IoT gateway, which would relay the data over to the cloud. These data can then be analysed and displayed in such a fashion which would provide businesses or users with valuable information. Examples of this would be tracking the energy consumption of machines in factories, monitoring conditions of crops in agriculture, or monitoring locations of endangered species in African conservation parks.

An important part of the IoT system is a wireless network that is used to transport data from edge devices to gateways, or directly to the Internet. The choice of a wireless network is highly dependent on a type of problem an IoT solution is trying to solve. Factors such as required battery life, amount of data being sent, the distance that data have to travel and environment conditions of the edge device itself are important.

Because our early detection system demands a decent battery life of several months and needs to send a small amount of data over one or two kilometres, we will focus

on wireless technologies such as NB-IoT, Sigfox and LoRa.

Narrowband IoT or NB-IoT is a radio technology standard developed by the 3GPP standard organisation [24]. NB-IoT was made specifically with embedded devices in mind, it has a range of up to 15 km and it has deep indoor penetration [24]. Compared to Sigfox and LoRa it has better latency and a higher data rate, but also higher power consumption [25]. However it is unsuitable for our use case as it operates on the network provided by the cellular base towers, which is inconvenient as the mobile connection in Assam, India can be inconsistent [5].

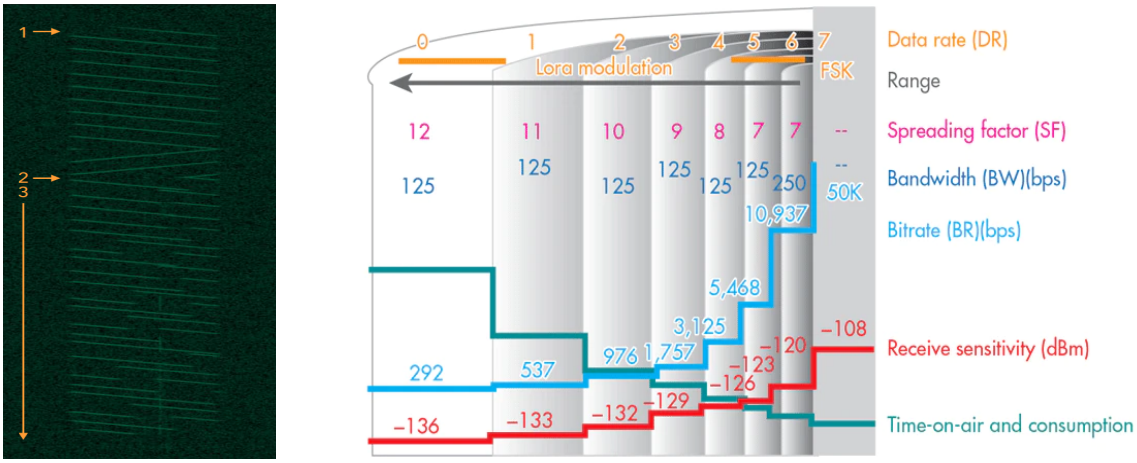
Sigfox is a radio technology developed by the company of the same name that operates on an unlicensed Industrial, Scientific and Medical (ISM) radio band. In many views it is similar to LoRa, as it has the comparative range and power consumption [25]. However, there are a few important differences. Although Sigfox modules are a bit cheaper when compared to Lora modules, each message is paid, devices are limited to 12 bytes per uplink, 140 uplinks per day and only 4 downlinks are available per day. Sigfox devices can also only communicate with base stations, installed by the Sigfox company [25]. This means that users can not build their own network and are dependent on the coverage provided by Sigfox.

This leaves us with the Lora protocol, which covers our use case from points view of long range, low power consumption and the ability to set up our own network.

### 2.4.1 LoRa and LoRaWAN

LoRa (Long Range) is a physical layer protocol that defines how information is modulated and transmitted over the air [26] [24]. The protocol is proprietary and owned by a semiconductor company, Semtech, who is the sole designer and manufacturer of Lora radio chips in the world. The LoRa protocol uses a modulation similar to chirp spread spectrum modulation [26]. As the protocol is proprietary, exact details of it are not known, although it was reverse engineered by a radio frequency specialist [27]. An example of a LoRa signal that was captured with a software defined radio can be seen in Figure 2.9a. Each symbol is modulated into a radio signal whose frequency is either increasing or decreasing with a constant rate inside of a specified bandwidth.

When the bandwidth boundary is reached, the signal "wraps around" and appears at the other boundary. Although the frequency is always changing with a constant rate, it is not continuous inside the bandwidth window, but it can change to a different frequency immediately and continue from there.




(a) LoRa signal

(b) Properties of LoRa signal

Figure 2.9: Lora signal (left) and different properties of LoRa with their effects on range, bit rate, receiver sensitivity, time on air and consumption (right). Image sources: [27] [28]

This kind of modulation gives LoRa extreme resiliency against the interference of other radio frequency signals that might be using the same frequency band [26] [28]. For example, on a lower part of Figure 2.9a we can see a signal with constant frequency transmitting inside the bandwidth window that the LoRa signal is using. This kind of interference is filtered out easily by a LoRa receiver.

The size of a bandwidth window, rate of frequency change (also known as a spreading factor) and transmitting power further define the LoRa signal. With these factors, we can influence the range, power consumption and bit rate of a LoRa signal. For example, as seen in Figure 2.9b, by increasing the spreading factor we increase the time on air, thus giving the receiver more time to sample the signal, which leads to better sensitivity, but increases power consumption.

While LoRa defines the physical layer, LoRaWAN defines the media access control protocol for wide area networks, which are built on top of LoRa [26]. Its specification is open, so anyone can implement it. LoRaWAN takes care of communication between

end-devices and gateways and manages communication frequency bands, data rates and transmitting power.

LoRaWAN has a star of stars topology [26]. Devices deployed in the field transmit messages on frequency bands that differ from region to region. Messages are received by gateways which relay them to the network server. The network server displays relayed messages, decodes them and sends them to various applications. If the same message is heard by several gateways, the server drops all duplicates. The server also decides which gateway will send a downlink message to a specific device.

Because LoRaWAN operates on an unlicensed ISM band, anyone can setup up their network without any licensing fees. For some use cases, a single gateway with an internet connection is enough to provide coverage to a large number of devices.

## 2.5 Thermal cameras

Thermal cameras are transducers that convert infrared (IR) radiation into electrical signals, which can be used to form a thermal image. A comparison between a normal and a thermal image can be seen in Figure 2.10. IR is an electromagnetic (EM) radiation, and covers part of the EM spectrum that is invisible to the human eye. The IR spectrum covers wavelengths from 780 µm to 1 mm, but only a small part of that spectrum is used for IR imaging (from 0.9 µm to 14 µm) [29]. We can classify IR cameras broadly into two categories: photon detectors or thermal detectors [29]. Photon detectors convert absorbed EM radiation directly into electric signals by the change of concentration of free charge carriers [29]. Thermal detectors covert absorbed EM radiation into thermal energy, raising the detector temperature [29]. The change of the detector's temperature is then converted into an electrical signal. Since photon detectors are expensive, large and therefore unsuitable for our use case, we will not describe them in greater detail.

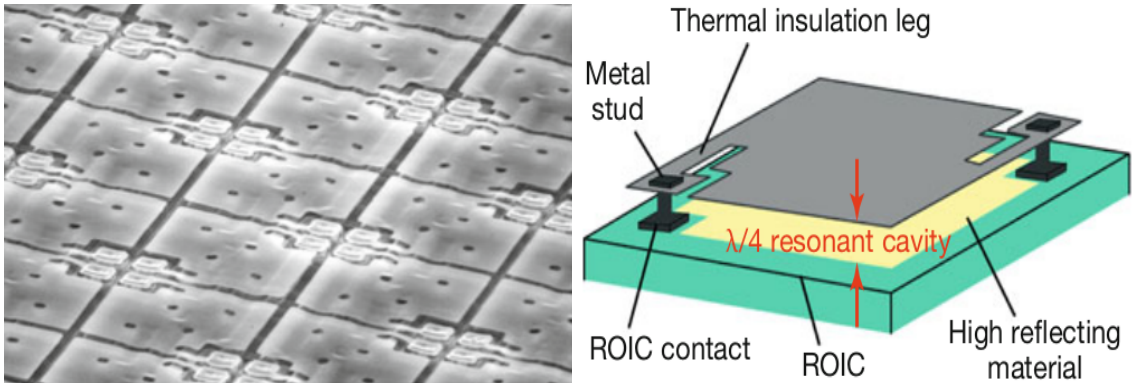
Common examples of thermal detectors are thermopiles and microbolometers. Thermopiles are composed of several thermocouples. Thermocouples consists of two


Figure 2.10: Comparison between a normal image and thermal image. Image source: Arribada Initiative [30]

different metals joined at one end, which is known as the hot junction. The other two ends of the metals are known as cold junctions. When there is a temperature difference between the hot and cold junctions, a voltage proportional to that difference is generated on the open ends of the metals. To increase voltage responsivity, several thermocouples are connected in series to form a thermopile [29]. Thermopiles have lower responsivity when compared to microbolometers, but they do not require temperature stabilisation [29].

Microbolometers can be found in most IR cameras today [29]. They are sensitive to IR wavelengths of 8 to 14 µm, which is a part of the longwave infrared region (LWIR) [29]. Measuring part of a microbolometer is known as Focal Point Array (FPA) (Figure 2.11a). FPA consists of IR thermal detectors, bolometers (Figure 2.11b), that convert IR radiation into an electric signal. Each bolometer consists of an absorber material connected to a readout integrated circuit (ROIC) over thermally insulated, but electrically conductive legs [31].

