UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Marko Sagadin

Energy efficient system for detection of elephants with Machine Learning

Energetsko učinkovit sistem za detekcijo slonov s pomočjo strojnega učenja

Master's thesis

Maribor, December 2020

Marko Sagadin

Energy efficient system for detection of elephants with Machine Learning

Energetsko učinkovit sistem za detekcijo slonov s pomočjo strojnega učenja

Master's thesis

Maribor, December 2020

Energy efficient system for detection of elephants with Machine Learning

Energetsko učinkovit sistem za detekcijo slonov s pomočjo strojnega učenja

Master's thesis

Student:	Marko Sagadin
Study Programme:	Second Cycle Bologna Study Programme of Electrical Engineering
Module of study program:	Electronics
Mentor:	Dr. Iztok Kramberger
Co-mentor:	MECE. Vojislav Dragan Milivojević
Lector:	Mrs. Shelagh Hedges

Zahvala

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

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

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

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

Acknowledgement

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

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

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

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

Energetsko učinkovit sistem za detekcijo slonov s pomočjo strojnega učenja

Ključne besede: strojno učenje, mikrokrmilnik, inferenca na napravi, termalna kamera, sistem z majhno porabo

UKD: 004.85:004.932(043.2).

Povzetek

1. Uvod

Konflikti med ljudmi in sloni predstavljajo velik problem ohranjanja populacije slonov. Zaradi fragmentacije in pomanjkanja habitata sloni, v iskanju hrane, pogosto zaidejo na riževa polja in plantaže, kjer pridejo v stik s človekom. Po podatkih skupnosti WILDLABS, zaradi konfliktov, samo v Indiji, letno umre povprečno 400 ljudi in 100 slonov. Sistemi zgodnje opozoritve nadomeščajo vlogo človeških stražarjev in opozarjajo bližnjo skupnost o bližini, potencialno nevarnih, slonov in tako pripomorejo k zmanjševanju konfliktov med ljudmi in sloni.

V tem magistrskem delu predstavljamo strukturo sistema zgodnje opozoritve, ki je sestavljen iz večih, nizko porabnih, vgrajenih sistemov, ki so opremljeni s termalnimi kamerami in ene dostopne točke oz. prehoda (gateway). Vgrajeni sistemi so postavljeni na terenu, ob zaznavi slona pošljejo opozorilo preko brezžičnega omrežja do dostopne točke, ki nato lahko opozori lokalno skupnost. Za prepoznavo slonov iz zajetih termalnih slik smo uporabili metode strojnega učenja, bolj specifično konvolucijske nevronske mreže. Glavni cilji tega magistrskega dela so bili zasnova, izvedba in ovrednotenje modelov strojnega učenja, ki jih je možno poganjati na mikrokrmilnkih pod pogoji nizke porabe.

2. Teoretični opis gradnikov sistema

V tem poglavju opisujemo osnovna znanja, ki jih bralec potrebuje za razumevanje tega magistrskega dela. Opišemo kako lahko strojno učenje pomaga reševati probleme, ki bi s klasičnimi tehnikami zahtevali kompleksne rešitve. Podrobno predstavimo izvajanje modelov strojnega učenja v vgrajenih sistemih. Ugotovimo, da kljub omejitvam kot so nizke procesorske moči in majhni spomini, prednosti kot so hitra odzivnost na dogodke, zmanjšanje porabe zaradi manšje potrebe po pošiljanju podatkov v oblak in povečane stopnje zasebnosti, hitro odtehtajo slabosti. Lotimo se opisa nevronskih mrež, aktivacijskih funkcij, konvolucijskih nevronskih mrež in tehnik prenosnega učenja. Predstavimo tudi platformo TensorFlow Lite for microcontrollers, ki nam je omogočila implementacijo nevronskih mrež na mikrokrmilnikih. Naredimo pregled možnih brezžičnih tehnologij in argumentiramo zakaj smo se odločili za LoRaWAN. Nazadnje opišemo delovanje termalnih kamer in predstavimo kako je potekala izbira optimalne termalne kamere. Izbrana kamera je bila FLIR Lepton.

3. Zasnova modela nevronske mreže

V tem poglavju podrobno opišemo celoten proces zasnove modela, ki je sposoben klasificirati termalne slike in predvideti kateri objekt je na sliki. Pri zasnovi smo se omejili na prepoznavo 4 različnih razredov: sloni, ljudje, krave in narava oz. nakjučni objekt. Zadali smo si cilj, da klasificiranje termalne slike ne sme trajati več kot 1 sekundo in da mora biti model dovolj majhen, da ga lahko naložimo na mikrokrmilnik. Na začetku opišemo orodja in delovno okolje, ki smo jih uporabljali pri zasnovi modela (uporabljali smo platformo TensorFlow), nato pa se lotimo analize nabora termalnih slik, ki jih je zbralo podjetje Arribada Initiative. Iz nabora termalnih slik smo izbrali slike, ki so ustrezale našim zahtevam. Nabor termalnih slik je vseboval veliko število slik slonov in ljudi, ampak ne veliko slik krav ali narave. Slednje smo posneli sami na terenu, s hitrim prototipom, ki smo ga izdelali sami.

Opisali smo kako smo so slike pripravili za učenje modela in predstavili smo osnovno arhitekturo modela. Odločili smo se za klasično convolucijsko arhitekturo, kjer se konvolucijski sloji in pooling sloji ponovijo nekajkrat nato pa se priključijo na tesno

povezani neuronski sloj. Opisali smo tudi, kako poteka optimizacija modelov, ki bodo tekli na mikrokrmilnikih.

Nazadnje ponovno opišemo potek zasnove modela od začetka do konca, ampak tokrat to storimo s Edge Impulse Studijem.

4. Zasnova in izvedba sistema zgodnje opozoritve

V četrtem poglavju predstavimo sprva generalne gradnike sistema in njihove funkcije, nato pa predstavimo konkretne komponente. Odločili smo se za sistem z dvema mikrokrmilnikoma. Mikrokrmilnik NRF52840 kontrolira delovanje celotnega sistema in preživi večino časa v režimu nizke porabe. Ob signalu iz PIR sensorja se zbudi iz spanja in vklopi drugi mikrokrmilnik, STM32F767ZI. STM32F767ZI je visoko zmogljiv mikrokrmilnik s Cortex-M7 jedrom. Povezan je s FLIR Lepton termalno kamero in kontrolira zajemanje slik ter njiovo procesiranje s nevronsko mrežo. STM32F767ZI sporoči rezultate klasifikacije nRF52840 mikrokrminlniku, ki jih obdela in nato pošlje preko LoRaWAN omrežja na dostopno točko. Za LoRa brezžični modul smo uporabili LR1110 čip.

Veliki del magistrskega dela se je ukvarjal s prenosom TensorFlow Lite for Microcontrollers platfrome na platformo naše izbire, libopencm3. V procesu prenosa smo se podrobno spoznali s prevajanjem in povezovanjem kode, saj nismo uporabljali programskega okolja, ki bi to naredilo za nas. Tako smo ustvarili odprto-kodni projekt MicroMl, ki omogoča uporabo TensorFlow lite kode na platformi libopencm3. Sestava in uporabo MicroML-a smo podrobno opisali. MicroMl smo uporabili pri pisanju kode za mikrokrmilnik STM32F767ZI, za nRF52840 pa smo uporabili operacijski sistem Zephyr.

5. Meritve in rezultati

Izvedli smo vrsto različnih meritev in testov. V 3. poglavju smo predstavili osnovno arhitekturo modela, ampak nismo določili točnih vrednosti hiperparametrov. Ker je iskanje optimalnih hiperparametrov nehevristična naloga, smo določili možni razpon hiperparametrov in izvedli algoritem naključnega iskanja, ki je naučil večje število

modelov z različnimi hiperparameteri. V prvem koraku smo naučili 300 modelov, največja dosežena natančnost modela je bila 98.35 %. Opazili smo da nekaj je nekaj modelov doseglo primerljive rezultate, s mnogo manjšim številom parametrov. Ker se manjše število parametorv prevede v krajši čas inference, smo ponovili iskanje, tokrat z zmanjšanjim možnim razponom hiperparametrov. Iz rezultatov smo izbrali nekaj obetajočih modelov in jih primerjali s modeli, ki smo jih ustvarili s Edge Impulse Studijem.

Vse izbrane modele smo tudi stestirali na mikrokrmilniku, beležili smo čas inference in velikost kode v flash in ram pomnilniku. Večina modelov je izvedla inferenco pod 200 milisekundami. Skoraj vsi modeli so zasedli manj kot 1 MB flash pomnilnika. Vsi Edge Impulse modeli so zasedli manj ram pomnilnika v primerjavi z našimi modeli, saj so uporabljali drugačen pristop izvajanja inference.

Za najbolj uspešne modele so se izkazali modeli, ki so bili trenirani s tehniko prenosnega učenja. Dosegali so visoke natančnosti in se kljub večjemu številu parametrov izvajali hitreje kot klasični konvolucijski modeli s manjšimi števili parametrov.

Želeli smo predvideti življensko dobo sistema z baterijskim napajanjem, zato smo izvedli več testov porabe. Skupna poraba nRF52840 mikrokrmilnika in LR1110 modula je bila večja kot pričakovana, okoli 207 µA*, pričakovali smo porabo pod 10* µA*. Povprečna poraba celotne sekvence detekcije, kjer je mikrokrmilnik naredil 5 slik in na vsaki izvajal inferenco, ter nato poslal sporočilo preko LoRaWAN omrečja, je bila 52* mA *in je trajala 8 sekund. Za izračun smo si izbrali litijsko baterijo s nominalno kapaciteto 3350* mA h*. V izračunu smo predvideli število detekcij na dan, omejili smo se na 100, 200, 300, 400, 500 in 600 detekcij. Hkrati smo predpostavili, da lahko ima naš sistem do 6 baterij. Izračunana življenska doba sistema z šestimi baterijami in 600 detekcijami na da je bil 10 mesecov, v primeru 100 detekcij je bila 44 mesecov.*

6. Povzetek

V tem magistrskem delu smo predstavili rešitev, sistem zgodnje opozoritve, ki bi lahko pripomogla k zmanjševanju konfliktov med ljudmi in sloni. Prikazali smo celotni postopek od analize problema, predloga rešitve, analize nabora termalnih slik,

procesiranja slik in zasnova modela. Opisali smo implementacijo izvedbe inference, kako smo izvedli prenos TensorFlow-a na našo platformo in celotno programsko kodo vgrajenega sistema. Podrobno smo primerjali naučene modele in jih stestirali na mikrokrmilniku. Izračunali smo predvideno življensko dobo sistema iz zajetih meritev porabe.

Magistrska naloga zajema več različnih področij, vsako je možno izboljšati. Da bi izboljšali natančnost modelov, bi bilo potrebno zbrati več termalnih slik in podrobneje raziskati tehniko prenosnega učenja, saj je ta pokazala zelo dobre rezultate.

Iz vidika porabe sistema, bi bilo potrebno nadalje zmanjšati porabo celotne detekcije in porabo v režimu nizke porabe. To je izvedljivo s izdelavo primernega tiskanega vezja, kakor tudi proučevanjem nepotrebne programske kode.

Seveda je potrebno celotni sistem stestirati na terenu, tako bi pridobili nove ideje in zahteve za izboljšave.

Energy efficient system for detection of elephants with Machine Learning

Key words: machine learning, microcontroller, on-device inference, thermal camera, low-power system

UKD: 004.85:004.932(043.2).

Abstract

Human-Elephant Conflicts are a major problem in terms of elephant conservation. According to WILDLABS, an average of 400 people and 100 elephants are killed every year in India alone because of them. Early warning systems replace the role of human watchers and warn local communities of nearby, potentially life threatening, elephants, thus minimising the Human-Elephant Conflicts.

In this Master's thesis we present the structure of an early warning system, which consists of several low-power embedded systems equipped with thermal cameras and a single gateway. To detect elephants from captured thermal images we used Machine Learning methods, specifically Convolutional Neural Networks. The main focus of this thesis was the design, implementation and evaluation of Machine Learning models running on microcontrollers under low-power conditions. We designed and trained several accurate image classification models, optimised them for on-device deployment and compared them against models trained with commercial software in terms of accuracy, inference speed and size. While writing firmware, we ported a part of the TensorFlow library and created our own build system, suitable for the libopencm3 platform. We also implemented reporting of inference results over the LoRaWAN network and described a possible server-size solution. We finally a constructed fully functional embedded system from various development and evaluation boards, and evaluated its performance in terms of power consumption. We show that embedded systems with Machine Learning capabilities are a viable solution to many real life problems.

List of Abbreviations

WWF	World Wide Fund for Nature
HEC	Human-Elephant Conflicts
ML	Machine Learning
NN	Neural Networks
CNN	Convolutional Neural Networks
DNN	Deep Neural Networks
IoT	Internet of Things
RMS	Root Mean Square
ReLu	Rectified Linear Activation Unit
ISM	Industrial, Scientific and Medical
3GPP	The 3rd Generation Partnership Project
LoRa	Long Range
IR	Infrared
EM	Electromagnetic
LWIR	Long Wave Infrared Region
ROIC	Readout Integrated Circuit
VOx	Vanadium-Oxide
NETD	Noise Equivalent Temperature Difference
CPU	Central Processing Unit
FPA	Focal Point Array
TWI	Two Wire Interface

LFRC	Low Frequency Resistance-Capacitance Oscillator
GPIO	General Purpose Input/Output
UART	Universal Asynchronous Receiver/Transmitter
SPI	Serial Peripheral Interface Bus
PIR	Passive Infrared Sensor
I2C	Inter-Integrated Circuit
MOSI	Master Out Slave Input
MISO	Master Input Slave Out
USB	Universal Serial Bus
FPA	Focal Point Array
GCC	the Gnu Compiler Collection
TTN	The Things Network
DK	Development Kit
EVK	Evaluation Kit
GNSS	Global Navigation Satellite System
DWT	Data Watchpoint Trigger
TTN	The Things Network
SQL	Structured Query Language
CSV	Comma Separated Value

Table of Contents

List of Figures

List of Tables

List of Listings

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 plantation 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 warning system

Key component of minimizing Human-Elephant Conflicts is a reliable early warning system. A system capable of detecting the presence of nearby elephants would warn nearby communities and give them enough time to prepare and respond non-violently. 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 communicate wirelessly with a server. Server would display messages and alerts, and if needed, directly notify local community, if elephants were detected in local area.

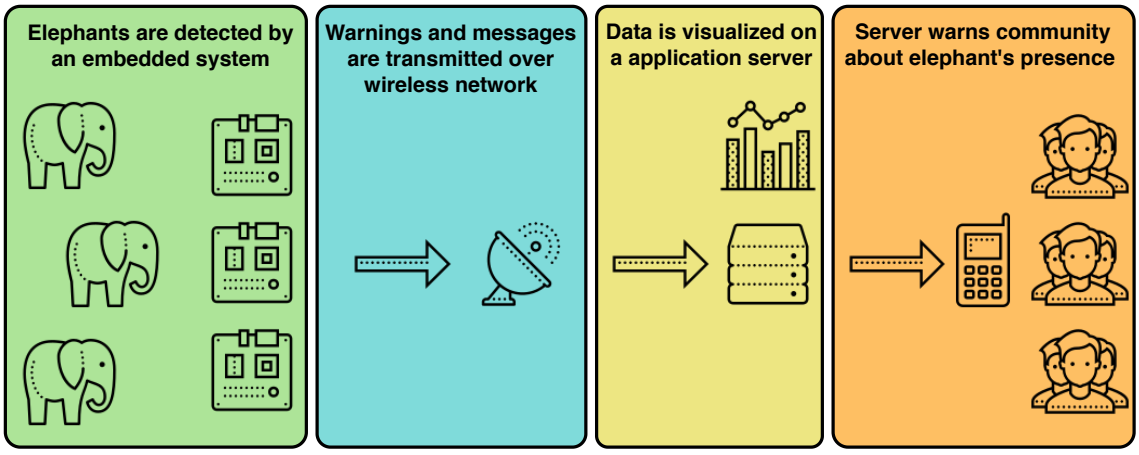


Figure 1.2: Diagram of an early warning system. Icons source: [11]

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 warning 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 used for training the Neural Networks and 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 to 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 deployed 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 difficult 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 basic 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 the available 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.

Over the past several years there has been a growing desire to expand ML applications to embedded devices. Executing ML algorithms directly on microcontrollers has it's benefits and presents new challenges compared to executing those algorithms on PC's or 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 is 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 from 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 to recognize animals, especially elephants, from thermal images, with Machine Learning algorithms, running directly on a microcontroller.

Additionally we will 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.

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. Performance of the model is

measured in **accuracy**. It tells us what percentage of input data has been predicted 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 and Expectation Maximization, 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 a 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.

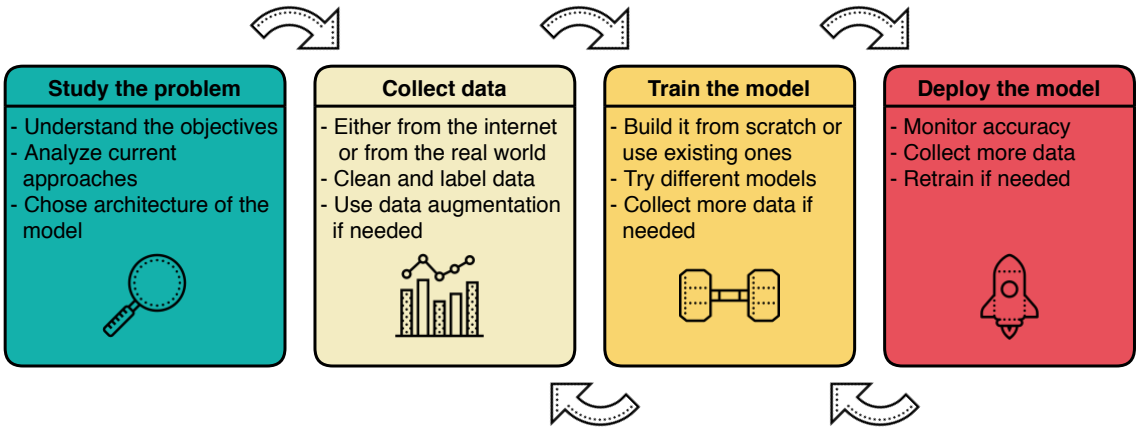


Figure 2.1: Workflow diagram of solving a generic Machine Learning problem. Icons source: [11]

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 TinyML

Machine Learning on embedded devices, also known as TinyML is an emerging field, which coincides nicely with the Internet of Things. The field is experiencing a large boom in popularity, when we started working on this thesis in May, 2020, there was considerable fewer resources about it, then they are now, in December, 2020. Nevertheless the resources about it are limited, 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] [20] [21].

Running machine 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 2.4.1.

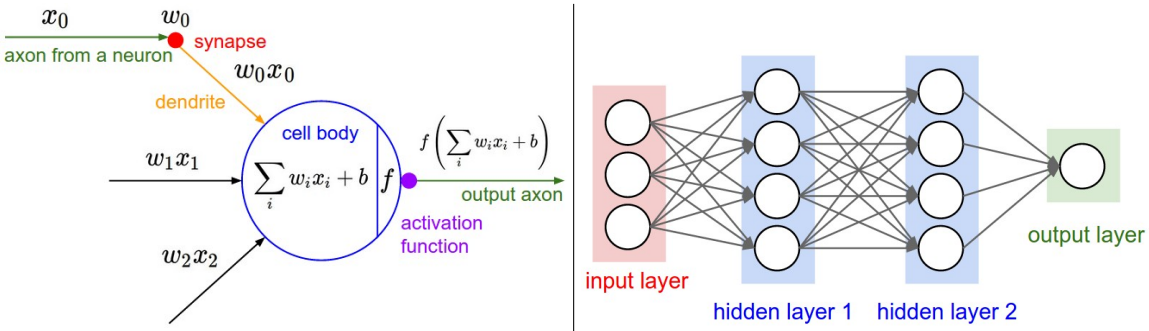
2.2 Neural Networks

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, several decades of research and technological progress had to pass 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 Neuron takes several inputs, multiplies each input with its **weight** and sums them up. A **bias** parameter is added to the sum, which is then passed to 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) 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 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.

Another commonly used activation function is a softmax function (seen in 2.1, which takes a vector as an input, computes the exponential of every element and divides

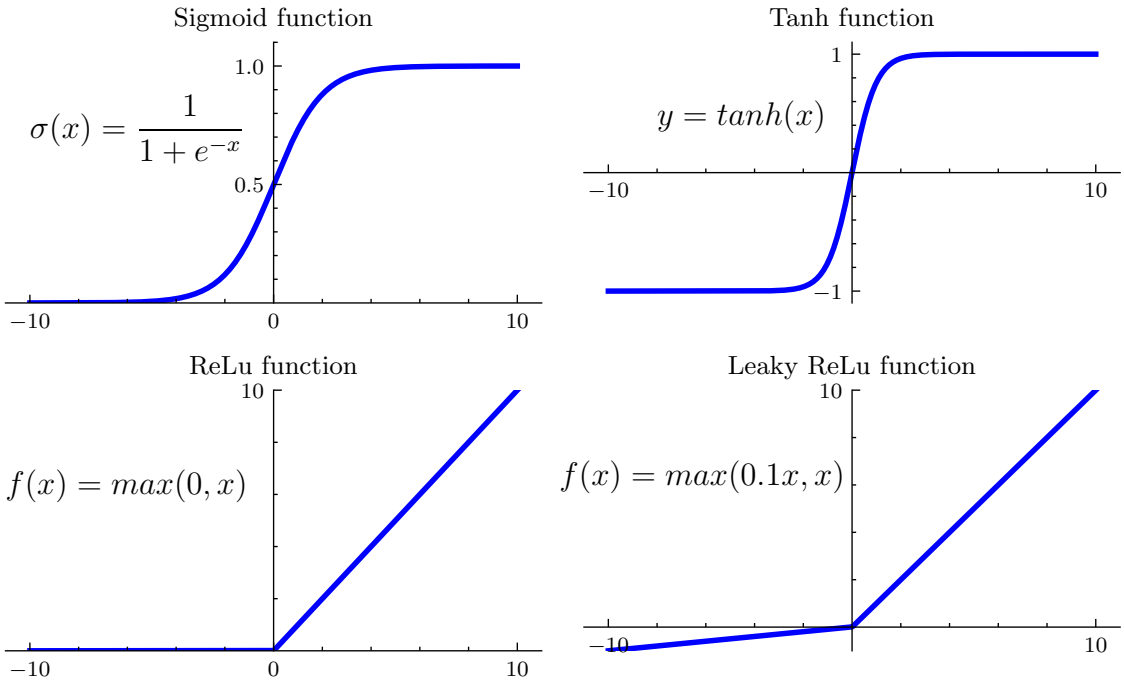


Figure 2.3: Different activation functions and their equations.

that with the sum of exponentials of all elements [14] The input vector of values becomes a vector of probabilities. Softmax is usually used as an activation in the last layer of a classifier network.

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

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.

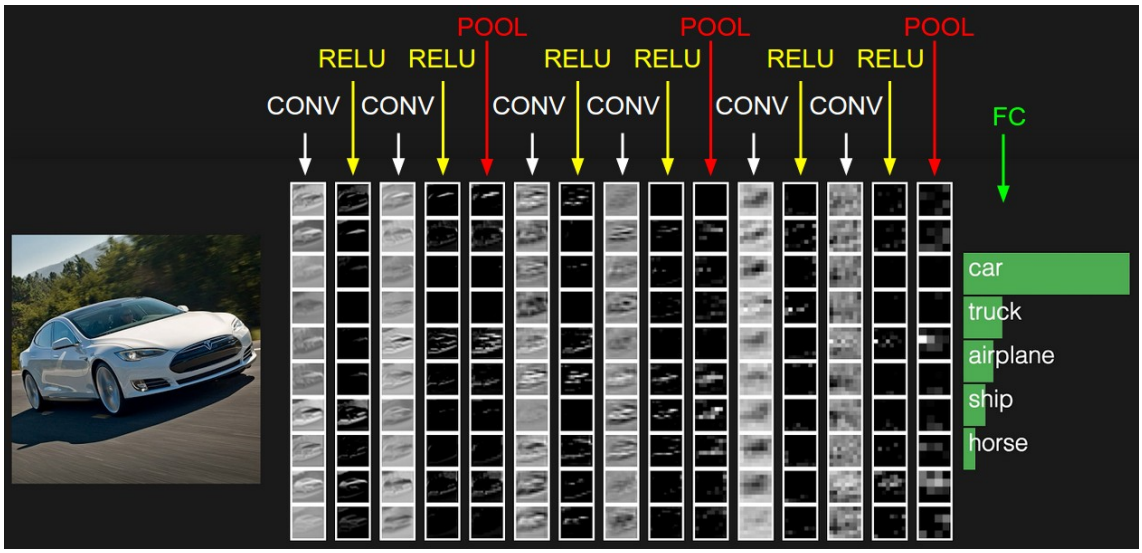


Figure 2.4: Structure of a Convolutional Neural Network. Image source: [16]

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 which produces probability results in given classes.

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

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.

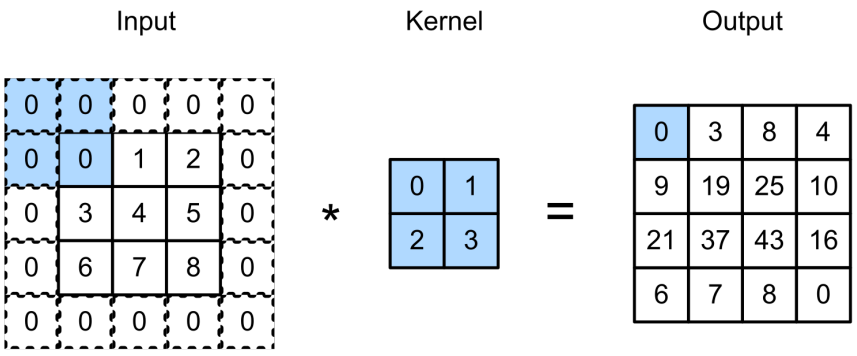


Figure 2.5: Dot product operation between filter and zero-padded input matrix. Image sources: [22]

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

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

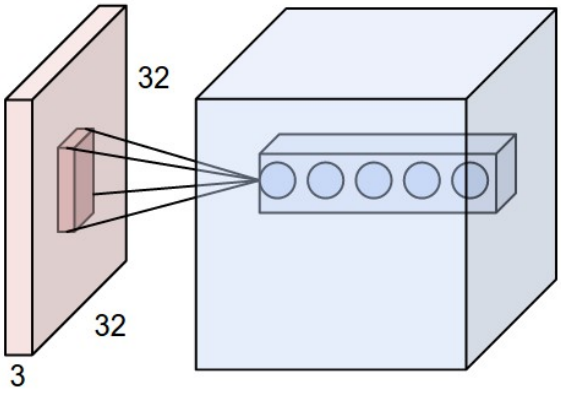
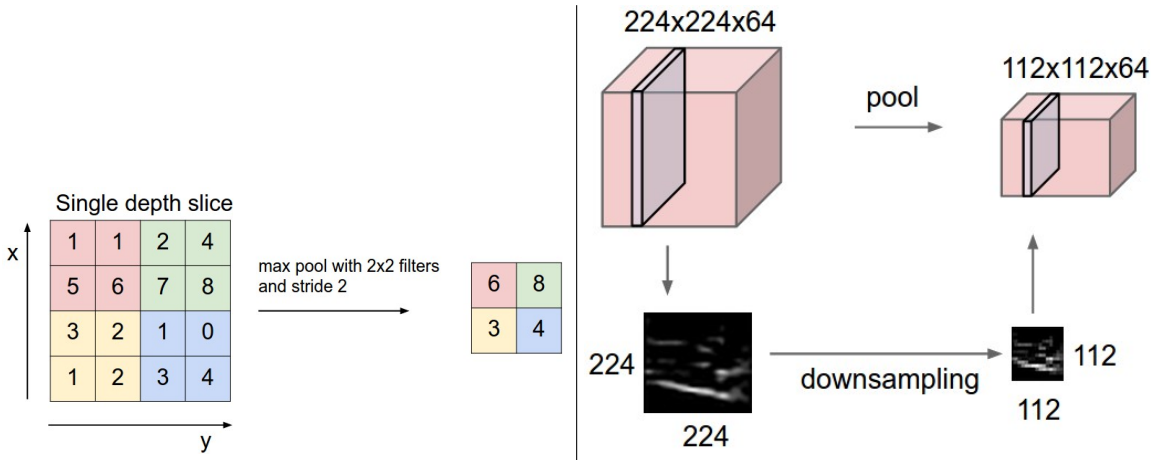


Figure 2.6: Convolutional layer with five different filters. Image sources: [22] [16]

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.



(a) Max pooling operation

(b) Effect of polling on input dimensions

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

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.

2.3 Transfer Learning

Transfer Learning is a ML technique, which transfers knowledge that was learned on one problem, to another, similar problem.

We mentioned earlier in Section 2.2.3 that CNNs learn in first few layers how to detect simple shapes, like edges and lines, which are in later layers used to detect boxes, circles, faces and so on. This behaviour is not limited only to CNNs, but to all DNNs.

The task of detecting simple shapes in first few layers of DNNs can be common between models that are trying to solve a similar problem. Example would be a model classifying images depicting different types of objects and a model classifying images with different types of animals. First model is trying to solve a more general problem than the seconds one, however, the learned weights in first few layers from first model can be reused in the second model.

And this is exactly what Transfer Learning is all about, it is possible to train a big, complex model on a large dataset and then reuse first few layers in a different model. Example Transfer learning is shown in Figure 2.8. It is important to note that the weights of transferred layers are fixed, they are not being trained.

Benefits of using Transfer learning are:

- **Shorter training time.** Using a pre-trained model with fixed weights and adding few trainable layers on its top greatly lower training time, simply because there are less parameters to train.
- **No need for large datasets.** Training DNN usually requires large amount of data, which we might not have. Using a pre-trained model, that was trained on a large dataset, and applying it to a specific small dataset yielded better performance than just training a model from scratch [14].

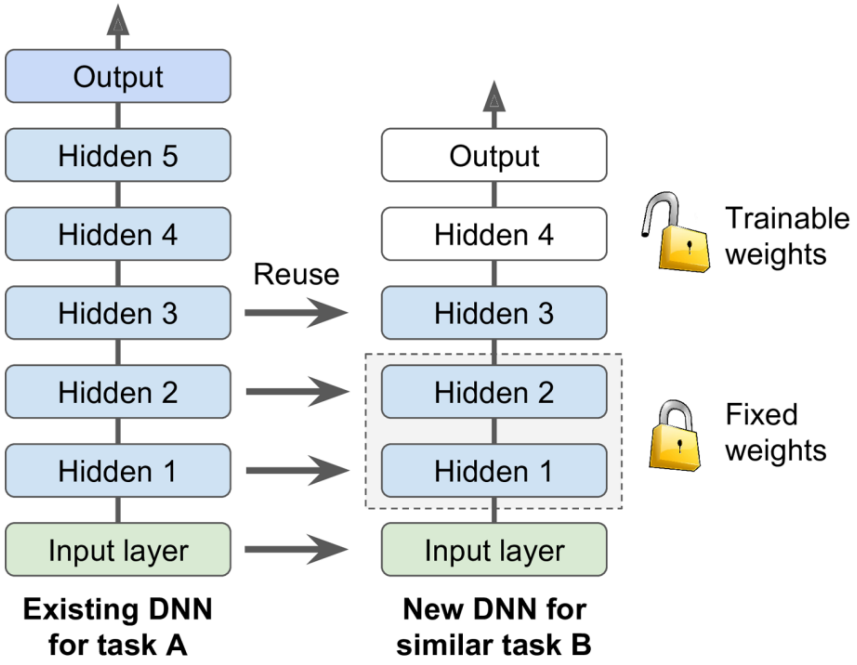


Figure 2.8: Transfer Learning. Image source: [14]

2.4 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 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.