Absorber material is either made out of metals such as gold, platinum, titanium, or, more commonly, out of semiconductors such as vanadium-oxide (VOx) [31]. The important property of absorber materials is that electrical resistance changes proportionally with the material's temperature [29]. When IR radiation hits absorber material, it is converted into thermal energy, which raises the absorber's temperature, thus changing its resistance. To detect the change in resistance, ROIC applies a

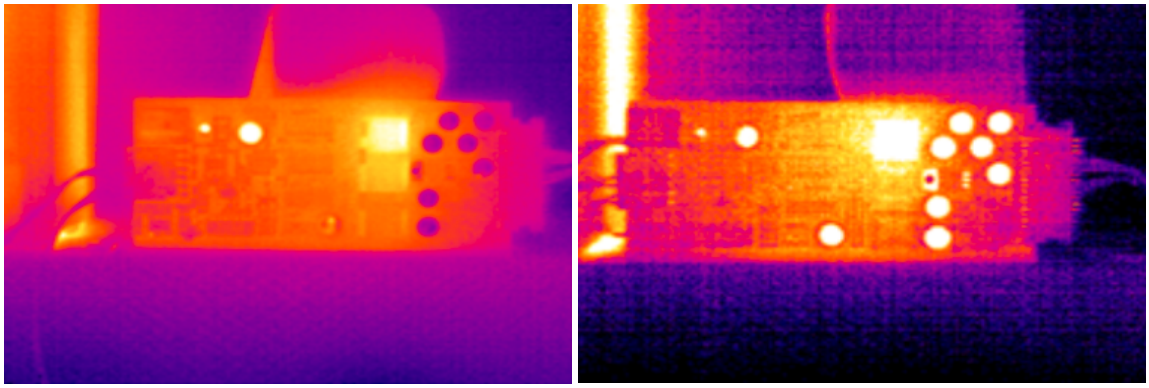



(a) Focal point array (b) Bolometer

Figure 2.11: (a) Focal point array under electronic microscope. (b) Bolometer with $\lambda/4$ resonant cavity. Image source: Vollmer, Möllmann [29]

steady-state bias current to the absorber material, while measuring voltage over the conductive legs [29].

When deciding between different types of thermal cameras we are often comparing them in terms of the cost, size and image resolution. One important property that also has to be taken into account is temperature sensitivity, also known as Noise Equivalent Temperature Difference (NETD). NETD is measured in mK, and tells us the minimum temperature difference that can still be detected by a thermal camera. In microbolometers, NETD is proportional to the thermal conductance of the absorber material, among other factors [29]. The thermal conductance of bolometers is minimised by enclosing FPA into the vacuum chamber, thus excluding thermal convection and conduction due to the surrounding gases. The only means of heat transfer that remain are radiant heat exchange (highly reflective material below the absorber is minimising its radiative losses), and conductive heat exchange through the supportive legs. NETD also depends on the temperature inside the camera, as higher ambient temperatures can raise the internal temperature, thus increasing the NETD and noise present in the thermal image. Today's thermopiles can achieve NETD of 100 mK, microbolometers 45 mK, while photon detectors can have NETD of 10 mK. Although tens of mK does not seem a lot, we can see in Figure 2.12 what a difference of 20 mK means for image resolution and noise.



(a) NETD is 60 mK (b) NETD is 80 mK

Figure 2.12: Comparison of images of the same object taken with cameras with different NETD values. Low NETD values are more appropriate for object recognition. Image source: MoviTherm [32]

### 2.5.1 Choosing the thermal camera

The choice of thermal camera was made by the Arribada Initiative [30]. They tested several different thermopiles and microbolometers while searching for the desired properties. The camera had to be relatively inexpensive, and small enough so that it could be integrated into a relatively small housing. The main property that they searched for was that elephants could be recognised easily from thermal images. That meant that the camera needed to have decent resolution and low NETD. Cameras were tested in Whipsnade Zoo and the Yorkshire Wildlife Park where images of elephants and polar bears could be made.

They tested two thermopile cameras (Heimann 80x64, MELEXIS MLX90640) and two microbolometer cameras (ULIS Micro80 Gen2, FLIR Lepton 2.5). Although thermopile cameras were cheaper than microbolometer cameras, the quality of images they produced was inferior, as can be seen in Figure 2.13.

The MELEXIS MLX90640 camera had a resolution of 32 x 24 pixels and NETD of 100 mK, while the Heimann camera had a resolution of 80 x 64 pixels and NETD of 400 mK. It was concluded that images taken by either one of the thermopile cameras could not be used for object recognition, merely only if the object was present or not [30].

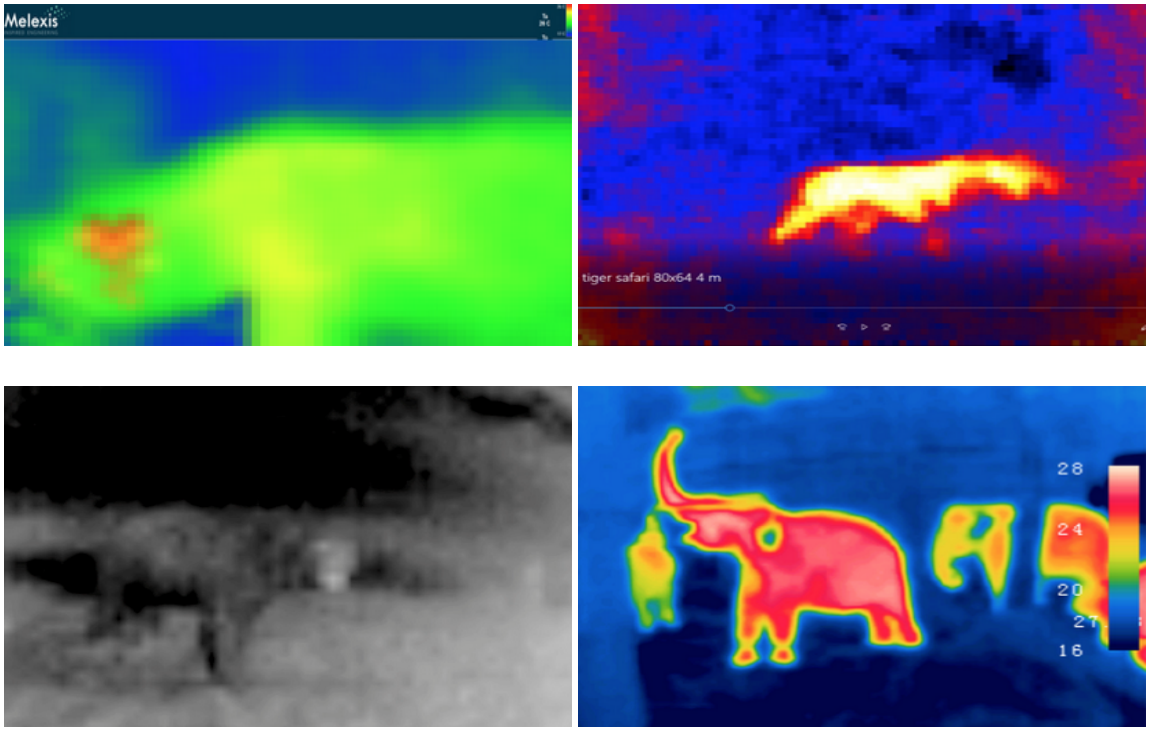



Figure 2.13: Comparison of image quality made by different thermal cameras, MELEXIS MLX90640 (top left), Heimann 80x64 (top right), ULIS Micro80 Gen2 (bottom left) and FLIR Lepton 2.5 (bottom right). Image source: Arribada Initiative [30]

Microbolometers produced better results. Both the Ulis Micro80 and FLIR Lepton had a similar resolution, 80 x 80 and 80 x 60 respectively, but the Ulis Micro80 had two times bigger NETD compared to the FLIR Lepton camera, 100 mK and 50 mK, respectively. Images produced by the FLIR Lepton were much cleaner, so it was chosen as an appropriate camera for the task.

It is important to note that the FLIR Lepton, like all microbolometers, requires frequent calibration to function properly. In temperature non-stabilised cameras small temperature drifts can have a major impact on image quality [29]. Calibration is done either by internal algorithms of the camera or by exposing the camera to a uniform thermal scene. The FLIR Lepton camera comes with a shutter, which acts as a uniform thermal signal and enables regular calibration. Calibration in the FLIR Lepton is by default automatic, triggering at startup, and every 3 minutes afterwards or if camera temperature drifts by more than 1.5 °C.

The FLIR Lepton camera comes in two versions, 2.5 and 3.5. Both cameras function the same and have exactly the specifications, they only differ in resolution: the 3.5 has a resolution of 120 x 160, while the 2.5 has 60 x 80. Both were used in the process of image collection.

# 3 Neural network model design

In this chapter we describe the design of a convolutional neural network that can process thermal images and predict what object they contain. The workflow that we followed will largely be a combination of workflows presented in Figures 2.1 and 2.8.

We first had to set concrete objectives, while keeping in consideration various constraints. Tools and development environment that were used in the process are then described. Methods of dataset creation are described afterwards, first the dataset that was created by Arribada Initiative, then dataset provided by us.

We then explored both datasets, analyzed different class representations and decided, if they are appropriate for accomplishing objectives that we set earlier.

In the image preprocessing phase we imported images and connected them with metadata that was parsed from the excel database. We analyzed the dataset, split it into different sets and applied image correction procedures. We then decided on a rough CNN architecture with variable hyperparameters and ran a random search algorithm, which searched for best performing models based on accuracy.

We finish this chapter by going again through the same design process, but this time using tools provided by Edge Impulse.

## 3.1 Model objectives

The accuracy of our early detection system should be equal or similar to the one of the human observers, no matter if it is operating in daytime or nighttime. Although the system will be placed on the paths that are regularly traversed by elephants,

they are not the only possible objects that can appear on taken thermal images. Humans and various livestock, such as goats and cows, could also be photographed. Reporting false positives should be avoided, which means that the system should not incorrectly label a human or a livestock animal as an elephant. At the same time, false negatives also need to be avoided, as an elephant could pass the system undetected. These kinds of mistakes could undermine the community's confidence in the early detection system and defeat the purpose. This means that besides elephant detection, we should also focus on correctly classifying humans and livestock while providing a nature/random class for all other unexpected objects or simply images of nature.

It would be beneficial, if the thermal camera can take several images of the same object in a short time, thus increasing the confidence of the computed label of the object. However, this is constrained by the image processing time and the camera's field of view. Thermal camera FLIR Lepton has a horizontal field of view of 57 degrees. The closer object passes by a thermal camera, the quicker it traverses the camera's field of view, thus giving the camera less time for capture. This problem can be solved by minimizing the execution time of the ML model or by placing the early detection system on a position that is several meters away from the expected elephant's path. As the latter option might not be always possible, we should strive to keep the whole image processing time as short as possible.

Finally, as our neural network has to run on a microcontroller and not on a computer or a server, we have to keep it lightweight in terms of memory. Extra model complexity that brings few percents of accuracy does not matter much if the model is too large to fit on a microcontroller or takes too long to run.