2.4.1 TensorFlow Lite for Microcontrollers

TensorFlow Lite (TFLite) is a set of tools and libraries that enable running ML inferences on constrained devices [23]. 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 the `printf()` function. As microcontroller binaries are usually quite big, flashing firmware to a microcontroller is a time consuming process. It makes sense to first 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 4.2.5.

The TFLite Micro library is available publicly as a part of a much larger TensorFlow project on GitHub [23]. 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 2.9 shows all steps that are needed to prepare an ML model for running on a microcontroller.

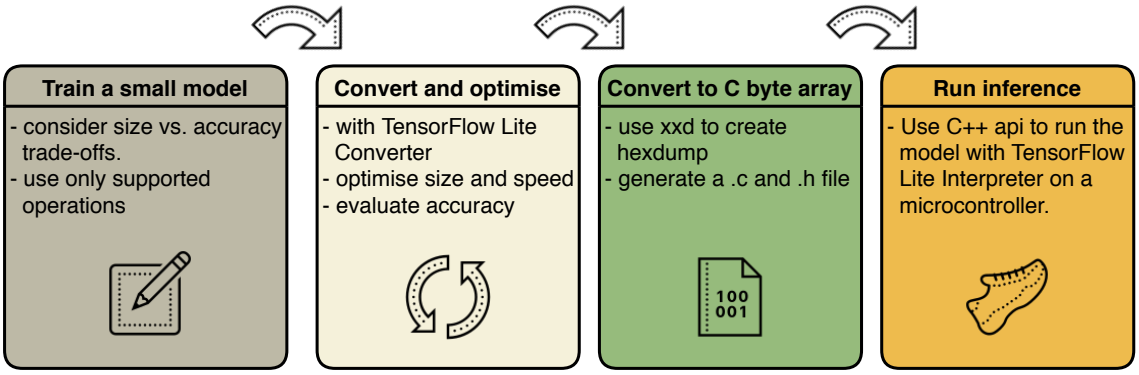


Figure 2.9: Workflow of preparing a ML model for an inference on a microcontroller. Icons source: [11]

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.

2.4.1.1 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.

2.5 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 [24]. 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 warning 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 [25]. 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 [25]. Compared to Sigfox and LoRa it has better latency and a higher data rate, but also higher power consumption [26]. 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].

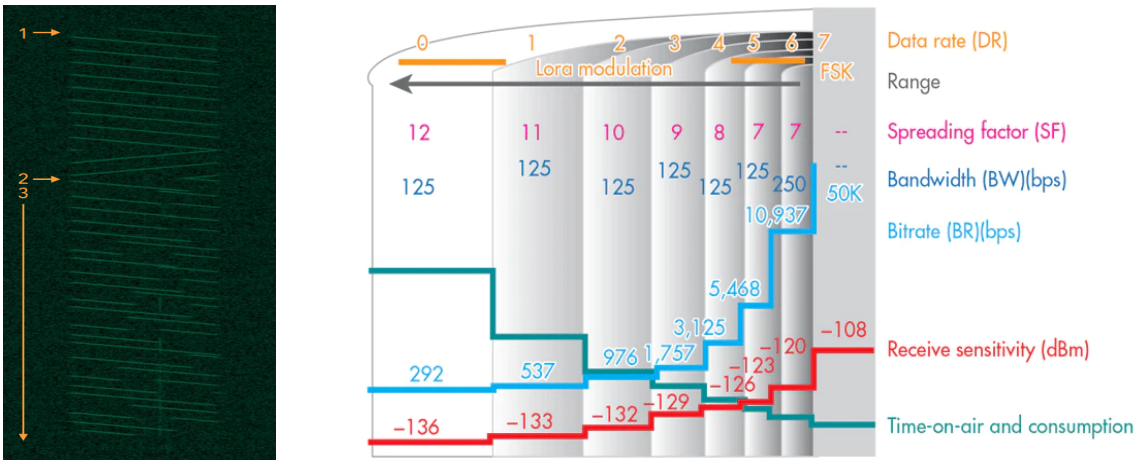
Sigfox is a radio technology developed by the company of the same name that operates on an unlicensed Industrial, Scientific and Medical (ISM) radio band. In many views it is similar to LoRa, as it has the comparative range and power consumption [26]. However, there are a few important differences. Although 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 [26]. 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.

2.5.1 LoRa and LoRaWAN

LoRa (Long Range) is a physical layer protocol that defines how information is modulated and transmitted over the air [27] [25]. 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 [27]. As the protocol is proprietary, exact details of it are not known, although it was reverse engineered by a radio frequency specialist [28]. An example of a LoRa signal that was captured with a software defined radio can be

seen in Figure 2.10a. 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 2.10: 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: [28] [29]

This kind of modulation gives LoRa extreme resiliency against the interference of other radio frequency signals that might be using the same frequency band [27] [29]. For example, on a lower part of Figure 2.10a 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 2.10b, 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 [27]. 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 [27]. 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.

2.6 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 2.11.

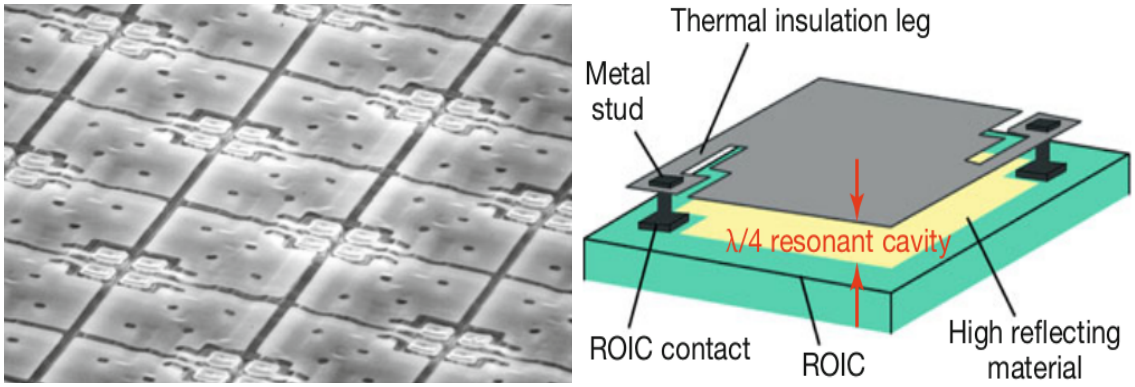
Figure 2.11: Comparison between a normal image and thermal image. Image source: Arribada Initiative [30]

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) [31]. We can classify IR cameras broadly into two categories: photon detectors or thermal detectors [31]. Photon detectors convert absorbed EM radiation directly into electric signals by the change of concentration of free charge carriers [31]. Thermal detectors covert absorbed EM radiation into thermal energy, raising the detector temperature [31]. 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 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 [31]. Thermopiles have lower responsivity when compared to microbolometers, but they do not require temperature stabilisation [31].

Microbolometers can be found in most IR cameras today [31]. They are sensitive to IR wavelengths of 8 to 14 µm, which is a part of the long wave infrared region (LWIR) [31]. Measuring part of a microbolometer is known as Focal Point Array (FPA) (Figure 2.12a). FPA consists of IR thermal detectors, bolometers (Figure 2.12b), 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 [32].

Absorber material is either made out of metals such as gold, platinum, titanium, or, more commonly, out of semiconductors such as vanadium-oxide (VOx) [32]. The important property of absorber materials is that electrical resistance changes proportionally with the material's temperature [31]. When IR radiation hits absorber

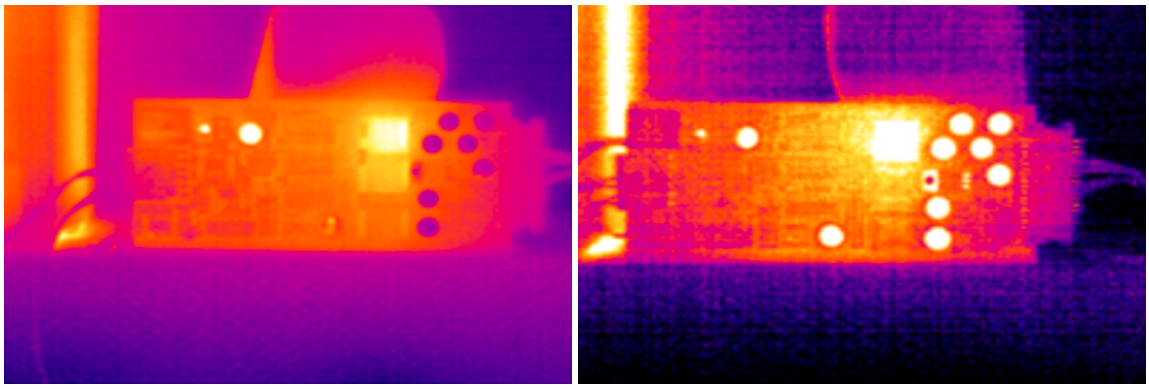


(a) Focal point array (b) Bolometer

Figure 2.12: (a) Focal point array under electronic microscope. (b) Bolometer with $\lambda/4$ resonant cavity. Image source: Vollmer, Möllmann [31]

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 steady-state bias current to the absorber material, while measuring voltage over the conductive legs [31].

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). Comparison between images taken with two cameras with different NETD values can be seen in Figure 2.13.



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

Figure 2.13: 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 [33]

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 [31]. 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 2.13 what a difference of 20 mK means for image resolution and noise.

2.6.1 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 2.14.

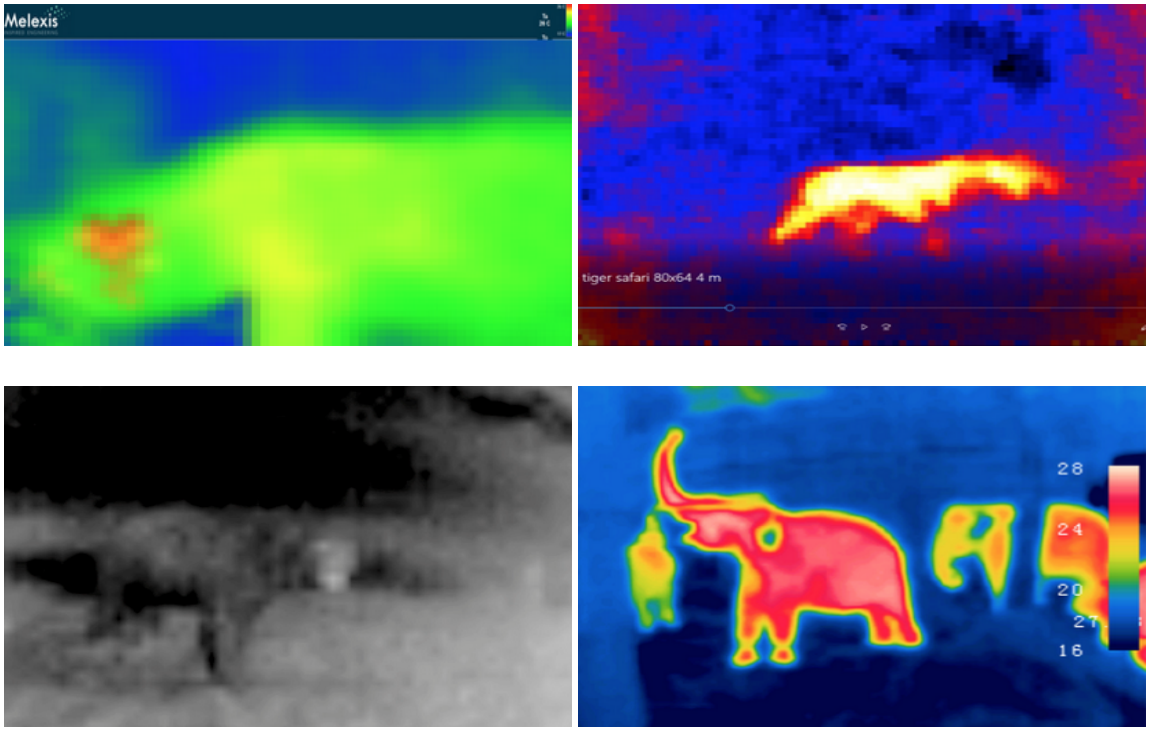


Figure 2.14: Comparison of image quality made by different thermal cameras, MELEXIS MLX90640 (top left), Heimann 80x64 (top right), ULIS Micro80 Gen2 (bottom left) and FLIR Lepton 2.5 (bottom right). Image source: Arribada Initiative [30]

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].

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 [31]. 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 is largely a combination of the workflows shown in Figures 2.1 and 2.9.

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

We then explored both datasets, analysed different class representations, and decided if they were appropriate for accomplishing the 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 analysed 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 the best performing models based on accuracy.

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

3.1 Model objectives

The accuracy of our early warning system should be equal or similar to the one of the human observers, no matter if it is operating in daytime or night time. Although the system will be placed on paths that are traversed regularly by elephants, they are not the only possible objects that can appear on the 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 label a human or a livestock animal incorrectly 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 warning system and defeat the purpose. This means that, besides elephant detection, we should also focus on classifying humans and livestock correctly, while providing a nature/random class for all other unexpected objects or simply images of nature.

It would be beneficial if the thermal camera could 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. The thermal camera FLIR Lepton has a horizontal field of view of 57 degrees. The closer an 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 minimising the execution time of the ML model, or by placing the early warning system in a position that is several metres away from the expected elephant's path. As the latter option might not always be 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 summarise:

- 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.
- The model should be small enough to fit on a microcontroller of our choice, while still leaving some space for the 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. The Neural Networks were designed in TensorFlow 2.4, using Keras high-level API and a Keras Tuner model was used for the 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 used instead. 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 the 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, and visual output

from modules such as Matplotlib is also supported. To use Juptyer Notebook on our cloud instance, we had to install and run it. We could then access the web service simply through a we 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, the major part of the thermal image dataset was provided by the Arribada Initiative [12] [13]. Images in the dataset came from two different locations: Assam, India, and ZSL Whipsnade Zoo, in the United Kingdom.

Assam served as a testing ground. The Arribada team positioned two camera traps at two locations that overlook paths commonly used by elephants. Cameras were built out of Raspberry Pi, a Passive Infrared Sensor (PIR) sensor, an FLIR Lepton 2.5 camera, and batteries, all of which were enclosed in a plastic housing. The 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]

The 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 [34]. With elephants in the enclosure, researchers could move cameras around and get images that were needed. The PIR sensor trigger approach was dropped in favour of a 5 second time-lapse trigger. Two cameras were used again, although, one of them now used an FLIR Lepton 3.5 camera with better resolution.

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

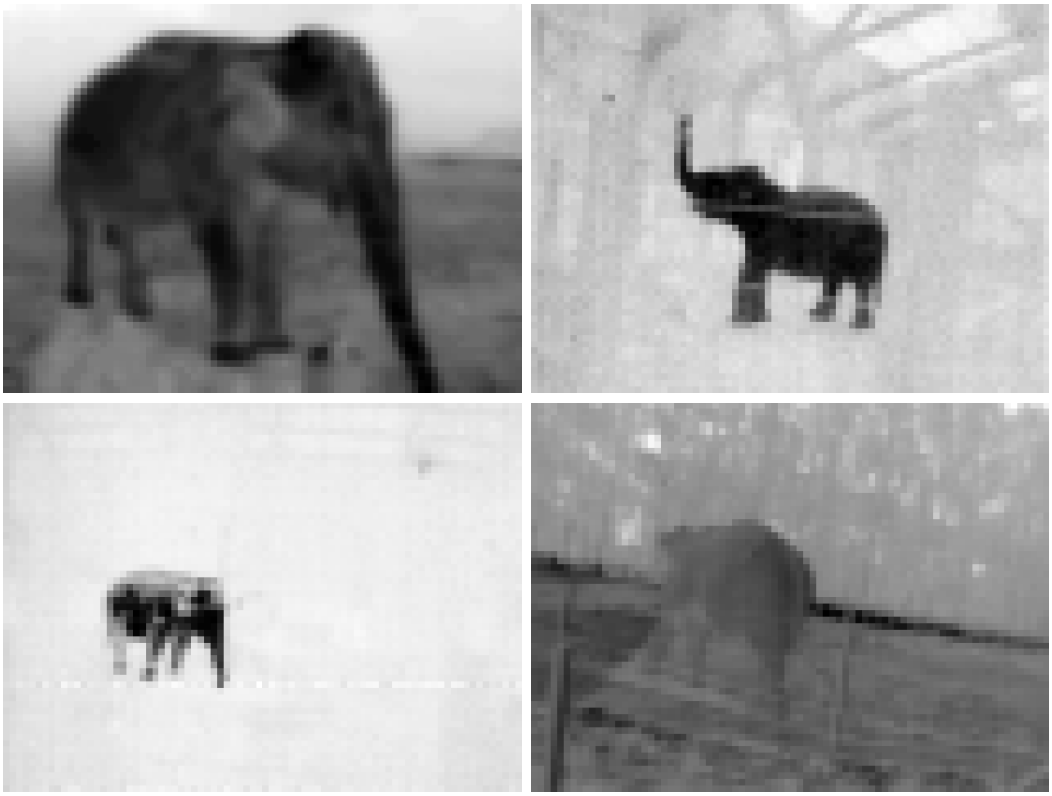


Figure 3.2: Thermal images of elephants from the dataset.

A small part of the 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 the nature/random class. We wanted to gather images as quickly and efficiently as possible, so we built a prototype camera made out of an FLIR Lepton 2.5 breakout

board, a Raspberry Pi Zero, and a power bank. We used an open-source library [35] 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.

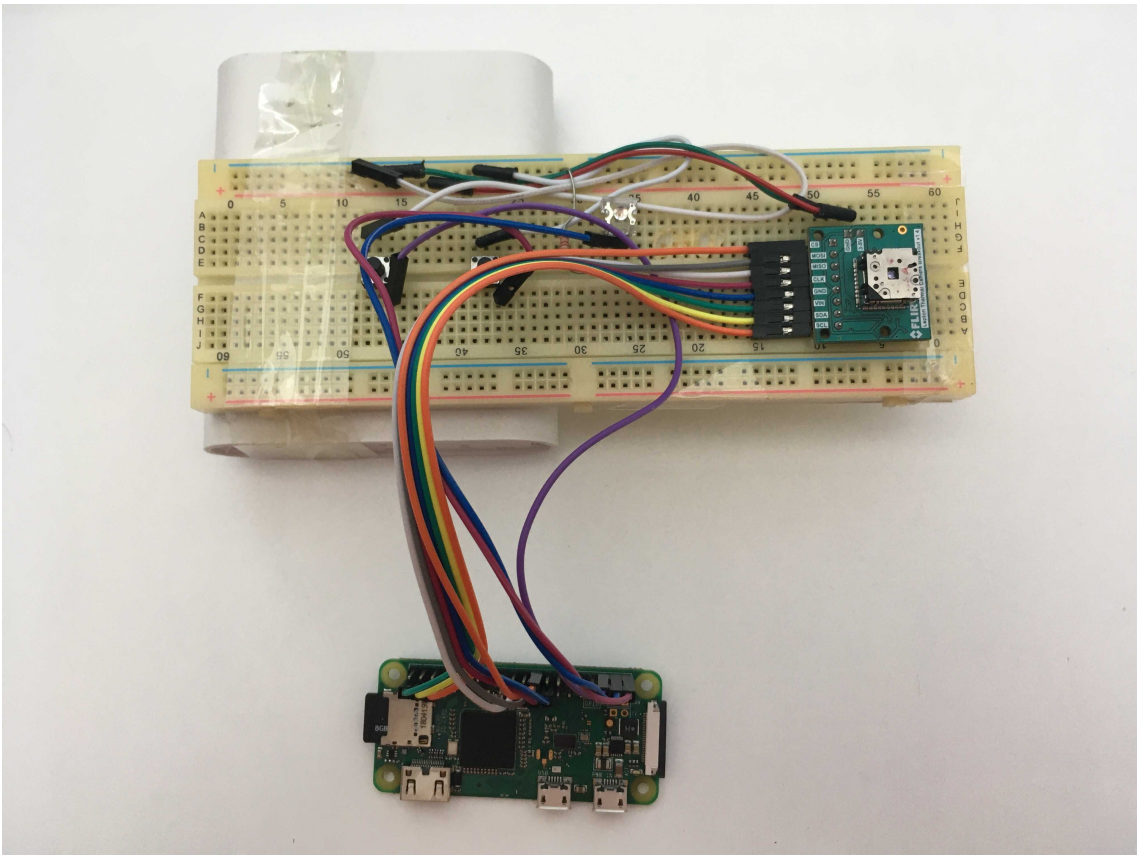


Figure 3.3: Camera setup used for taking thermal images with FLIR Lepton 2.5.

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 removing power from it forcibly would corrupt 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 sorted the images manually into appropriate folders and added them to the dataset.

3.4 Exploring the dataset

A thermal image dataset created by Arribada was given to us in the 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 localisation, the Neural Network draws bounding boxes around objects that it recognises 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 localisation 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, as we only wanted to know if there were 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 on where the image was taken and with what sensor. This enabled us to generate the graph seen in Figure 3.4.

   We used a total of 13,667 images from the thermal image dataset: almost 88 %

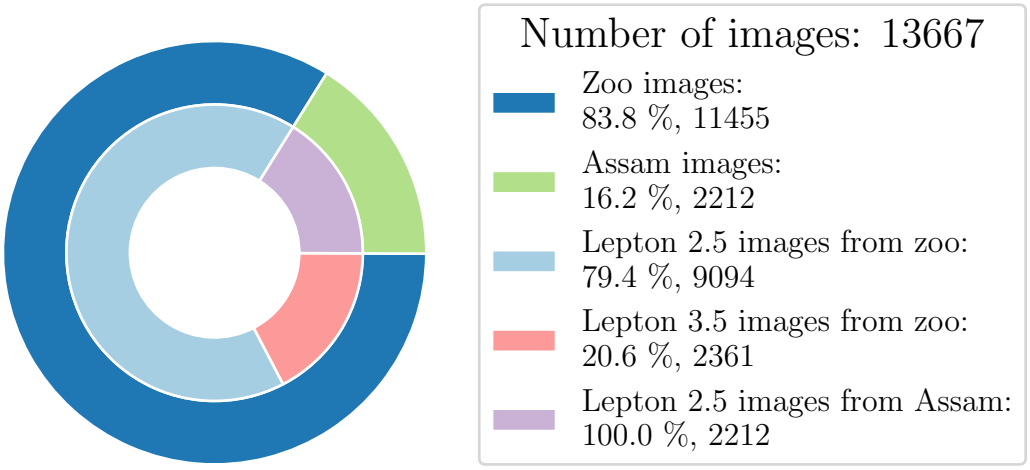


Figure 3.4: Distribution of used images from thermal dataset depending on image location and type of sensor.

of them were made in Whipsnade Zoo, while the rest of them were made in Assam. All images from Assam were made with the 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 inspecting the folder with goat images manually, we saw that it contained mostly images of a herd of goats standing around a single elephant. This folder was usable only for object localisation 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 the elephant class. We therefore dropped goat images out of our training data entirely. Livestock class was replaced with cow class.

4. We also realised that there was a large class imbalance, as seen in Figure 3.5 in favour of the elephant class.

   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.

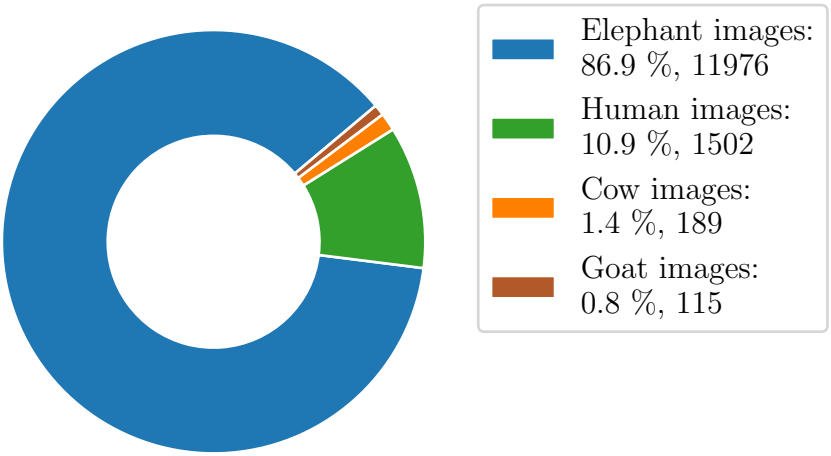


Figure 3.5: Class distribution of thermal images.

3.5 Image preprocessing

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

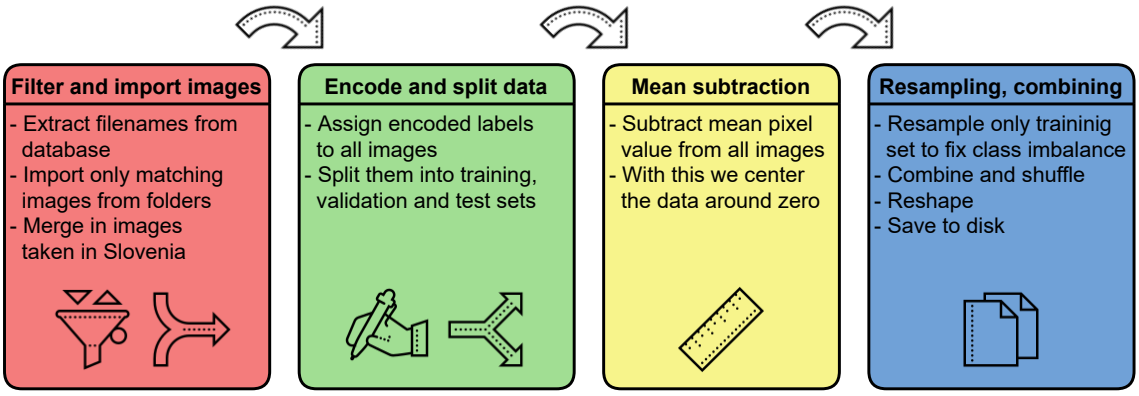


Figure 3.6: Image preprocessing pipeline. Icons source: [11]

At the start of the process, we compared the 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 the 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, the second and third dimensions stored images pixel values (60 and 80 pixels respectively).

The next step was assigning labels to each image. As the output of NNs are numbers, we cannot 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 assess 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 could be overfitting[1]a model and we would have no accurate measure of how well our model would 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 normalisation 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 centring the data. This is a common preprocessing step in every ML image preprocessing pipeline, which is usually combined with standardisation[2]. 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.

[1]Overfitting means that the model performs well on the training data, but it does not generalise well to real-world examples [14].

[2]Standardisation 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 the FLIR camera were all 8-bit encoded and therefore had the same range, this was not needed.

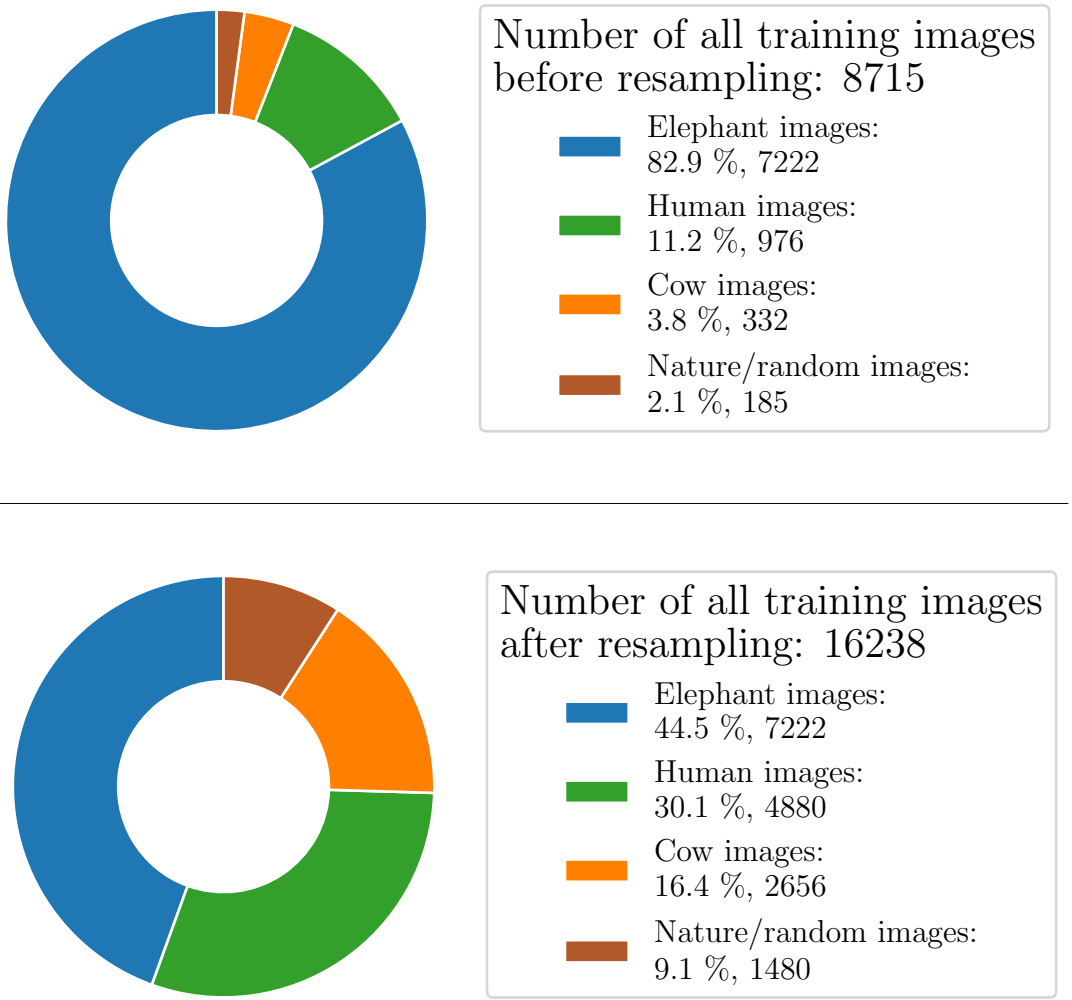


Figure 3.7: Distribution of training images before and after resampling.

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 phases.

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

3.6 Model creation and training

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

For a model architecture we decided to use a simplified version of the 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, which is shown in Listing 3.1.

```
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')
```

Listing 3.1: CNN architecture written in Python using Keras Sequential API.

The model consisted of two pairs of convolutional and max-pooling layers, followed by a final convolutional layer. For activation the 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. Pooling 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[3].

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. The model was set to use the Adam optimizer and Sparse Categorical Crossentropy loss function. Adam is an upgraded version of the gradient descent method, which adapts the learning rate automatically to decaying gradients [14]. It is generally easier to use than gradient descent, as it requires less tuning or learning rate hyperparameters. Sparse Categorical Crossentropy loss function is used when building a multi-class classifier.

The above set hyperparameters follow the 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 cannot be chosen heuristically.