To summarize:

- We will create an image classification ML model that will be capable of processing a thermal image and sorting it into one of 4 possible categories: elephant, human, livestock, and nature/random.
- Total image processing time should be as short as possible, we should try to

keep it under 1 second.

- Model should be small enough to fit on a microcontroller of our choice, while still leaving some space for application code. The microcontroller of our choice (STM32F767) has 2 MB of flash memory so the model size should be smaller than that.

## 3.2 Tools and development environment

All of the work connected with image preparation and ML model creation was done in Python 3.6, Numpy was used for image preprocessing, Pandas for Excel database manipulation, and Matplotlib for plot generation. Neural networks were designed in TensorFlow 2.4, using Keras high-level API, Keras Tuner model was used for hyperparameter search.

As training neural networks is a computationally demanding process, it would not be feasible to do it on a personal laptop. Amazon's Elastic Compute Cloud web service was instead used. Elastic Compute Cloud or EC2 enables users to create an instance of a server in a cloud with a specified amount of processing power and memory. Some instances come with dedicated software modules and dedicated graphics cards for an extra boost in performance. We created an instance of a Linux server that came with TensorFlow, Numpy, and other libraries pre-installed. Interaction with servers was done one command line through SSH protocol.

Instead of writing Python scripts and executing them through the command line, we used Juptyer Notebook. Juptyer Notebook is a web-based application that can run programs that are a mix of code, explanatory text, and computer output. Users can divide code into segments, which can be executed separately, visual output from modules such as Matplotlib is also supported. To use Juptyer Notebook on our cloud instance, we had to install it and run it. We could then access web service simply through a web browser by writing the IP address of the server, followed by the default Juptyer Notebook server port, 8888.

## 3.3 Creating the dataset

As mentioned in section 1.3, major part of thermal image dataset was provided by Arribada Initiative [12] [13]. Images in the dataset come from two different locations: Assam, India, and ZSL Whipsnade Zoo, United Kingdom.

Assam served as a testing ground. Arribada team positioned two camera traps on two locations that overlook paths commonly used by elephants. Cameras were built out of Raspberry Pi, PIR sensor, FLIR Lepton 2.5 camera, and batteries, all of which were enclosed in a plastic housing. Insides of the camera and an example of a deployed camera can be seen in Figure 3.1.

Figure 3.1: Camera trap used in Assam, India. Image source: Arribada Initiative [13]

PIR sensor functioned as a photo trigger, whenever an object passed in front of it, the camera made an image. This setup provided Arribada with elephant images in real-life scenarios, however, they could not capture elephants in a variety of different conditions. It is important to create an image dataset, where the object can be seen in different orientations, distances, angles, and temperature conditions. Models that were trained on diverse datasets end up being much more robust and therefore perform better on never before seen image data, when deployed in real life.

This was accomplished in ZSL Whipsnade Zoo, where they took many images of elephants in a variety of different conditions [33]. With elephants in the enclosure, researchers could move cameras around and get images that were needed. PIR sensor trigger approach was dropped in favor of a 5 second time-lapse trigger. Two cameras were used again, however, one of them now used FLIR Lepton 3.5 camera with better

resolution.

Images of elephants that came from both locations can be seen in Figure 3.2.

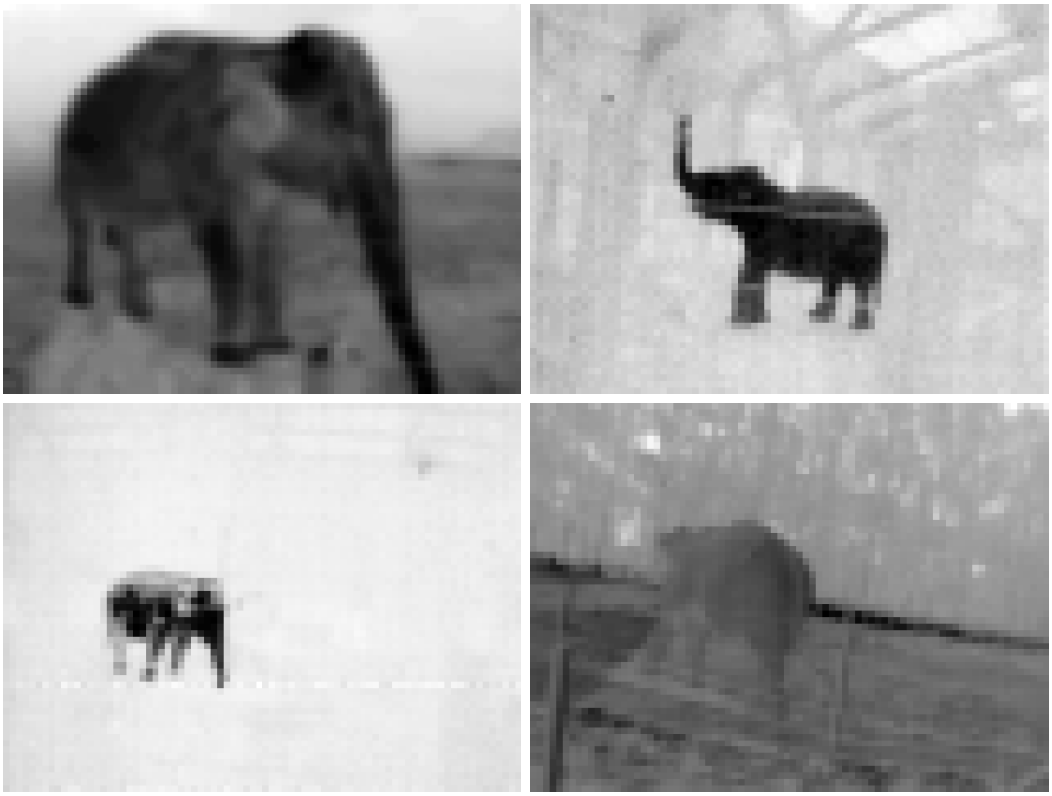


Figure 3.2: Thermal images of elephants from dataset.

Small part of thermal image dataset was provided by us. This was done because the number of images of cows was low compared to the number of human and elephant images and because we also did not have any images that could be used for nature/random class. We wanted to gather images as quickly and efficiently as possible so we build a prototype camera made out of FLIR Lepton 2.5 breakout board, Raspberry Pi Zero, and power bank. We used an open-source library [34] for the FLIR Lepton module which used a simple C program to take a single image with a thermal camera and save it to a Raspberry Pi. The image of the setup can be seen in Figure 3.3.

We wrote a simple Python script that executed the C program every time we pressed the trigger push-button. An additional shutdown button was added to call the Raspberry Pi shutdown routine, as forcibly removing power from it would corrupt

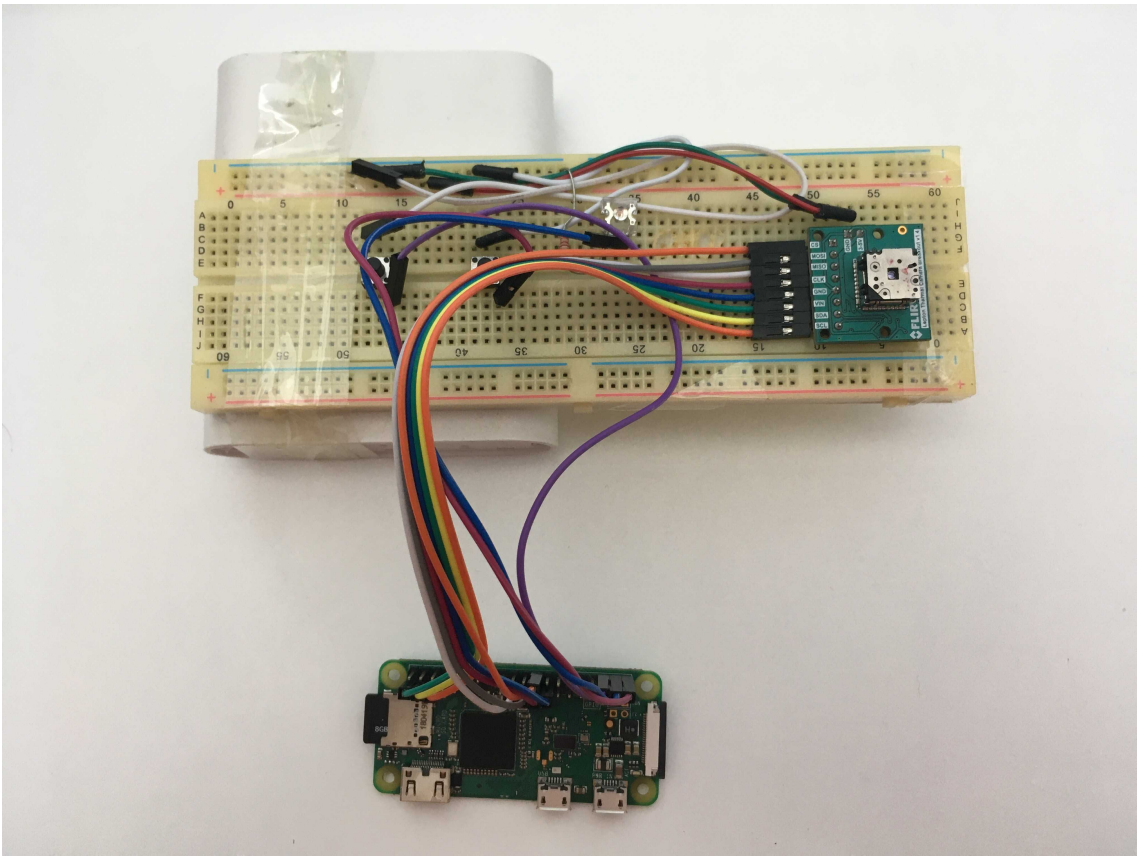


Figure 3.3: Camera setup used for taking thermal images with FLIR Lepton 2.5.

freshly taken thermal images on the Raspberry Pi's SD card.

With this setup, we made 365 images of cows in varying conditions, 308 images of nature, and 124 images of humans that were made on the go. We then manually sorted images into appropriate folders and added them to the dataset.

## 3.4 Exploring the dataset

Thermal image dataset created by Arribada was given to us in form of a Google Drive folder, which we downloaded to our computer.

After examining the folder, we came to several conclusions.