To find hyperparameters that would yield the highest accuracy we used the Keras Tuner module. The exact configuration of a Keras Tuner module and comparison of trained models is presented and discussed in Section 5.1.

3.7 Model optimisation

Keras supports saving models in h5 format, which model's architecture, values of weights, and information used while compiling the model. h5 format cannot 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.

[3]The 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 regularisation technique, which makes models more robust to the input data [14].

The TFLite converter can convert a model in .h5 format into four differently optimised 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 optimisation but slower than full integer optimisation.
- **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, although, 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 the 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, the xxd tool creates a hex dump of our model,and formats it as a char array in the C programming language.

To automate the optimisation process we wrote a Python script that took the model in raw .h5 format and converted it into every possible version of the optimised tflite model. Each model was then processed with the 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, because 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[4]with data with the Edge Impulse account, and transferring data to a specific project or by using the Edge Impulse command-line tools to upload image data from a computer directly to a project. We chose the S3 bucket approach, as once the data was uploaded it was trivial to transfer it to different projects.

After the data were uploaded, the rest of the NN design was done through the Edge Impulse web interface. In Figure 3.8 we can see the so-called Impulse Design tab, where we design a Neural Network by choosing different blocks.

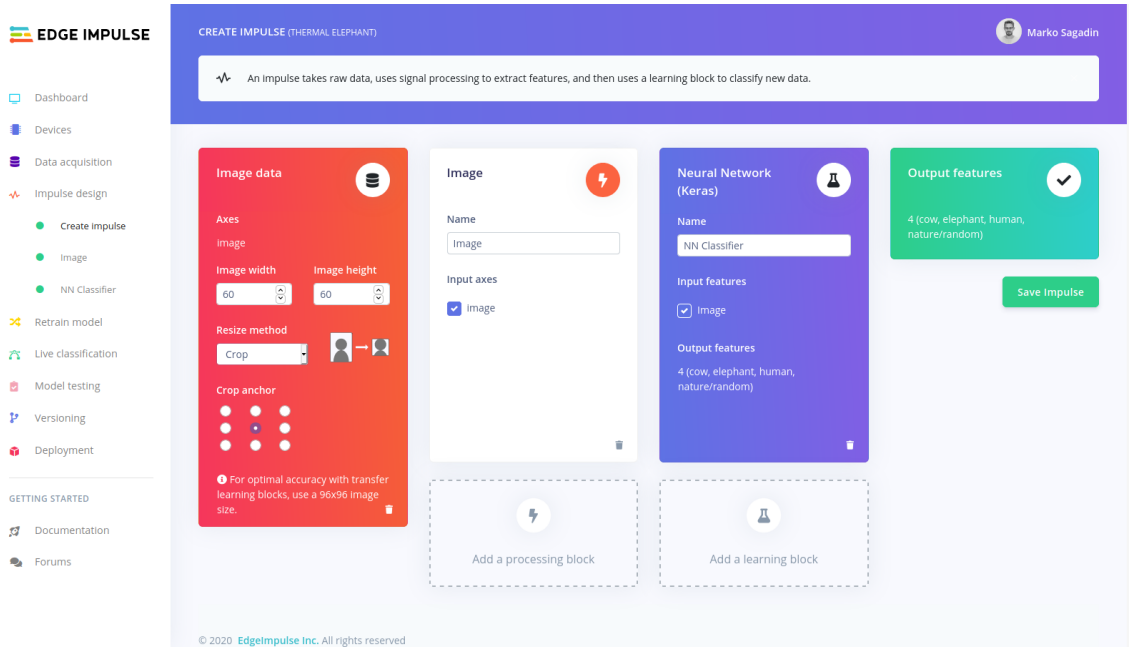


Figure 3.8: Creating a Neural Network in Edge Impulse Studio.

With input blocks, we tell what kind of data are we inputting, either image or time-series data, and with processing blocks we decide how are we going to extract

[4]Simple Storage Service or S3 is another service provided by Amazon, used for storing a large amount of data in the cloud.

features. Input and preprocessing blocks always return same output given the same input, while learning blocks are trainable and learn from previous experiences. For the 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.

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 resize our images to 60 x 60 pixels. Image input block offers different resize options, which are also shown in Figure 3.9. We chose crop option, where the excess pixel were simply dropped. For the processing block, we selected the image processing block as this was the only possible choice. For the learning block we were using both Keras's Neural Network block and Transfer Learning block.

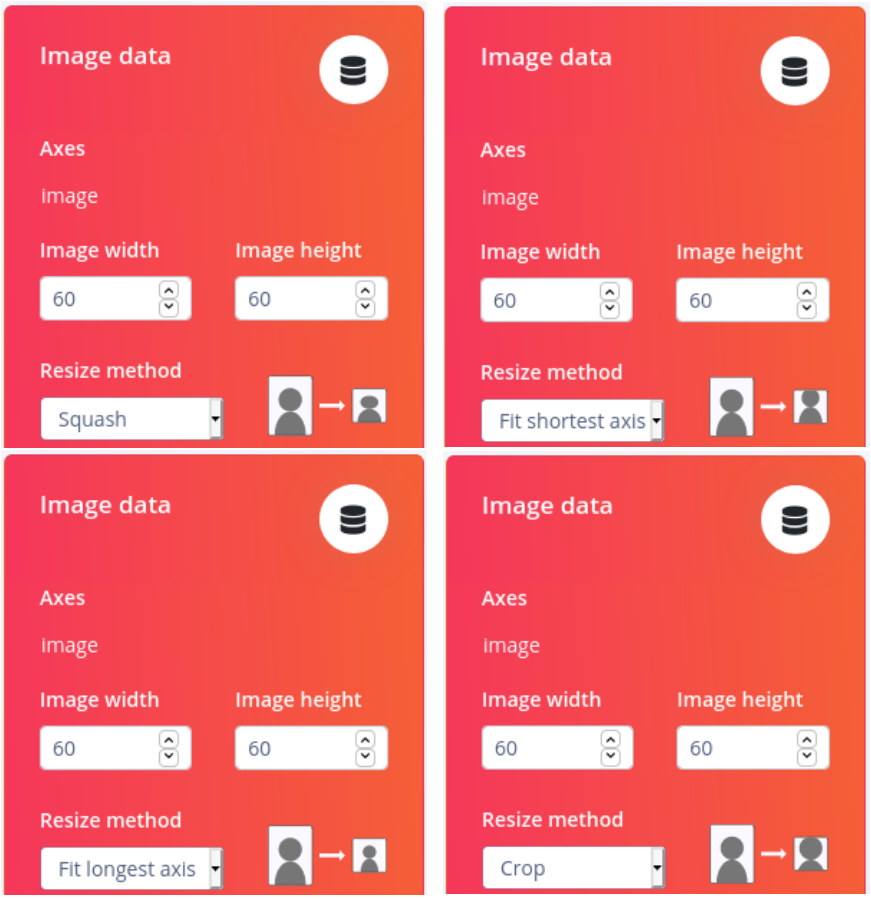


Figure 3.9: Different resizing methods

Both of the blocks are 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 make our adjustments. Settings such as learning rate, number of epochs[5], and confidence rating are also available, regardless of the option we chose. Training of Neural Networks inside Edge Impulse Studio is done in a cloud, so as users we do not have to worry about setting up a development environment. We only have 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 on-device inference would take, based on the frequency and the processor of the microcontroller.

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

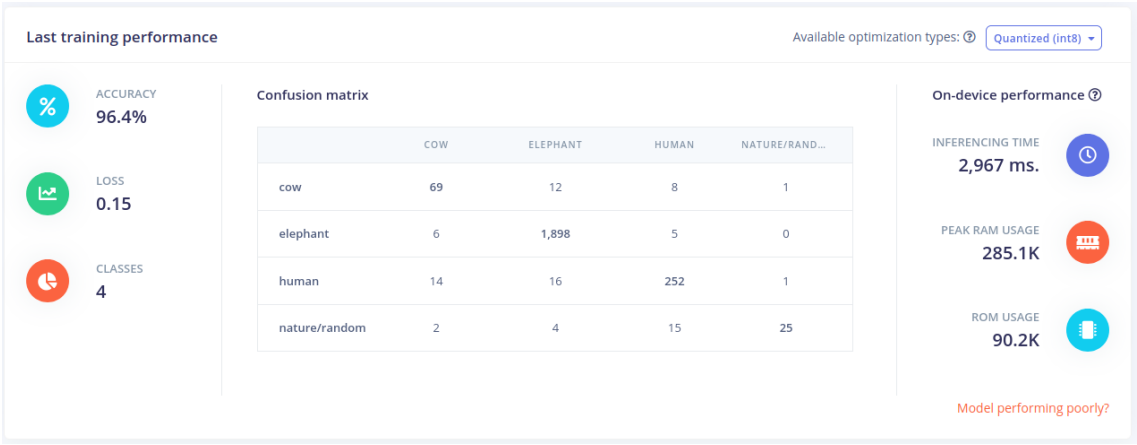


Figure 3.10: Training performance report.

The final step was deploying the trained model to the microcontroller. This step is fairly simple, as Edge Impulse provides a few example projects on their GitHub for the different platforms that it supports. As we wanted to compare the performance of the models on STM32f767ZI, we chose the Mbed platform. We copied the example Mbed project from GitHub, and in the Edge Impulse Studio we selected to generate

[5]Number of epochs tells us how many times the whole training set passed through model, during training process.

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 runs the inference continuously on one image and outputs results over the serial port. A performance comparison between this example project and our implementation is done in Section 5.2.3.

4 Design and implementation of the early warning system

General structure and tasks of an early warning system were already described in chapter 1.2. As mentioned before, an early warning system consists of two different components:

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

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

The general block diagram of an embedded system with a thermal camera is presented in Figure 4.1

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

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

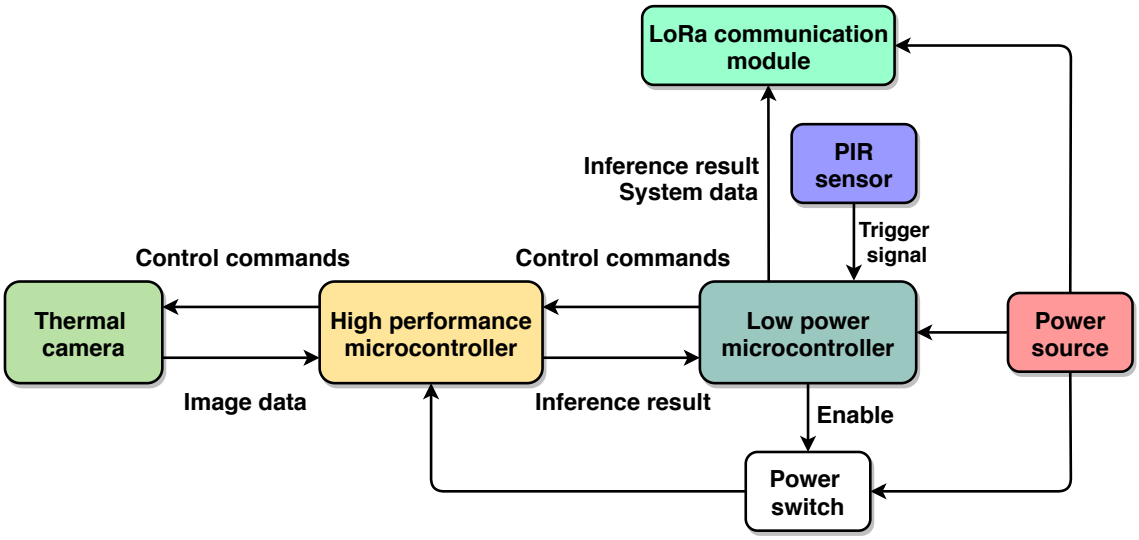


Figure 4.1: General block diagram of an embedded system

detects any IR radiation of a passing object.

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

The thermal camera only communicates with the high-performance microcontroller, which configures it and requests image data. Those data are then input into a Neural Network algorithm, which computes probability results that are sent back to the low-power microcontroller. The low-power microcontroller then packs the data and sends them over the radio through a wireless communication module. The power source to the high-performance microcontroller and thermal camera is then turned off to conserve power. A diagram of the described procedure can also be seen in Figure 4.2.

Figure 4.2: Diagram describing the behaviour of the embedded early warning system

4.1 Hardware

In this Section, we present the concrete components that we used to implement the embedded part of the early warning system. The hardware version of the embedded system diagram is presented in Figure 4.3. The system consists of various development and evaluation boards.

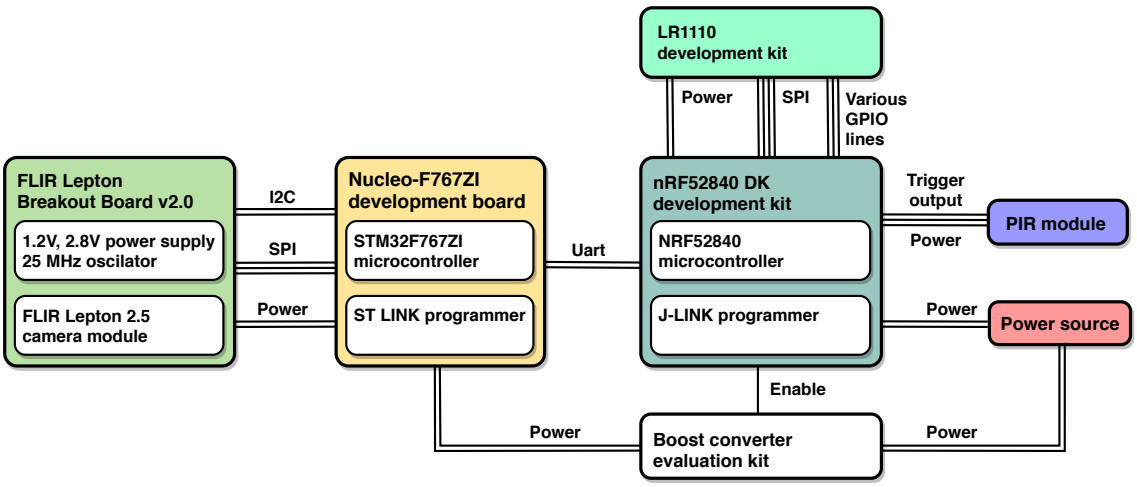


Figure 4.3: Hardware diagram of the embedded early warning system

4.1.1 Nucleo-F767ZI

Nucleo-F767ZI (seen in Figure 4.4) is a development board made by STMicroelectronics. It features an STM32F767ZI (STM32) microcontroller with a Cortex-M7 core, which has 2 MB of flash, 512 kB of SRAM, and can operate at a clock speed of 216 MHz. It also features memory caches and a flash accelerator, which provide an extra boost in performance. It is convenient to programme, as it includes an onboard ST-LINK programmer circuit.