1. We saw that the primary focus of the Arribada team was to build an object localization model, not an image classification model. In object localization, the neural network draws bounding boxes around objects that it recognizes and assigns them labels, while the image classification model only labels the image

as a whole. Object localization produces a bigger and more complex model than image classification and it is unsuitable for running on a microcontroller. All major work that was done by the Arribada team was contained in one folder where each image had an accompanying text file of the same name. Text files were produced by a DeepLabel software, which is used for preparing images for training object localization models. Each line in a text file described the location of the bounding box and its label. This dataset format was not suitable for us, as many images contained more bounding boxes, which would be troublesome to sort into a distinct label.

We later saw that there were a few folders with names such as "Human", "Single Elephant", "Multiple Separate Elephants", "Multiple obstructing Elephants", "Cows", "Goats" and so on, which contained sorted images that we could use. we merged all folders with elephant pictures into one folder, as we did not care if the model can differentiate how many elephants are on a taken image, we only wanted to know if there are any elephants on it or not.

2. We found out that all images were documented in a large Excel database. For each image, there was a row in a database that connected the image file name with the information where the image was taken and with what sensor. This enabled us to generate a graph seen in Figure 3.4.

   We used a total of 13667 images from the thermal image dataset, almost 88 % of them were made in Whipsnade Zoo, the rest of them were made in Assam. All images from Assam were made with FLIR Lepton 2.5, while both cameras were used in Whipsnade zoo, however, more photos were made with the 2.5 version of the thermal camera.

3. After manually inspecting the folder with goat images we saw that it mostly contained images of a herd of goats, standing around a single elephant. This folder was usable only for object localization ML models, where each goat could be tagged with a bounding box. In the case of an image classification model, this sort of training data is not desirable, as it would be too similar to another separate class, in our case elephant class. We therefore dropped goat images

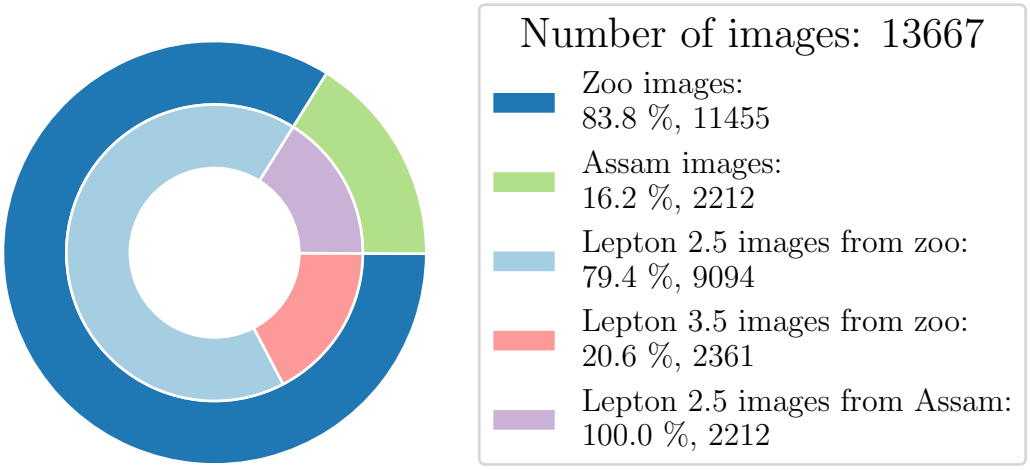



Figure 3.4: Distribution of used images from thermal dataset depending on image location and type of sensor.

out of our training data entirely. Livestock class was replaced with cow class.

4. We also realized that there was a large class imbalance, as seen in Figure 3.5 in favor of elephant class.

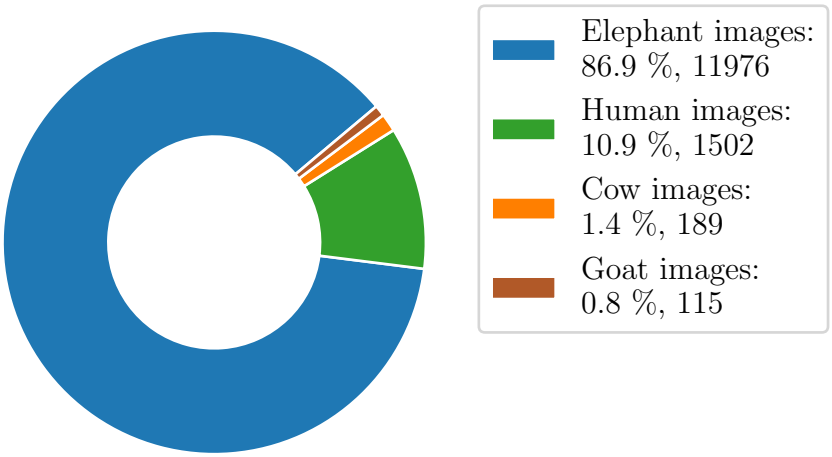



Figure 3.5: Class distribution of thermal images.

The number of elephant images was more than 4 times larger than the number of images of the all other classes combined. We solved this issue by acquiring more images of the minority class and oversampling the minority class.

## 3.5 Image preprocessing

The image preprocessing phase is a pipeline process that differs from project to project. Our process can be seen on Figure 3.6.

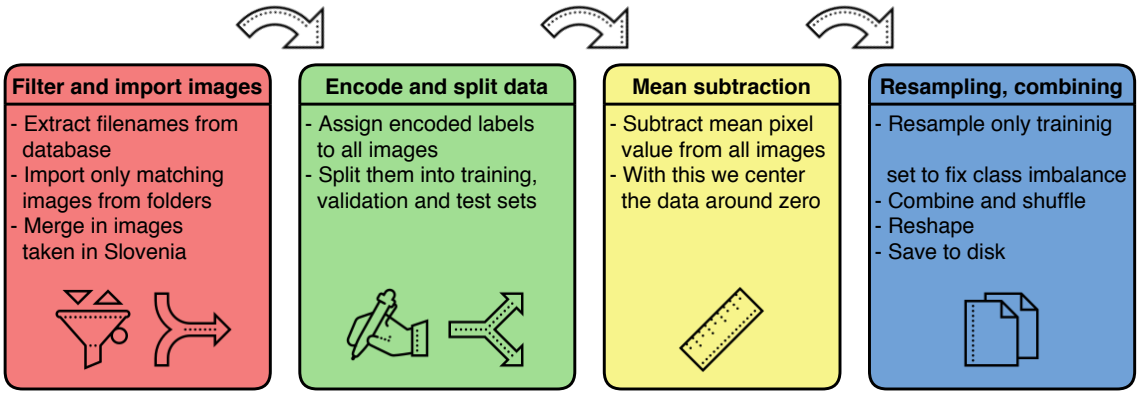



Figure 3.6: Image preprocessing pipeline. Icons source: [11]

At the start of the process, we compared filenames of each separate folder to the list of filenames found in the Excel database. We imported only the images found in both sources, as lists were not identical and we wanted to keep track of different metadata information. As some images were made with two different FLIR Lepton cameras with different resolutions (60 x 80 and 120 x 160), we downscaled higher resolution images directly in the importing process. After this, we added images that were taken by us in Slovenia. At this point, we had four separate Numpy arrays, one for each class, with 3 dimensions: the first dimension stored a number of different images in that class, second and third dimensions stored image's pixel values (60 and 80 pixels respectively).

The next step was assigning labels to each image. As the output of NNs are numbers, we can not just assign labels in strings format to data. Instead, we assigned every image a single number that represented that class, 0 for an elephant, 1 for a human, 2 for a cow, and 3 for a nature/random class. We shuffled images inside of each class and then split them into training, validation and test sets.

The training set was used for model training, while the validation set helped to choose the best model based on accuracy. The test set is normally set aside and

used only at the end, after the model is chosen, to asses how the model performs on never seen data. If we did not use the validation set and only chose the best model according to the test set, we would be overfitting a model and we would have no accurate measure of how well would our model perform on unseen data.

At end of this step, we had 4 different Python dictionaries for each class. Each dictionary had 3 key-value pairs for every training, validation, and test set, which held image data and encoded labels.

We next applied the simplest form of normalization to all images, a mean subtraction. We calculated a two-dimensional matrix that held mean values of pixels averaged over the whole training set, which we subtracted from all images, essentially zero centering the data. This is a common preprocessing step in every ML image preprocessing pipeline, which is usually combined with standardization[1].

We achieved this by resampling the human, cow, and nature/random classes. The human class was resampled 5 times, while both cow and nature/random classes were resampled 8 times. Figure 3.7 shows the distribution of training images before and after resampling.

We only resampled training sets, not validation or test sets. If we resampled everything, the model would be seeing the same image several times during testing, thus reporting incorrect accuracy in the validation and test phase.

After resampling we merged and shuffled all data, and saved it to the disk for later use.

[1]Standardization scales the whole range of input pixel values into -1 and 1 interval. This is only needed if different input values have widely different ranges [16]. Because images that were created with FLIR camera were all 8-bit encoded, therefore had the same range, this was not needed.

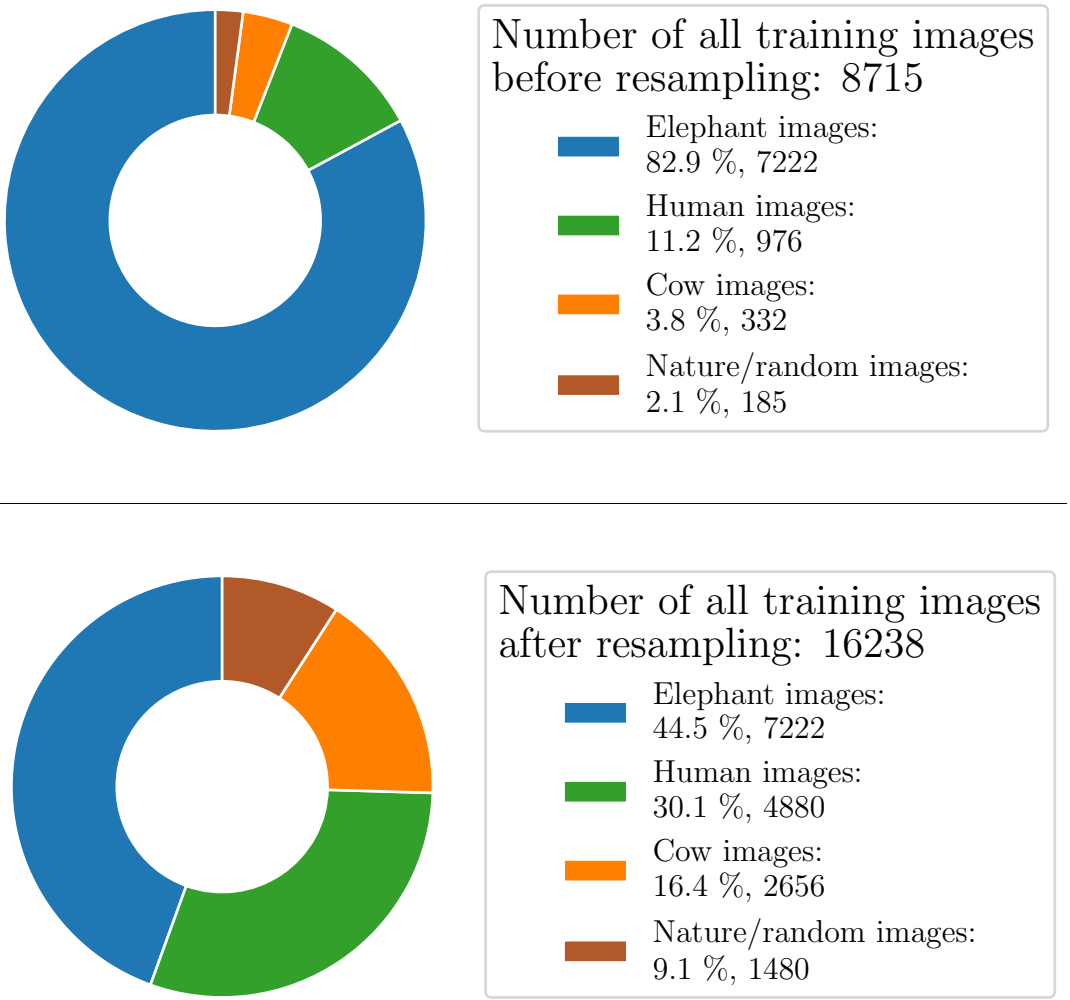



Figure 3.7: Distribution of training images before and after resampling.

## 3.6 Model creation and training

For the creation of CNN models, we used Keras Sequential API and Keras Tuner module. Sequential API abstracted many low-level details of model design. While specifying layers, we only had to specify what type of layer we wanted, its size and layer-specific features. We did not had to keep track of any connections between or in layers, this was automatically done by Keras.

For a model architecture, we decided to use a simplified version of a common CNN architecture that was shown in Figure 2.4. The best way to present the model is by

inspecting the Sequential API code that creates it, code is shown in Figure 3.8.

The model consisted of two pairs of convolutional and max-pooling layers, followed by a final convolutional layer. For activation function ReLu was chosen, as it is currently the most effective and popular option [16] [14]. The padding option was set to same, which meant that a spatial dimension of a volume would not change before and after a convolutional layer. Polling layer kernel size was set to 2 x 2, with a default stride of 2.

The output volume of the last convolutional layer was flattened out into a single vector and fed into a dense layer, which was followed by a dropout layer[2].

The last dense layer was a final output layer with only 4 neurons, each one representing one class. Softmax activation was used to calculate class probabilities. Model was set to use Adam optimizer and sparse categorical crossentropy loss function. Adam is an upgraded version of gradient descent method, which automatically adapts learning rate to decaying gradients [14]. It is generally easier to use than gradient descent as it requires less tuning or learning rate hyperparameter. Sparse categorical crossentropy loss function is used when building a multi-class classifier.

Above set hyperparameters follow general rules of thumb and serve as a good starting point when building CNNs [16]. However, hyperparameters such as the number of filters, filter size, size of a hidden dense layer, dropout rate, and learning rate are specific to each dataset and can not be chosen heuristically.

To find hyperparameters that would yield the highest accuracy we used the Keras Tuner module. Exact configuration of Keras Tuner module and comparison of trained models is presented and discussed in section **??**.

[2]Dropout layer decides with probability $p$ in each training step how many activations from the previous layer will be passed on to the next layer. It is active only during the training phase, during the testing phase activations are multiplied with $(1 - p)$ factor to compensate. It is a very popular type of regularization technique, which makes models more robust to the input data [14].

```
        model = models.Sequential()

        model.add(Conv2D(FilterNum1, FilterSize,
                         activation='relu',
                         padding="same",
                         input_shape=(60,80, 1)))

        model.add(MaxPooling2D((2, 2)))

        model.add(Conv2D(FilterNum2, FilterSize,
                         activation='relu',
                         padding="same")

        model.add(MaxPooling2D((2, 2)))

        model.add(Conv2D(FilterNum3, FilterSize,
                         activation='relu',
                         padding="same")

        model.add(Flatten())

        model.add(Dense(DenseSize, activation='relu'))
        model.add(Dropout(DropoutRate))
        model.add(Dense(4), activation='softmax')

```

Figure 3.8: CNN architecture written in Python using Keras Sequential API.

## 3.7 Model optimization

Keras supports saving models in h5 format, which model's architecture, values of weights, and information used while compiling the model. h5 format can not be used directly for running trained models on mobile devices and microcontrollers, conversion to a .tflite format has to be done with the TFLite Converter tool.

The TFLite converter can convert a model in .h5 format into four differently optimized tflite models:

- **Non-quantized tflite model,** no quantization, just basic conversion from .h5 to .tflite format is done.
- **float16 model,** weights are quantized from 32-bit to 16-bit floating-point values. The model size is split in half and the accuracy decrease is minimal, but there is no boost in execution speed.
- **dynamic model,** weights are quantized as 8-bit values, but operations are still done in floating-point math. Models are 4 times smaller and execution

speed is faster when compared to float16 optimization but slower from full integer optimization.

- **Full integer model,** weights, biases, and math operations are quantized, execution speed is increased. It requires a representative dataset at conversion time.

A full integer model is an ideal choice for running models on microcontrollers, however, it should be noted that not all operations have full integer math support in TFLite Micro.

Furthermore, created tflite models need to be converted into a format that is understandable to C++ TFlite API running on a microcontroller. This is done with the **xxd**, a Linux command-line tool that creates a hex dump out of any input file. By setting `-i` flag, xxd tool creates a hex dump of our model and formats it as a char array in C programming language.

To automate the optimization process we wrote a Python script that took the model in raw .h5 format and converted it into every possible version of the optimized tflite model. Each model was then processed with xxd tool and pairs of .c and .h files were created, ready to be included in our application code.

## 3.8 Neural network model design in Edge Impulse Studio

Designing a neural network with Edge Impulse is a much less involved process than the one we described above, as many steps of image preprocessing are automated. To start with NN design, we first had to upload our image data to the Edge Impulse Studio project. This can be done either by connecting an S3 bucket[3]with data with the Edge Impulse account and transferring data to a specific project or by using Edge Impulse command-line tools to upload image data from a computer directly to a project. We chose the S3 bucket approach, once the data was uploaded it was trivial to transfer it to different projects.

[3]Simple Storage Service or S3 is another service provided by Amazon, used for storing a large

After the data was uploaded, the rest of NN design was done through Edge Impulse web interface. In Figure 3.9 we can see the so-called Impulse Design tab, where we design by selecting different blocks. With input block, we tell what kind of data are we inputting, either image or time-series data, and with processing block we decide how are we going to extract features. With learning block, we can choose to use a neural network provided by Keras, an anomaly detection algorithm, or a pre-trained model for transfer learning.

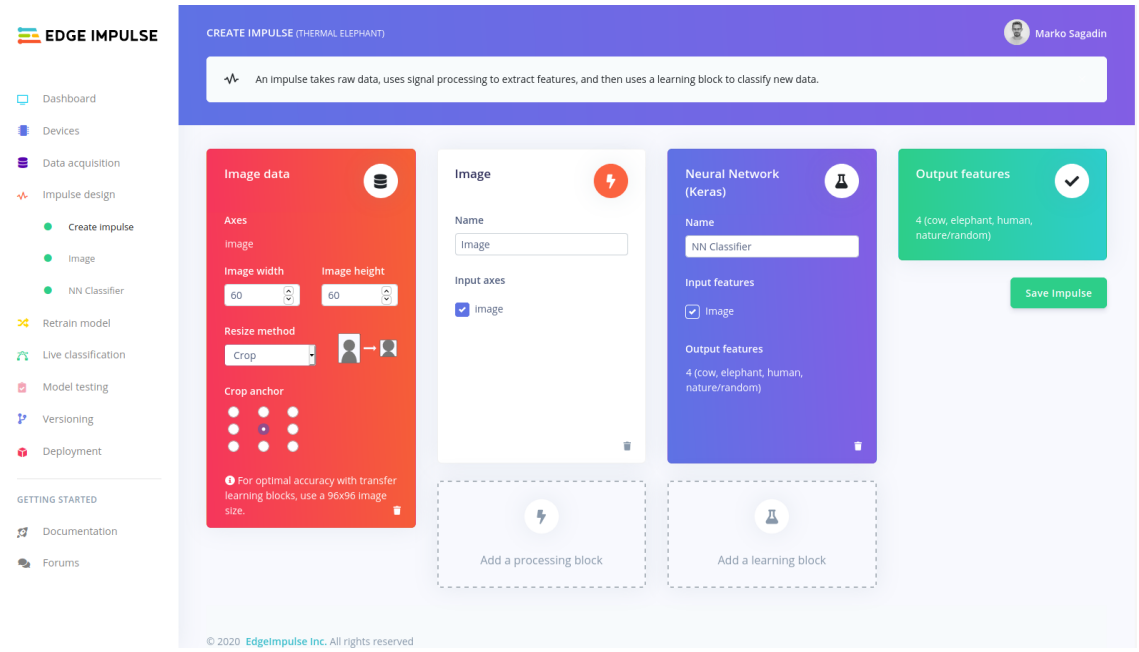



Figure 3.9: Creating a neural network in Edge Impulse Studio.

Since we were training with image data, we selected an image input block. As Edge Impulse did not support images of different image ratios at the time, we had to crop our images to 60 x 60 pixels. For processing block, we selected the image processing block as this was the only possible choice and for learning block, we selected Keras's neural network block.

Neural network block is configurable, we could either define our network with different blocks representing layers or switch to the text editor with Keras Sequential API code, where we could do our adjustments. Settings such as learning rate, number of epochs, and confidence rating are also available regardless of the option we chose.

amount of data in the cloud.

Training of neural networks inside Edge Impulse Studio is done in a cloud, so as users we do not have to worry about settings up a development environment. We only had to start it and wait for it to finish.