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

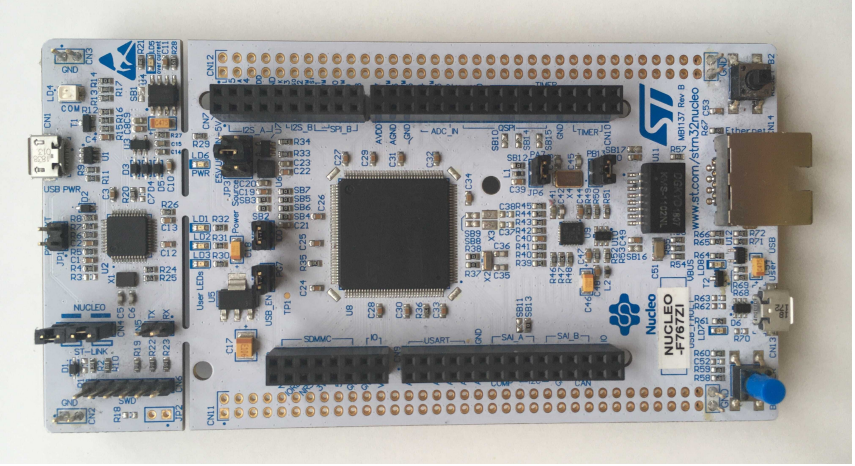


Figure 4.4: Nucleo-F767ZI development board

4.1.2 nRF52840 DK

For the part of the system which had to contain the low-power microcontroller and would control the communication module and power control for the Nucleo-F767ZI board, we decided to use the nRF52840 DK development kit. The development kit, made by Nordic Semiconductor, can be seen in Figure 4.5

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

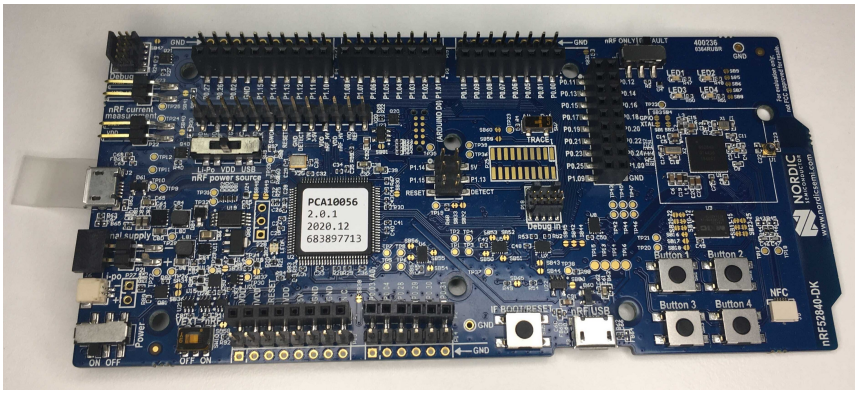


Figure 4.5: nRF52840DK development board

4.1.3 LR1110 development kit

For the role of the LoRa transceiver module we decided to use Semtech's development kit which uses the LR1110 chip. LR1110 is a multi-functional solution, as it contains a LoRa transceiver, and GNSS and WiFi geoposition scanning modules. The development kit seen in Figure 4.6 contains an LR1110 chip, three different antennas and their respective tuning networks. It comes in a convenient Arduino shield form factor, which means that we can attach it directly to the nRF52 without any jumper wires.

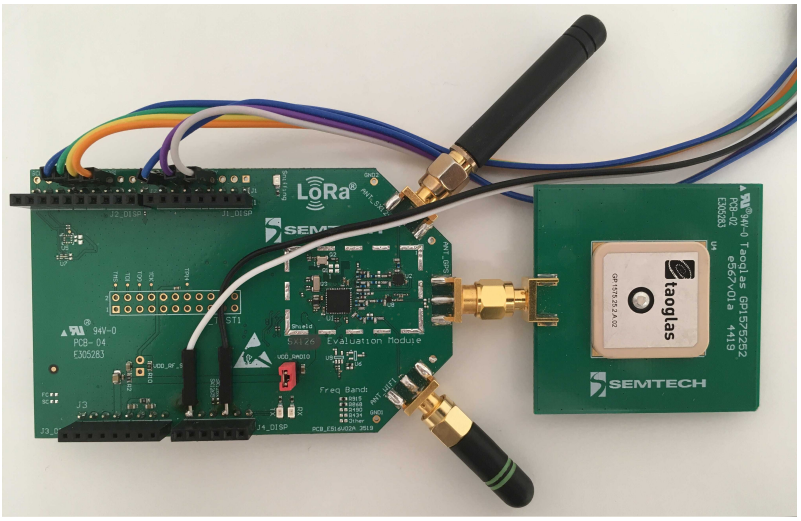


Figure 4.6: LR1110 development kit

4.1.4 Boost converter evaluation kit

The power to the Nucleo-F767ZI board and the FLIR camera is provided by the MAX17225ENT+T boost converter chip. The breakout board containing the chip is shown in Figure 4.7. Operating the boost converter chip is simple, its enable line can be connected directly to a microcontroller pin driving it high enables output, and driving it low disables it. The output voltage is controlled by an external resistor.

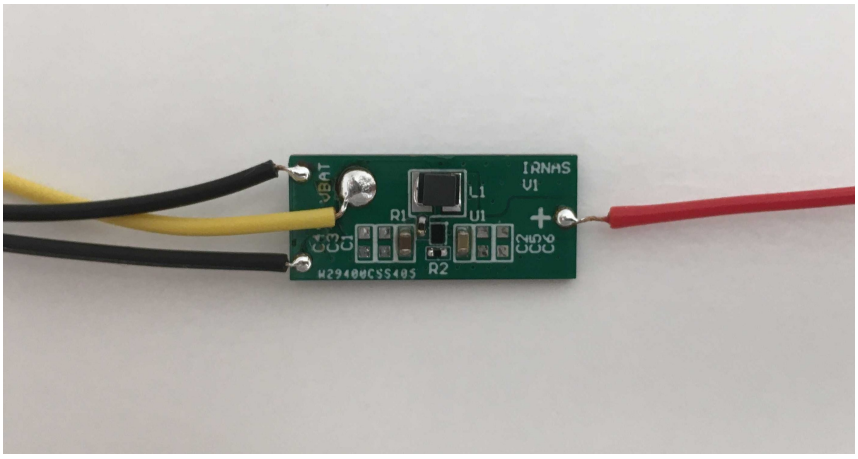


Figure 4.7: MAX17225ENT+T boost converter breakout board

4.1.5 FLIR Lepton 2.5 camera module and Lepton breakout board

Section 2.6 described what kinds of thermal cameras exist and how they work, and Section 2.6.1 described why the FLIR Lepton 2.5 was chosen. However, not much was said about what sort of support circuitry the FLIR camera needs and how we use it.

The FLIR Lepton camera is powered from two different sources, 1.2 V and 2.8 V, and requires a reference clock of 25 MHz. All of this is provided by the Lepton breakout board, which can be seen in Figure 4.8. The front side of the breakout board contains an FLIR module socket, and the backside contains two voltage regulators and an oscillator. The breakout board can be powered from 3.3 to 5 V, and also, conveniently breaks out all communication pins in the form of header pins.

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

- AGC – Automatic Gain Control, affects image contrast and quality
- SYS – System Information
- VID – Video Processing Control
- OEM – Camera configuration for OEM customers
- RAD – Radiometry

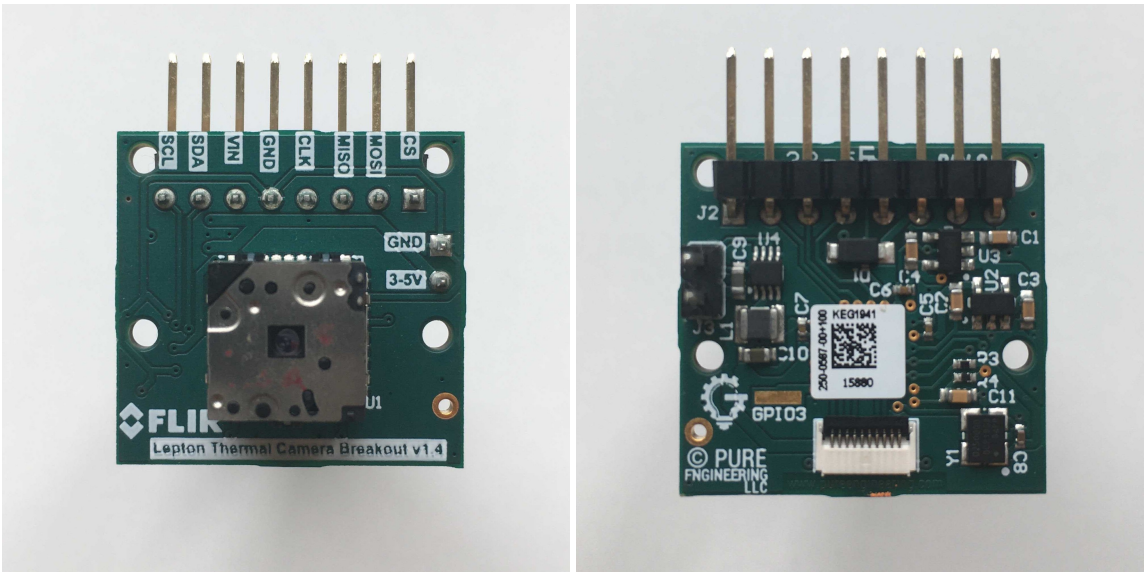


Figure 4.8: Front and back side of FLIR Lepton breakout board with thermal camera module inserted.

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

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

4.1.6 PIR Sensor

We used a cheap, generic PIR sensor, that can be seen in Figure 4.9. It has two potentiometers, which are used to adjust the sensor's sensitivity and detection delay. The PIR sensor runs on 3.3 V, which enables us to power it directly from the nRF52.

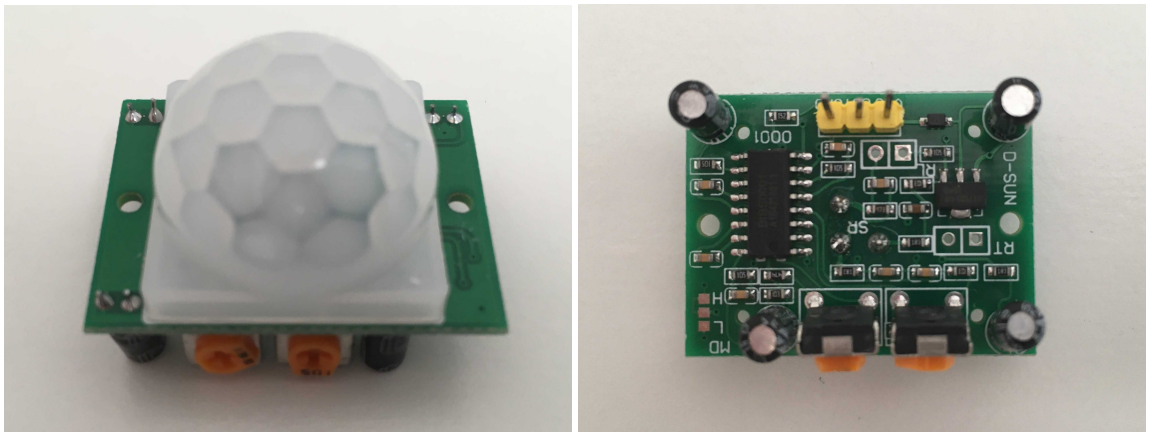


Figure 4.9: Front and back side of a PIR sensor.

4.2 Firmware

4.2.1 Tools and development environment

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

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

4.2.1.1 Development environment for STM32F767ZI

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

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

As libopencm3 does not provide `printf` functionality out of the box we used an excellent library by GitHub user mpaland [36].

4.2.1.2 Development environment for nRF52840

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

4.2.2 Architecture design

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

As seen in Figure 4.10 we split the firmware into a hardware module and application module.

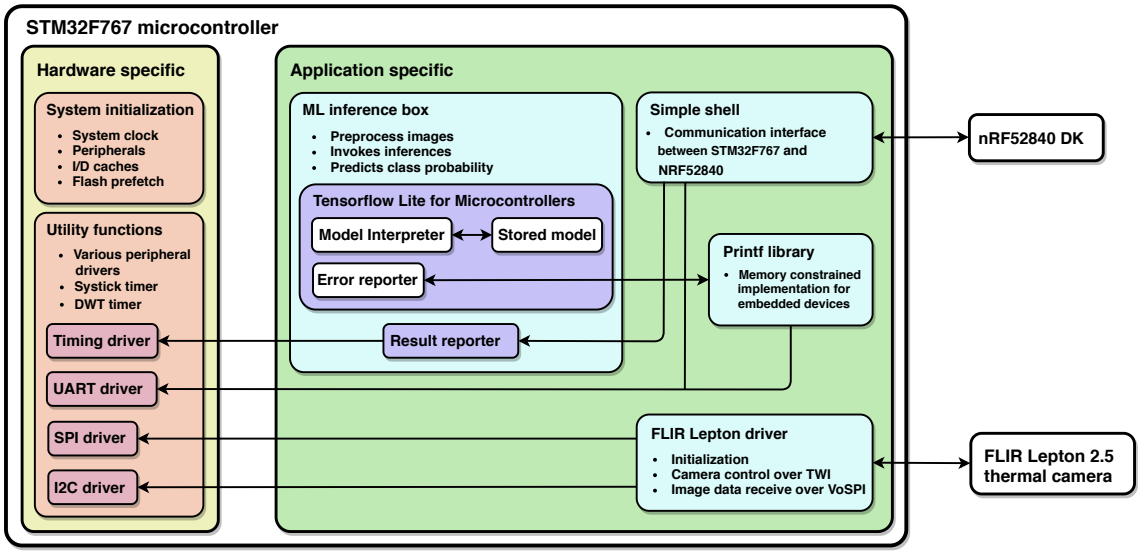


Figure 4.10: Architecture diagram of the firmware that is running on the STM32 microcontroller.

The hardware specific module is using mostly libopencm3 API to set the system clock and initialise peripherals. Small function wrappers had to be written to make

use of various peripheral drivers easier.

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

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

The architecture diagram for nRF52 can be seen in Figure 4.11. For the nRF52 microcontroller we did not have to write any peripheral drivers, as they were provided by Zephyr itself.

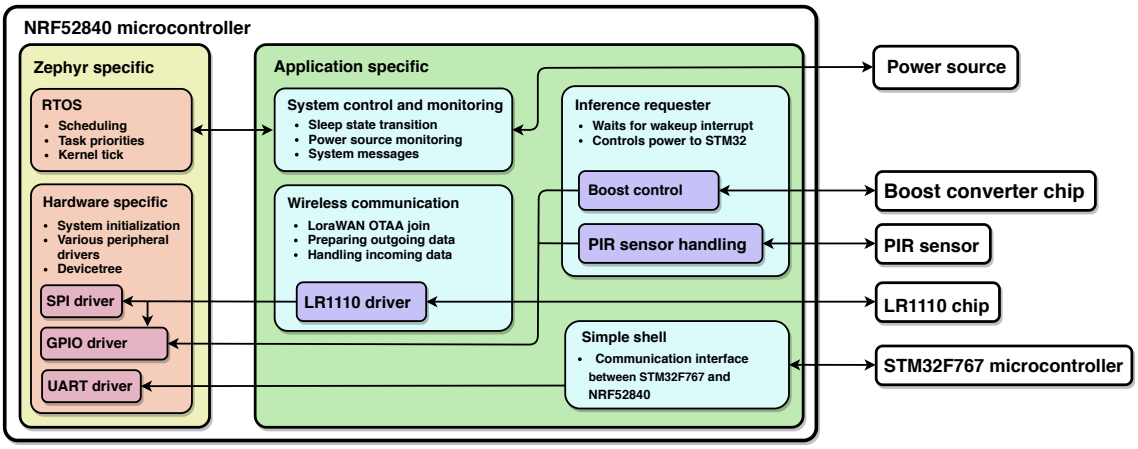


Figure 4.11: Architecture diagram of the firmware that is running on the nRF52840 microcontroller.

Extra care was given to low power consumption. That meant that the nRF52 had to spend most of its time in low-power mode, only waking up for regular system checks and PIR trigger signals. When the a PIR trigger signal is received, the interrupt wakes up the nRF52, which then enables a boost converter, thus enabling power to the STM32 and FLIR Lepton camera. Handling transitions to low-power state in Zephyr is quite straightforward, although, each individual peripheral has to be turned off explicitly. When entering a low-power state we turned off the UART and SPI peripherals, while GPIO stayed active, as we needed a wakeup interrupt. After

receiving the interrupt we had to turn on each peripheral manually.

For the communication interface, we decided to implement a simple UART shell module. We also wrote a communication module, which took care of controlling the LR1110 chip, joining the LoRaWAN network, preparing outgoing messages and sending them over the LoRaWAN network.