After training was done, Edge Impulse showed how well the model was performing on the validation data, how much flash and RAM would it need, and approximately how long will on-device inference take, based on the frequency and the processor of a microcontroller.

An example of the output is presented in Figure 3.10, inferencing time is estimated for a Cortex-M4 microcontroller, running at 80 MHz. Edge Impulse also does conversion to an optimized full-integer model automatically.

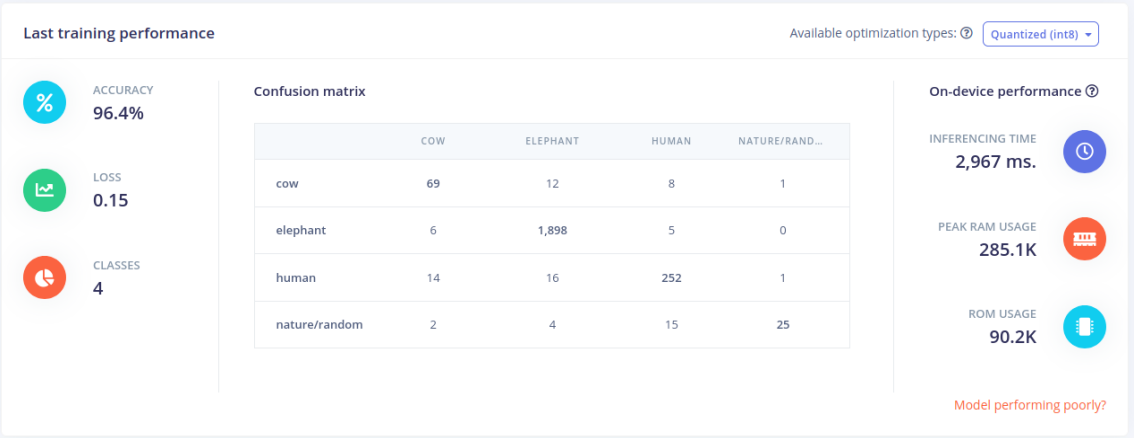



Figure 3.10: Training performance report.

The final step was deploying the trained model to the microcontroller. This step is fairly simple, Edge Impulse provides few example projects on their GitHub for different platforms that it supports. As we wanted to compare the performance of the models on ab STM32f767ZI, we chose the Mbed platform. We copied the example Mbed project from GitHub and in Edge Impulse Studio we selected to generate an inferencing library with our model for the Mbed platform. We extracted the library, which consisted of C++ files, into an example project and compiled it. An example project just continuously runs the inference on one image and outputs results over the serial port. A performance comparison between this example project and our implementation is done in section TODO ADD REFERENCE.

# 4 Planning and design of early warning system

General structure and tasks of an early detection system were already described in chapter 1.2. 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 4.1

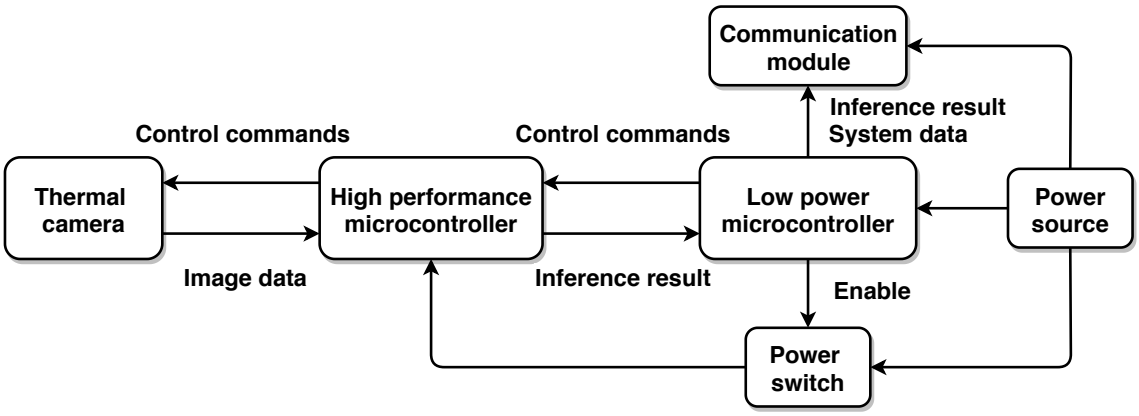



Figure 4.1: General block diagram of an embedded system

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 4.2.

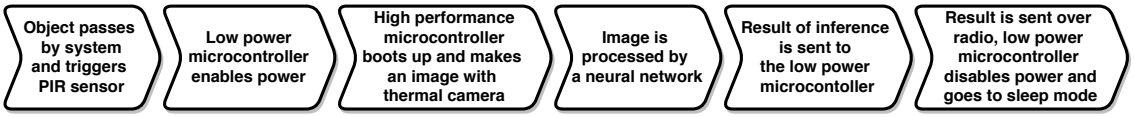



Figure 4.2: Diagram describing behavior of embedded early detection system

## 4.1 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 4.3. 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.

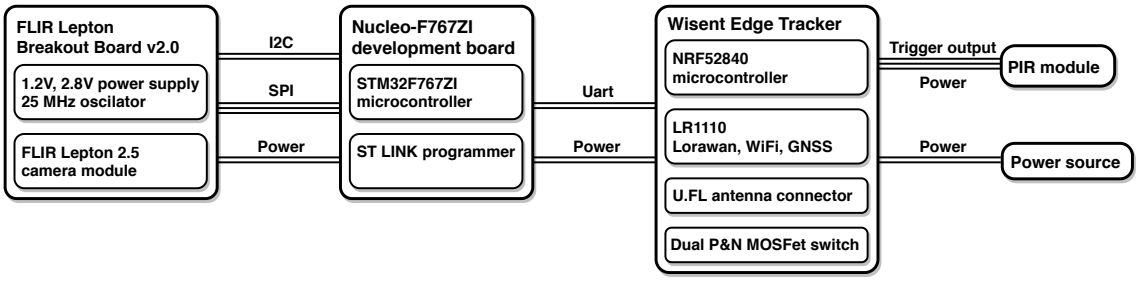



Figure 4.3: Hardware diagram of embedded early detection system

### 4.1.1 Nucleo-F767ZI

Nucleo-F767ZI (seen on Figure 4.4) 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.

### 4.1.2 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-poaching

[1]As a part of OpenCollar project, the design of Wisent Edge is open-source and available on GitHub [35], alongside other hardware and firmware tracker projects.

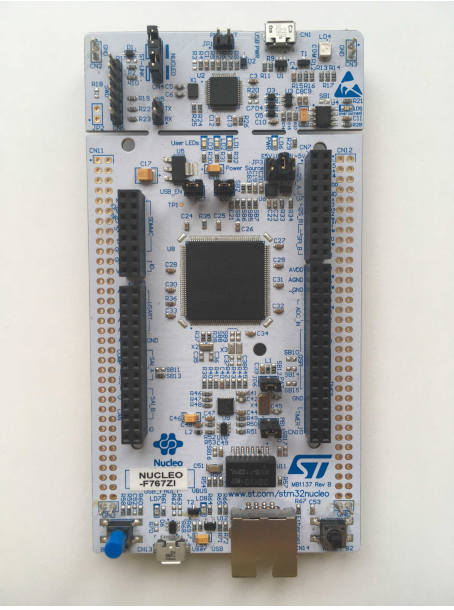



Figure 4.4: Nucleo-F767ZI development board

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 [36].

channel p and n MOSFET, circuit can be seen on Figure 4.5. 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.

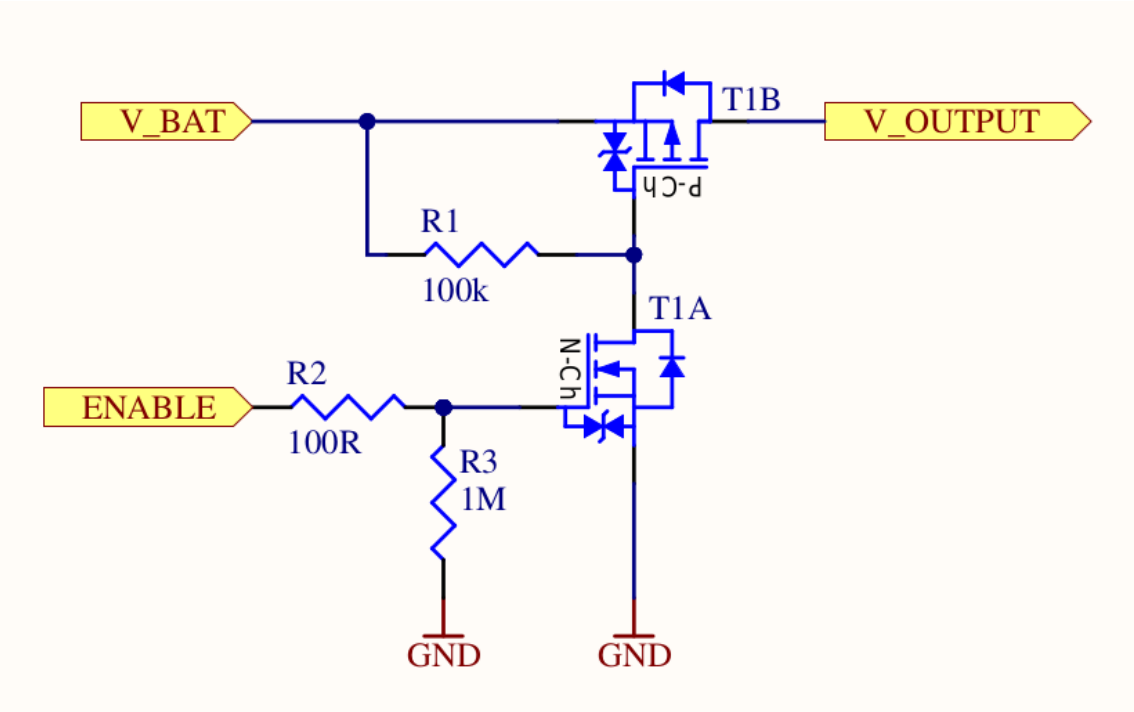



Figure 4.5: Dual P and N MOSFET configuration for power switching

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

In section 2.5 it was described what kinds of thermal cameras exist and how do they work, and in section 2.5.1 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 4.6. 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.

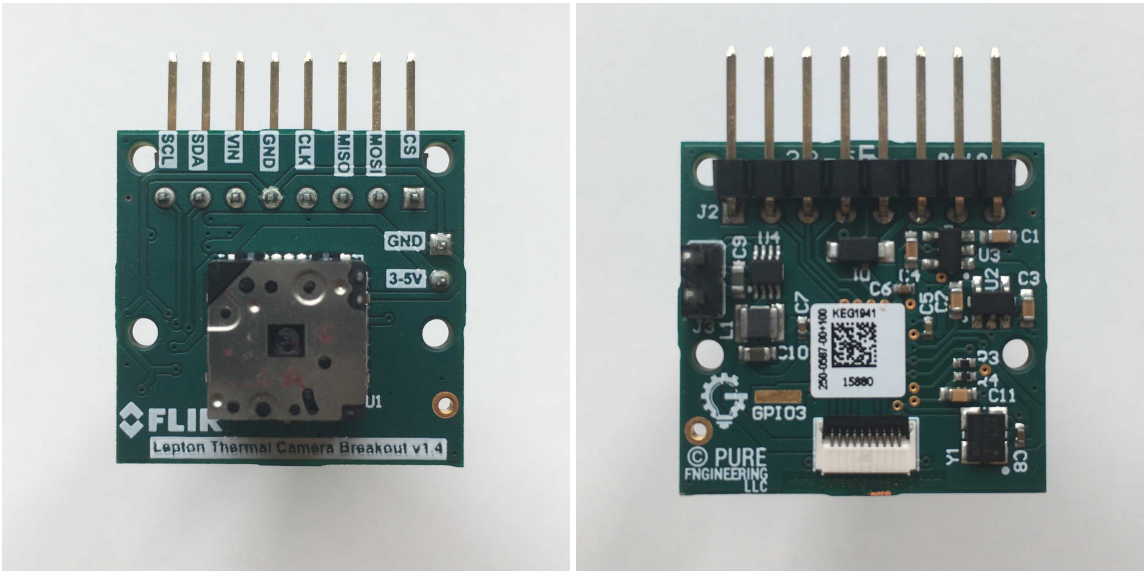



Figure 4.6: Front and back side of Flir Lepton breakout board with thermal camera module inserted.

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 4.7. 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.

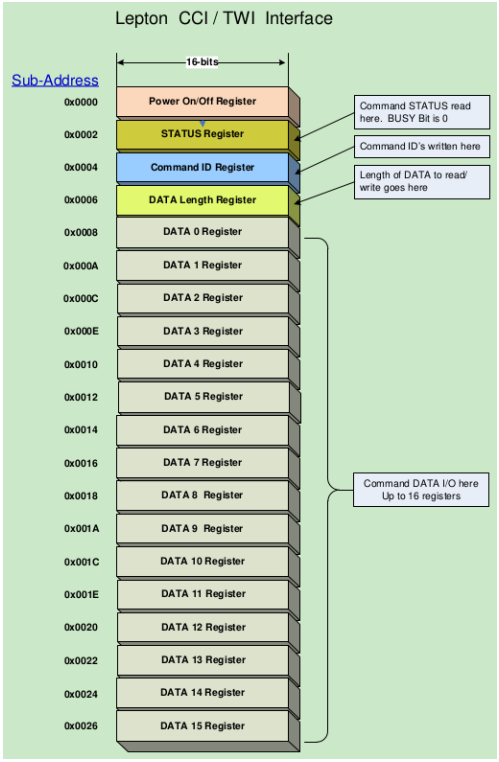



Figure 4.7: Command and control interface of FLIR Lepton camera.

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.

### 4.1.4 PIR Sensor

## 4.2 Firmware

### 4.2.1 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.

#### 4.2.1.1 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 [37]

#### 4.2.1.2 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 functionalities

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

### 4.2.2 Architecture design

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

As seen on Figure 4.8 we split the firmware into two hardware and application modules.

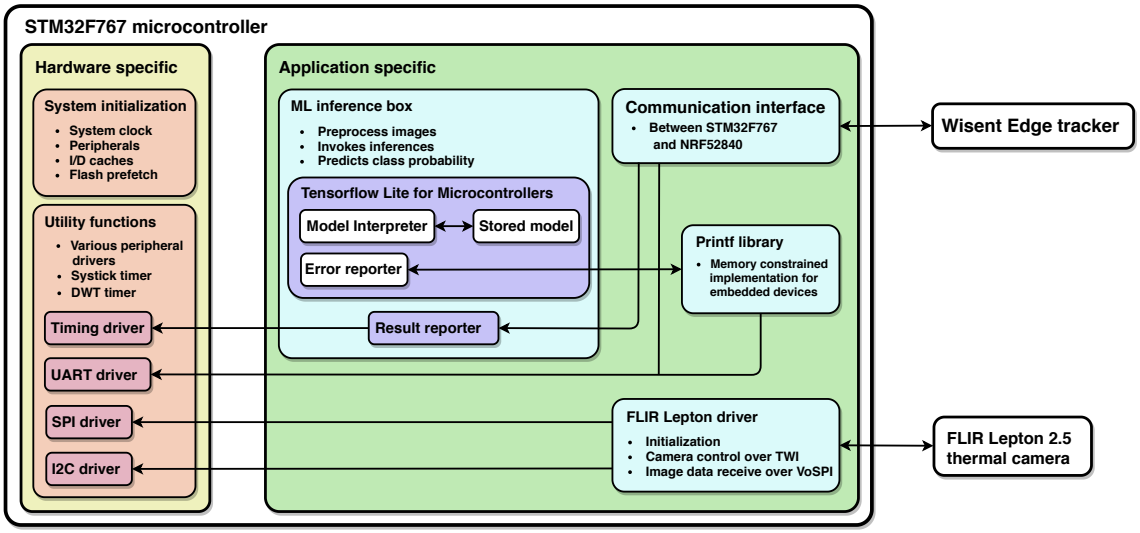



Figure 4.8: Architecture diagram of the firmware that is running on a STM32F767ZI microcontroller.

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 4.9. 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.

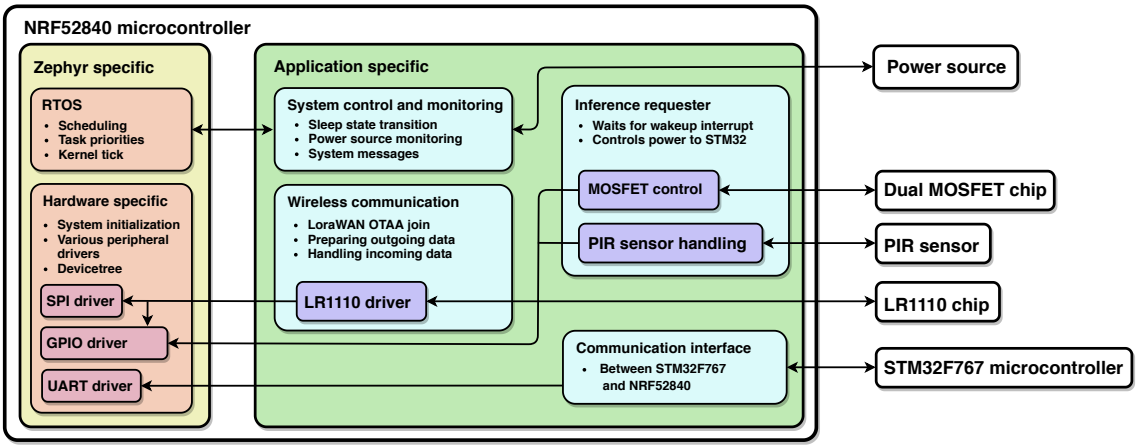



Figure 4.9: Architecture diagram of the firmware that is running on a NRF52840 microcontroller.

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.

### 4.2.3 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 4.10

[4]Project is open-source and publicly available on GitHub [38].

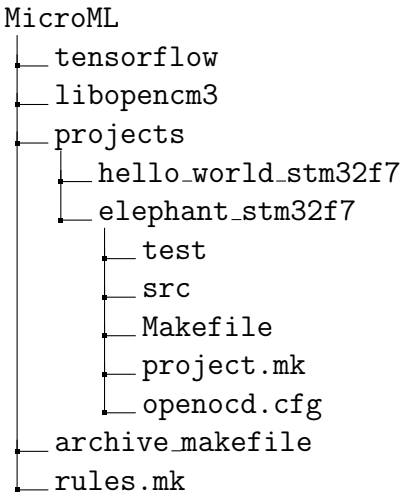



Figure 4.10: Directory structure of MicroML project.

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 4.11 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

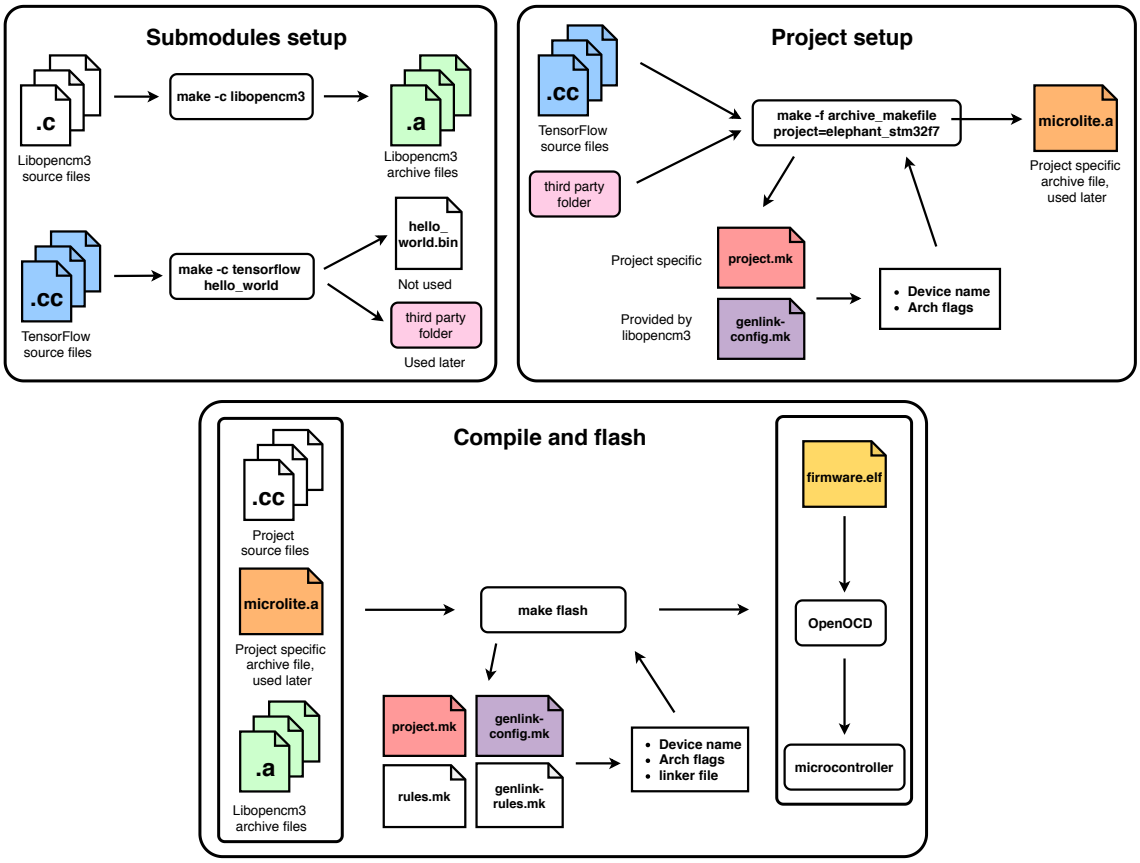



Figure 4.11: Build system of MicroML project.