4.2.3 FLIR Lepton driver

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

TWI is a variation of an I2C protocol, but instead of 8 bits all transfers are 16 bits. The internal structure of the Lepton's control block can be seen in Figure 4.12. Whenever we are communicating with the FLIR camera we have to specify which subsystem we are we addressing, what type of action we want to do (get, set or run), length of data and the data themselves.

We wrote the driver in such a way that the API hid low-level details of exact data transfers. Two examples of such an API can be seen in Listing 4.1.

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

Listing 4.2 shows our implementation of a `get_picture` function that was reading images from VoSPI protocol.

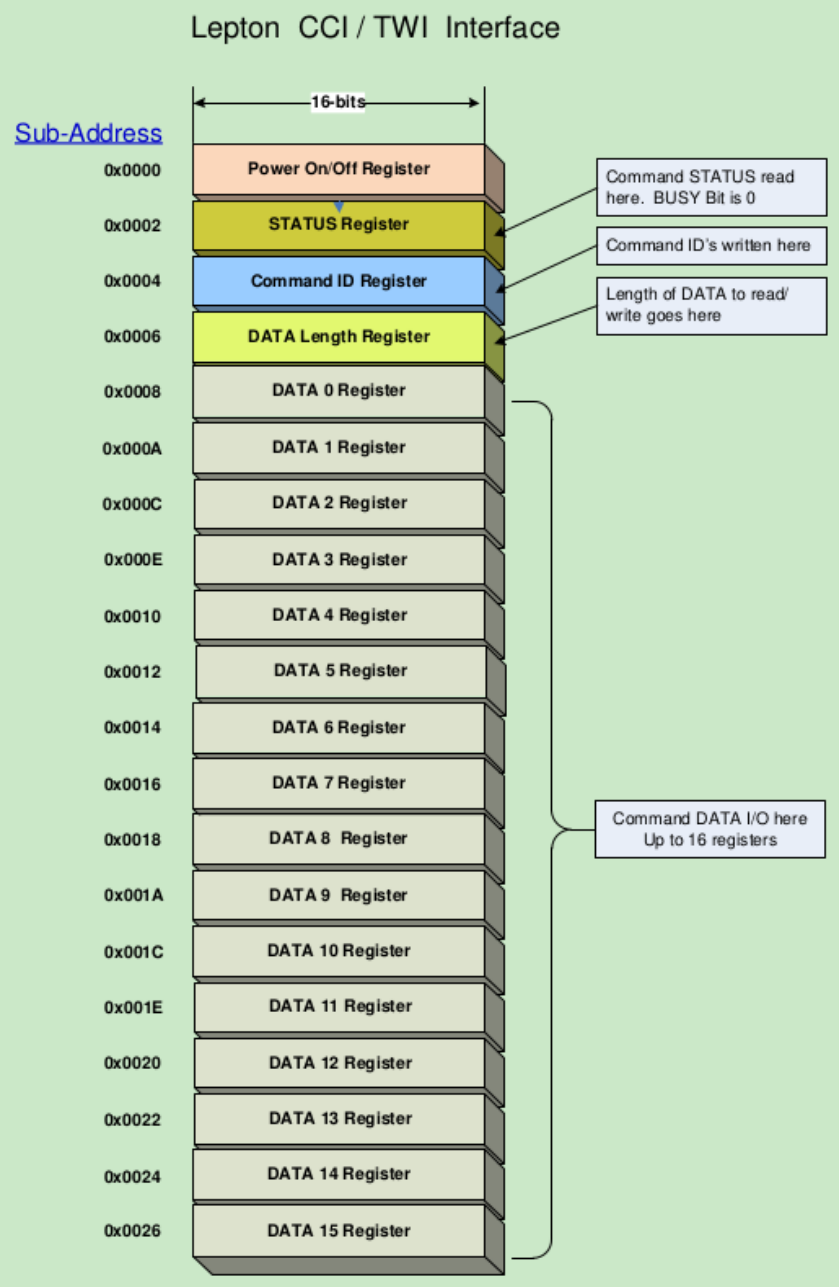


Figure 4.12: Command and control interface of an FLIR Lepton camera. Image source: [37]

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

To capture images from the VoSPI stream we implemented a finite state machine with three states: `INIT`, `OUT_OF_SYNC` and `READING_FRAME`. State `INIT` executes the chip select sequence expected by the FLIR camera. After that, we start clocking out a stream of VoSPI frames over the MISO line. Whenever we start communication, we do not know where exactly in stream we are, additionally, FLIR is also transmitting discard packets between valid frames. To solve this problem we have to check the ID field of every VoSPI packet and look for an ID byte with a value `0x00`, while discarding packets with values `0xFF`. When the first frame row is detected we simply start storing all incoming frame rows into a `frame` variable, while checking that the ID byte is correct. We loop until all VoSPI packets of a single frame are received. If we somehow missed the VoSPI packet, we return early from the function.

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

Listing 4.1: Examples of FLIR Lepton driver API.

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

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

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

Listing 4.2: Example of finite state machine implementation for reading FLIR images over SPI.

4.2.4 Simple shell

The simple shell module controls the execution of all subroutines inside the STM32 firmware. The nRF52 microcontroller acts as a host and sends commands to the STM32, which executes commands and sends back the results. Listing 4.3 shows the main `simple_shell` function.

```
/*!
 * @brief       Supported shell commands
 */
typedef enum
{
    INVALID_CMD,
    BLINK,
    ML,
    FLIR,
} shell_cmd;

/*!
 * @brief       Entry point to simple shell, which does not
 *              ever return. It calls all other modules and
 *              functions.
 */
void simple_shell()
{
    char buf[SHELL_BUF_LEN];
    uint16_t len;
    shell_cmd cmd;

    put_line("$");
    while (1) {
        len = get_line(buf, SHELL_BUF_LEN);
        if (len) {
            cmd = parse_command(buf, len);

            if (execute_command(cmd)) {
                get_command_response(cmd, buf, SHELL_BUF_LEN);
                put_line(buf);
            }
            else {
                put_line("\nNOT OK\n");
            }
        }
    }
}
```

Listing 4.3: Code snippet of simple shell implementation.

At the start of the while loop, function `get_line()` returns when it receives a string, terminated with a newline character. Function `parse_cmd` then decides with a set of `strcmp` functions if the received command is supported. If yes, that command is later executed by the function `execute_cmd` and a response is returned, returning

result in the case of successful execution, or a fail reason in the case of failure.

The module is written to be easily scalable, and when new functionality is added, it is trivial to extend the number of possible commands. Once we knew that the UART communication worked correctly, we could issue commands directly from the computer's serial port, which enabled us to develop and test firmware for the STM32 separately from the nRF52 firmware.

4.2.5 MicroML and build system

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

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

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

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

1. We wanted to keep the project-specific code, libopencm3 code and TFLite Micro code separated.
2. We wanted a system where it would be easy to change a microcontroller specific part of a building process.
3. We wanted to reuse the static library principle that we saw in the TFLite Micro build process.

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

To solve our problem we started developing a small project that we named MicroML[2]. MicroML enables users to develop ML applications on microcontrollers supported by libopencm3. The project's directory structure can be seen in Figure 4.13

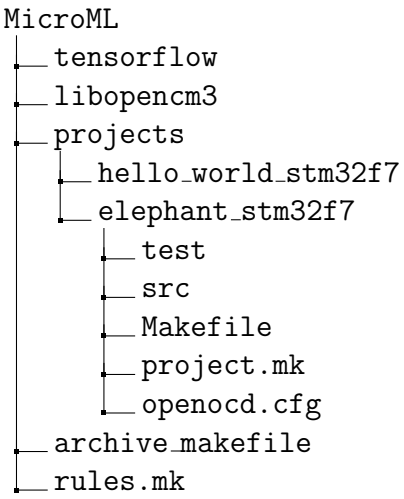


Figure 4.13: Directory structure of the MicroML project.

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

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

- **project.mk** - It contains information on which files need to be compiled inside the project folder. It defines for which microcontroller the code needs to be compiled, and what kind of optimisation flags should be used.
- **openocd.cfg** - A configuration file that tells OpenOCD which programmer interface needs to be used to flash a microcontroller, and the location of the binary file that needs to be flashed.
- **Makefile** - The project's makefile that gathers source files inside the project folder. It makes it possible to call `make` directly from the projects directory, which eases the development process. It does not specify any building rules; those are specified in the `rules.mk` file in the 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 4.14 represents the complete build process.

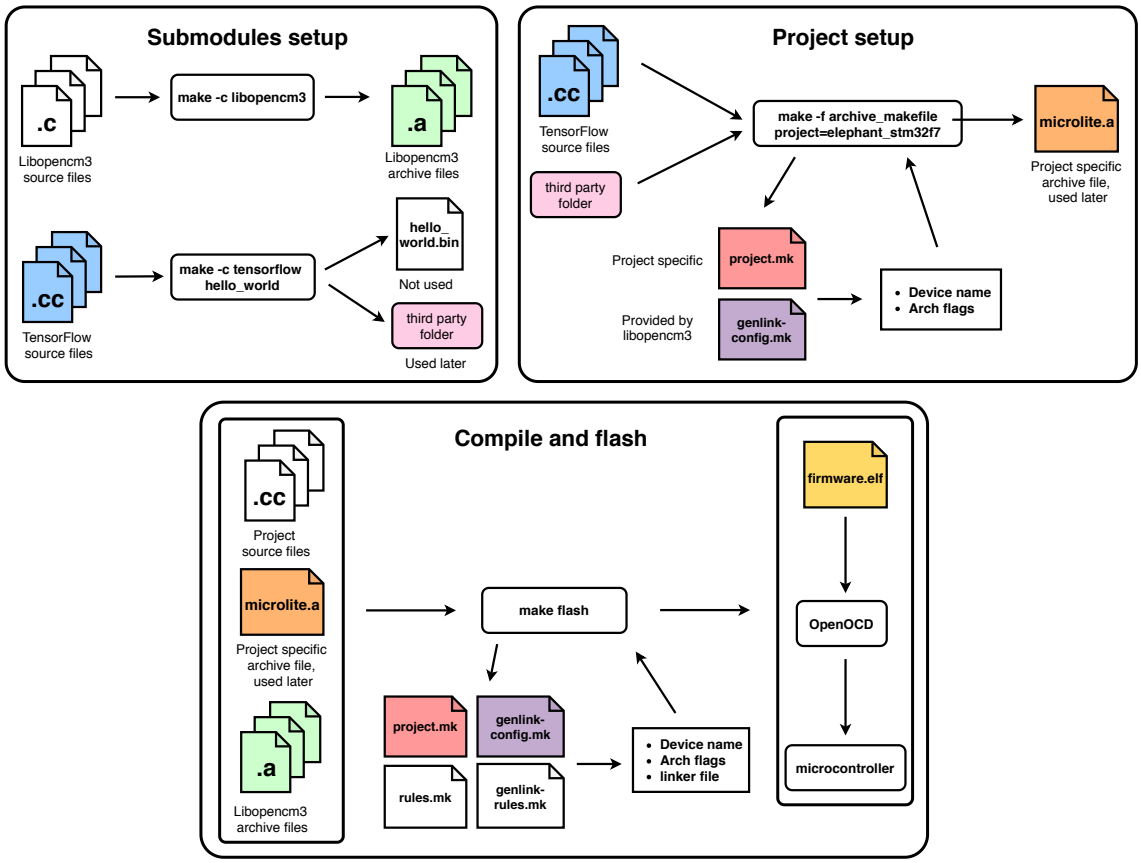


Figure 4.14: Build system of MicroML project.

In the *submodules setup* stage we first compile both of the submodules; this step requires two makefiles that are already provided by each submodule. Compiling libopencm3 creates a group of archive files (static libraries), which contain all platform specific code. Compiling a TensorFlow Hello World example does not produce any archive files that we would need, although, it does execute several scripts which download several different third party files. The TFLite Micro library depends on these files, which means that MicroML does as well. The *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 the *project setup* stage. From the main directory we execute `make` with `archive_makefile` and define the `PROJECT` variable with the name of our project. `Archive_makefile` looks into `project.mk` and extracts the `DEVICE` variable. Libopencm3's `genlink-config.py` script then determines which microcontroller specific compilation flags[3]are needed. All needed TensorFlow source files and third party files are then compiled and a project specific `microlite.a` archive file is created in our project's folder.

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

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

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

4.2.6 Running inference on a microcontroller

TFLite Micro API is fairly simple to use, and general enough that it can be copied from project to project without many modifications. Listing 4.4 shows a simplified inference code example, copied from our project. As a first step, we needed to define the size of the `tensor_arena` array, which holds the memory of input, output, and intermediate arrays. The exact size of the `tensor_arena` is determined by trial and error: We set it to some big value and then decrease it in steps.

In lines 9 and 10 we created an instance of the `ErrorReporter` object. This object serves as a thin wrapper around the platform specific `printf` implementation. If some part of TensorFlow code crashes, `ErrorReporter` notifies us what went wrong. In line 13 we pull in our ML model in hex dump format that we created with xxd. `Full_quant_model` is defined in a different file, not seen in this example.

In lines 16 to 24 we created an operation resolver. One way to do it is to specify each required operation specifically (which is done in the example) or simply pull in all operations. The latter approach is not recommended, as it results in a large binary size. To find out exactly which operations were required we used the online tool Netron [39], which showed us a deconstructed view of a trained model.

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

```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));
}
```

Listing 4.4: Example of TensorFlow Lite inference code in C++.

In line 47 we finally invoked interpreter and ran inference on input data. The whole expression was surrounded by the timing functions, which were used to keep track of the time spent computing inference.

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

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

4.3 Server-side components and software

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

The system that we use consists of different tools, each one with a distinct task. These tools are The Things Network (TTN), Node-RED, InfluxDB and Grafana. The flow of information and tasks for each tool is presented in Figure 4.15.

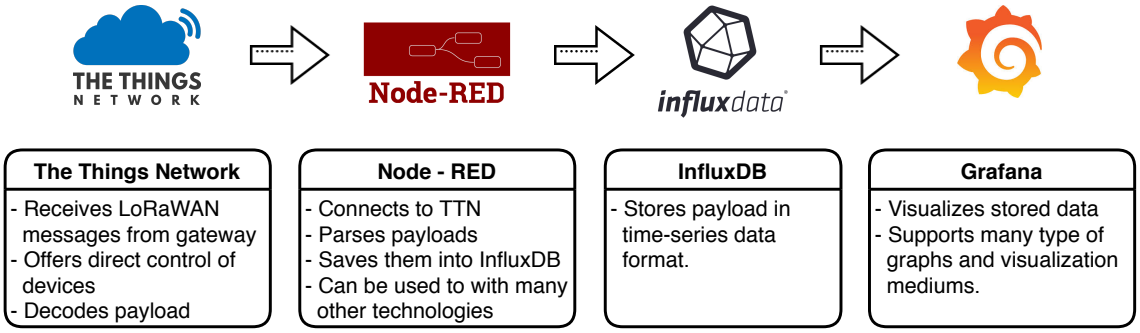


Figure 4.15: Server side flow of information. Icons source: [11]

TTN is responsible for routing packets that are captured by a gateway to the application server. Since it is open-source and free, anyone can register their gateway device into the network and, thus, help to extend it. TTN is web-based, so we can see

payload messages directly in the browser. Since data are usually encoded in binary format, we can provide a decoder-script written in JavaScript and TTN will pass each message into it automatically, thus decoding it.

Node-RED functions as a glue logic that parses packets and shapes them into a format that is required by InfluxDB. Node-RED provides a browser-based flow editor, where actual programming can be done graphically. Logic is programmed by choosing different blocks, called *nodes*, and connecting them. This is convenient, as Node-RED provides different nodes for communicating with different technologies, such as MQTT, HTTP requests, emails, Twitter accounts and others. In our use case, we needed to use the nodes seen in Figure 4.16. The node *Elephant Gateway* is connected to a specific application on TTN, which is used for the collection of packets from our devices in the field. Any packet that will appear in that TTN application will also appear in Node-RED. The node *Parse packet* extracts information contained in each packet and stores it in a specific format, which is finally sent to the *Elephant Database* node.