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.

### 4.2.4 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 4.12 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 [39].

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.

### 4.2.5 Wisent board control firmware

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

## 4.3 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 4.13.

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 4.14 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

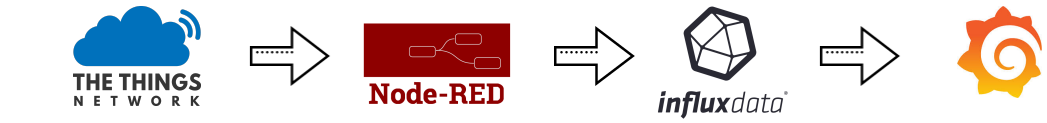
graphs, gauges, heat maps, alert lists and others. In our use case we could display information about various devices in the field, such as battery voltage, number of wakeup triggers, results of each inference and others.

Example of Grafana graph can be seen on Figure 4.15.

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 4.12: 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 4.13: Server side flow of information. Icons source: [11]


Figure 4.14: Node-RED flow.

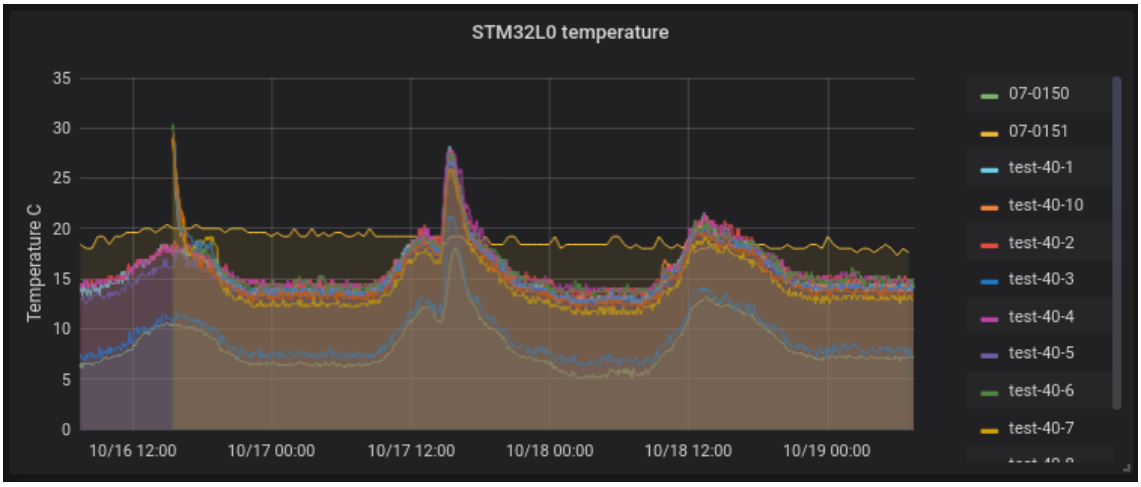



Figure 4.15: Example of Grafana graph.

# Bibliography


[1] Nyhus, P. J. Human–Wildlife Conflict and Coexistence. *Annual Review of Environment and Resources*, 41, (2016), 11, pages 143–171.

[2] SARPO, WWF. Human Wildlife Conflict Manual. Available on: `https://wwf.panda.org/our_work/wildlife/human_wildlife_conflict/hwc_news/?84540/Human-Wildlife-Conflict-Manual`, [12.06.2020].

[3] WILDLABS, WWF. Human Wildlife Conflict Tech Challenge: Polar Bear Case. Available on: `https://www.wildlabs.net/hwc-tech-challenge-polar-bear-case`, [14.06.2020].

[4] WILDLABS, WWF. Human Wildlife Conflict Tech Challenge: Tiger Case. Available on: `https://www.wildlabs.net/hwc-tech-challenge-tiger-case`, [14.06.2020].

[5] WILDLABS, WWF. Human Wildlife Conflict Tech Challenge: Asian Elephant Case. Available on: `https://www.wildlabs.net/hwc-tech-challenge-elephant-case`, [14.06.2020].

[6] Ganesh, S. Human-elephant conflict kills 1,713 people, 373 pachyderms in 3 years. Available on: `https://www.thehindu.com/news/national/human-elephant-conflict-kills-1713-people-373-pachyderms-in-3-years/article26225515.ece`, [14.06.2020].

[7] Guha N., In the Heart of the Conflict: Understanding the Human Elephant Dynamics in Udalguri. Available on: https://www.econe.in/post/in-the-heart-of-the-conflict-understanding-the-human-elephant-dynamics-in-ud [08.09.2020].

[8] Save our species, Human wildlife conflict - global challenge: local solutions. Available on: https://www.saveourspecies.org/news/human-wildlife-conflict-global-challenge-local-solutions, [08.09.2020].

[9] The Week, Arresting image of human elephant conflict wins photo prize. Available on: https://www.theweek.co.uk/89566/arresting-image-of-human-elephant-conflict-wins-photo-prize, [08.09.2020].

[10] WILDLABS, WWF. Human Wildlife Conflict Tech Challenge. Available on: https://www.wildlabs.net/hwc-tech-challenge, [14.06.2020].

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

[12] WILDLABS, WWF. HWC Tech Challenge Winners Announced. Available on: https://www.wildlabs.net/resources/news/hwc-tech-challenge-winners-announced, [20.06.2020].

[13] Dangerfield A. Progress report – January 2019 – Thermal imaging for human-wildlife conflict. Available on: https://blog.arribada.org/2019/01/10/progress-report-january-2019-thermal-imaging-for-human-wildlife-conflict, [20.06.2020].

[14] 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.

[15] Burkov, A. *The Hundred-Page Machine Learning Book.* Andriy Burkov, 2019.

[16] Li F., Karpathy A., "Cs231n: Convolutional neural net- works for visual recognition." Stanford University course. Available on: http://cs231n.stanford.edu/, [25.06.2020].

[17] Zhang, Y., Suda, N., Lai, L., and Chandra, V. Hello edge: Keyword spotting on microcontrollers. *ArXiv*, abs/1711.07128, (2017), 2.

[18] Louis, M. S., Azad, Z., Delshadtehrani, L., Gupta, S., Warden, P., Reddi, V. J., and Joshi, A. Towards deep learning using tensorflow lite on risc-v. *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, 1, (2019), 6.

[19] Warden P., Why the future of machine learning is tiny. Available on: https://petewarden.com/2018/06/11/why-the-future-of-machine-learning-is-tiny/, [06.07.2020].

[20] Situnayake D., Make deep learning models run fast on embedded hardware. Available on: https://www.edgeimpulse.com/blog/make-deep-learning-models-run-fast-on-embedded-hardware/, [08.07.2020].

[21] Dive into deep learning, Convolutional Neural Networks. Available on: http://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html, [17.9.2020].

[22] TensorFlow, GitHub repository. Available on: https://github.com/tensorflow/tensorflow, [21.9.2020].

[23] Rouse M., internet of things (IoT). Available on: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT, [27.9.2020].

[24] Ubidots, LoRaWAN vs NB-IoT: A Comparison Between IoT Trend-Setters. Available on: https://ubidots.com/blog/lorawan-vs-nb-iot/, [28.9.2020].

[25] Polymorph, IoT connectivity comparison (GSM vs LoRa vs Sigfox vs NB-Iot). Available on: https://www.polymorph.co.za/

iot-connectivity-comparison-gsm-vs-lora-vs-sigfox-vs-nb-iot/, [28.9.2020].

[26] Bäumker, E., Garcia, A., and Woias, P. Minimizing power consumption of lora ® and lorawan for low-power wireless sensor nodes. *Journal of Physics: Conference Series*, 1407, (2019), 11, page 012092.

[27] Knight M., A GitHub repository containing open-source implementation of the LoRa CSS PHY. Available on: https://github.com/BastilleResearch/gr-lora, [29.9.2020].

[28] Wong G.W., LoRa Rolls Into Philly. Available on: https://www.electronicdesign.com/technologies/embedded-revolution/article/21805205/lora-rolls-into-philly, [29.9.2020].

[29] Vollmer, M. and Möllmann, K. P. *Infrared Thermal Imaging: Fundamentals, Research and Applications*. Wiley-VCH, Boston, Massachusetts, 2018.

[30] Dangerfield A., HWC Tech Challenge Update: Comparing thermopile and microbolometer thermal sensors. Available on: https://www.wildlabs.net/resources/case-studies/hwc-tech-challenge-update-comparing-thermopile-and-microbolometer-thermal, [18.07.2020].

[31] Bhan, R., Saxena, R., Jalwania, C., and Lomash, S. Uncooled infrared microbolometer arrays and their characterisation techniques. *Defence Science Journal*, 59, (2009), 11, page 580.

[32] MoviTherm, What is NETD in a Thermal Camera? Available on: https://movitherm.com/knowledgebase/netd-thermal-camera/, [18.07.2020].

[33] Dangerfield A., Progress report – February 2020 – Thermal imaging for human-wildlife conflict. Available on: https://blog.arribada.org/2020/02/17/

progress-report-feburart-2020-thermal-imaging-for-human-wildlife-conflict/, [02.10.2020].

[34] GroupGets - LeptonModule, GitHub repository. Available on: https://github.com/groupgets/LeptonModule, [21.9.2020].

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

[36] 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].

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

[38] 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].

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