Figure 4.16: Node-RED flow

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

Data are then visualised in Grafana. Grafana is an open-source analytics and monitoring solution. Users define which database is set as a source, and Grafana provides graphical controls which are, at some point, converted into SQL-like language, understandable to InfluxDB. Grafana provides different types of visualisations, 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.

An example of a Grafana graph can be seen in Figure 4.17.

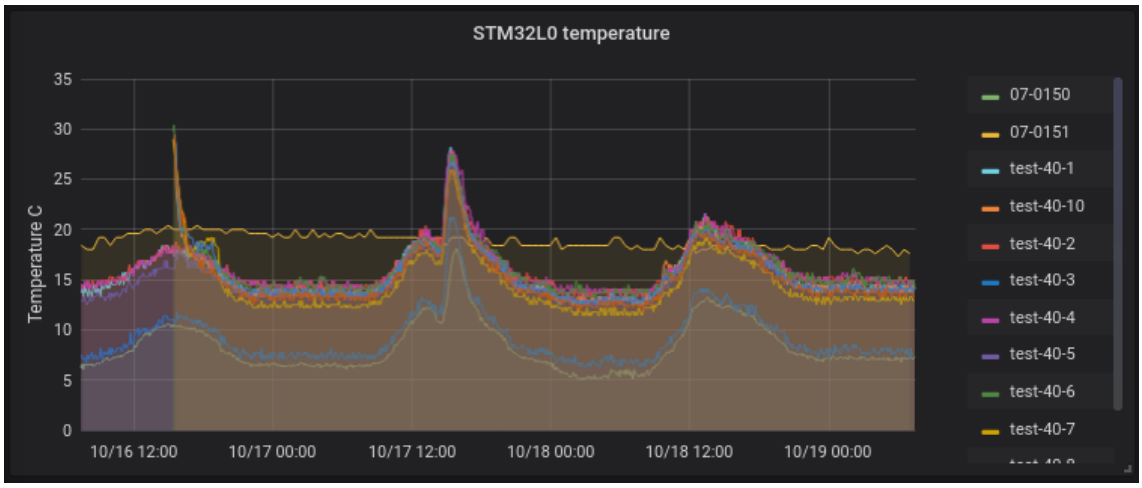


Figure 4.17: Example of a Grafana graph.

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

5 Measurements and results

5.1 Comparison of models

As mentioned in Section 3.6, we used a Keras Tuner model to find the hyperparameters that would yield the highest accuracy. Instead of hard-coding hyperparameters when building a model with Keras API, we defined a search space of possible values with `HyperParameter` class, and used that as a hyperparameter.

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

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

To distinguish models from one another we decided to mark them with a number and letters *a*, *b*, *ei* and *tl*. Models with letters *a* and *b* were trained using our system. Models marked with *ei* and *tl* were created in the Edge Impulse Studio. More specifically, *ei* models have a same basic CNN architecture as our models, while *tl* models were trained by the Transfer Learning technique, which reuses a pre-trained MobileNetV2 architecture. The Tables that are shown below list one of the metrics as accuracy. By accuracy we mean validation accuracy; it tells us how well the model performed on a validation dataset.

5.1.1 Hyperparameter search space and result's analysis

The general structure of the CNN model was already described in Section 3.6 and in Figure 3.1. We decided to search for the following hyperparameters:

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

The possible values of hyperparameters (also known as a hyperparameter search space) are specified in Table 5.1.

Table 5.1: First hyperparameter search space

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

The search spaces of `FilterNumX`, `DenseSize` and `DropoutRate` hyperparameters were chosen based on initial training tests conducted on a thermal image dataset and other various models that were trained on similar data. The value of `FilterSize` is usually 3 x 3; however, most of the example ML projects that we could find on

the Internet were training on image data of the same dimensions. We wanted to test how a filter with the same ratio of dimensions as image data (3 x 4 and 60 x 80 respectively) would perform. The hyperparameter `learning_rate` was chosen heuristically; we saw that 10 times higher values, such as 0.001 or 0.003, would leave the model's accuracy stuck at suboptimal optima, from where it could not be improved anymore.

We also had to set 3 variables that affected directly how long the random search would last. From initial tests we saw that models usually reached maximum possible accuracy at around the 20^{th} epoch, so, to give some headroom we set the number of epochs to 25. We kept the batch size relatively small, at 100, which meant that weights would get updated regularly. The `MAX_TRIALS` hyperparameter had the biggest impact on the training time, so we set it to 300.

The training lasted for about 12 hours. After it was done we compiled a list of all 300 trained models and their different hyperparameter values, number of parameters and achieved accuracies. Part of it can be seen in Table 5.2.

After analysing results we came to several conclusions:

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

2. We could not see any visible correlation between a specific choice of a certain hyperparameter and accuracy. This showed that selection of hyperparameters is a non-heuristic task, at least for our particular problem.

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

4. The first 200 models covered an accuracy range of 0.62 %. However, inside of this range, the number of parameters varied hugely, for example, the model *192a* had more than 8 times fewer parameters than the model *96a*, although the difference in accuracy (0.27 %) was negligible.

Table 5.2: Partial results of the first random search of hyperparameters

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

It was apparent from the results that large models were not necessary to achieve high accuracy, so we decided to run the random search of hyperparameters again. This time we lowered the maximum and the minimum numbers of filters and size of the dense layer. We decreased all steps from 8 to 2, thus increasing the number of possible configurations. We decided to lower the bottom boundary of `DropoutRate` from 0.2 to 0.0, which meant that some models would not be using the dropout layer at all. We expected that training without the dropout layer would produce suboptimal results, however, we wanted to test it. The redefined search space for the second random search can be seen in Table 5.3 We increased the number of `MAX_TRIALS` from 300 to 500, as we were expecting that more models would end up underfitting, and also because there would be more possible options because of the smaller step size.

Partial results of the random hyperparameter search can be seen in Table 5.4.

Table 5.3: Second hyperparameter search space

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

Some observations:

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

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

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

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

Table 5.4: Partial results of the second random search of hyperparameters

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

5.1.2 Comparison of selected, re-trained models

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

For comparison and later on-device performance testing we decided to pick and retrain[1]6 models: *0a*, *2a*, *0b*, *172b*, *338b* and *460b*. Their properties are listed in Table 5.5.

The chosen models vary greatly in the number of parameters. Models *0a*, *2a*, *0b* have a high number of parameters, but their accuracy is high. Models *172b*, *338b* and

460b were chosen because of their small size and reasonably good accuracy.

Table 5.5: Properties of selected models

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

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

When analysing the performance of a model on an imbalanced dataset it is more appropriate to use precision and recall metrics. They give us a better idea of how well the model is performing on data of the specific classes. Precision tells us what percentage of data points in a specific predicted class fall into that class. Recall tells us what percentage of data points inside a certain class were actually predicted correctly. How they are calculated is shown in 5.1.

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ Recall &= \frac{TP}{TP + FN} \end{aligned} \tag{5.1}$$

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

Where:

TP - Number of data points that were true positive

FP - Number of data points that were false positive

FN - Number of data points that were false negative

Calculated metrics can be seen in Table 5.6; we abbreviated precision to P and recall to R for clarity. We also colour coded each Table row, where bright green shows the highest value in the row, red shows the lowest, light-green and orange colours show values in between.

Table 5.6: Precision and recall metrics of trained models

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

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

Another way to compare models' performance is to look at a confusion matrix. Figure 5.1 shows a comparison between the confusion matrices of an *0a* model on the left and *460b* model on the right. In the case of *0a*, 19 elephant images were not classified correctly, and 17 images were wrongly classified as elephants. This is not ideal, however, it is much better compared to the performance of *460b*, where 53 elephants were wrongly classified and 63 of images classified as elephants were not actually elephants.

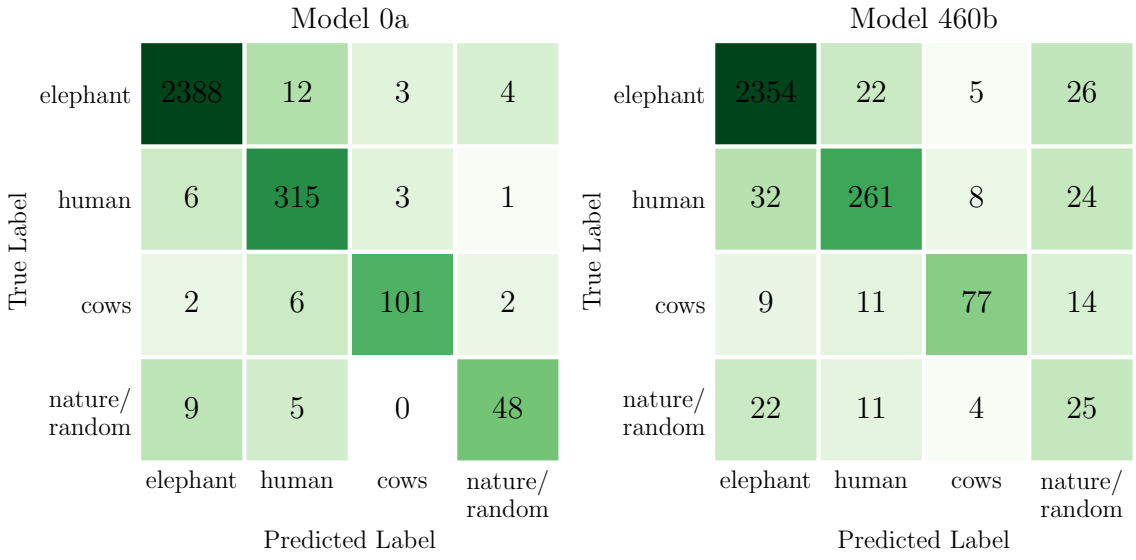


Figure 5.1: Confusion matrices of *0a* model (left) and *460b* model (right).

5.1.3 Comparison of Edge Impulse models

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

Studio. We also could not perform a random search of hyperparameters in the Edge Impulse Studio, as this feature was not fully supported at the time of writing this thesis.

We decided to train a few differently sized models, using the same general CNN architecture as before, but with some minor changes in hyperparameter values. We also trained a few models with the Transfer Learning technique. Edge Impulse offers scaled-down versions of pre-trained MobileNetV2[2]NN architecture, which we used.

Tables 5.7 and 5.8 show the properties of Edge Impulse models using CNN architecture and the Transfer Learning technique, respectively. Table 5.9 shows the calculated precision and recall values of Edge Impulse models using both approaches.

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

Table 5.7: Properties of Edge Impulse models using the CNN architecture.

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

[2]MobileNetV2 is a efficient, lightweight NN architecture, designed for image recognition tasks, suitable for mobile applications [14]. MobileNetV2 contains a width multiplier hyperparameter, which scales up or down the total number of parameters, thus providing a trade-off between accuracy and computation complexity. Edge Impulse offers three different width multiplier options: 0.35, 0.1 and 0.05.

Table 5.8: Properties of Edge Impulse models using the Transfer Learning technique.

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

Table 5.9: Precision and recall metrics of trained Edge Impulse models

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

Some observations:

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

5.2 On-device performance testing

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

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

Edge Impulse provides examples for testing out of the box, so not much work is needed to get the first-order approximation of performance. For profiling, the code execution Mbed API was used, which uses timer interrupts to track elapsed time. Figure 5.2 shows models ranked from the fastest to the slowest with marked corresponding accuracies and inference times.

We can see that all models performed inference in less than 1 second, which was a constraint that we set earlier in Section 3.1. The best time wise performing model was *338b*, with an inference time of 51 ms, but there are also many models that perform inference in under 300 ms.

We also discovered some unexpected trend in the results. We assumed that inference time is proportional to the number of parameters if the general structure of the model remains the same. As can be seen in Figure 5.2, there are few exceptions to this rule; model *338b* executed inference faster than *460b*, although it has more parameters (20K versus 13K). Model *172b* was slower than *0b*, although it has five times less parameters. This behaviour is not exclusive only to our models, but it can be seen in Edge Impulse models as well, for example, models *2ei* and *1ei*.

Edge Impulse models trained with the Transfer Learning technique *0tl*, *1tl*, *2tl* and

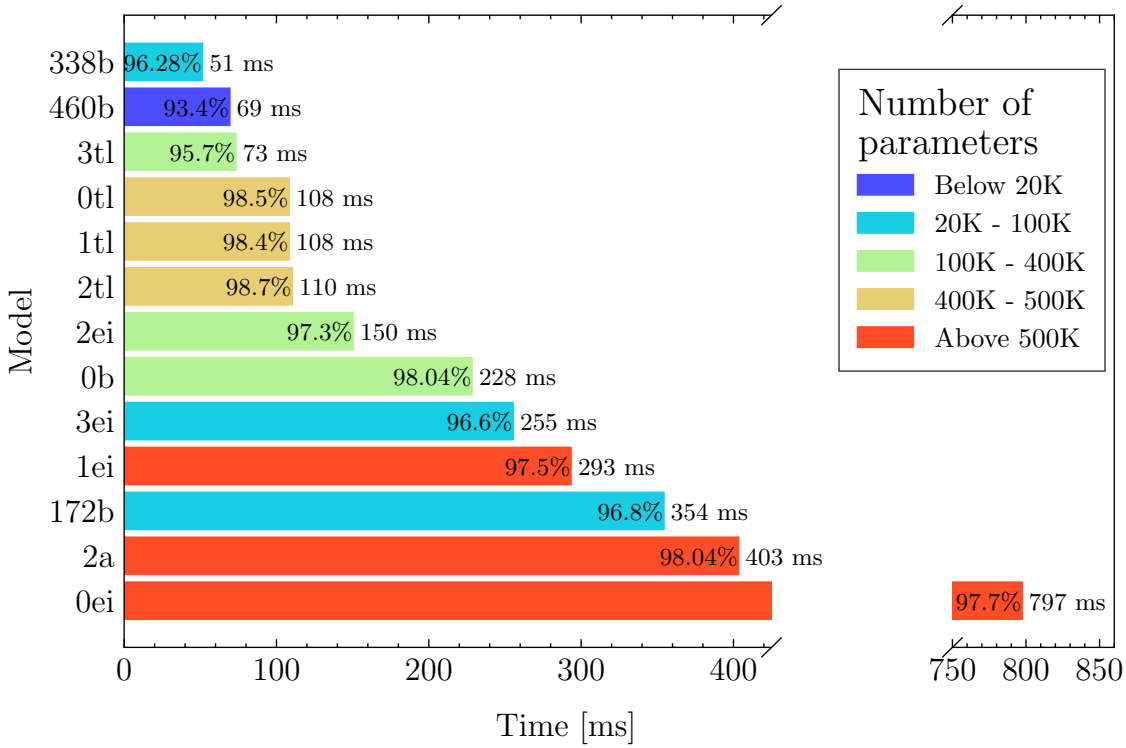


Figure 5.2: Comparison of time of inference of different models.

3tl should not be compared to other models in this sense, as the architecture of MobileNetV2 contains additional different operations.

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

5.2.1 Comparison of code sizes

We also wanted to inspect the Flash and RAM sizes of binaries that we compiled for on-device testing. For this task we used the `arm-none-eabi-size` command line tool, which returns sizes of `text`, `data`, `bss` sections in bytes, and an example of its output can be seen in Figure 5.1. To compute the used Flash we simply had add bytes from `text` and `data` sections, and to compute used RAM we added together `data` and `bss` sections[3].

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

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

Listing 5.1: Example output of arm-none-eabi-size command.

Code sizes for all models are presented in Figure 5.3, models are ordered the same way as they have been in Figure 5.2.

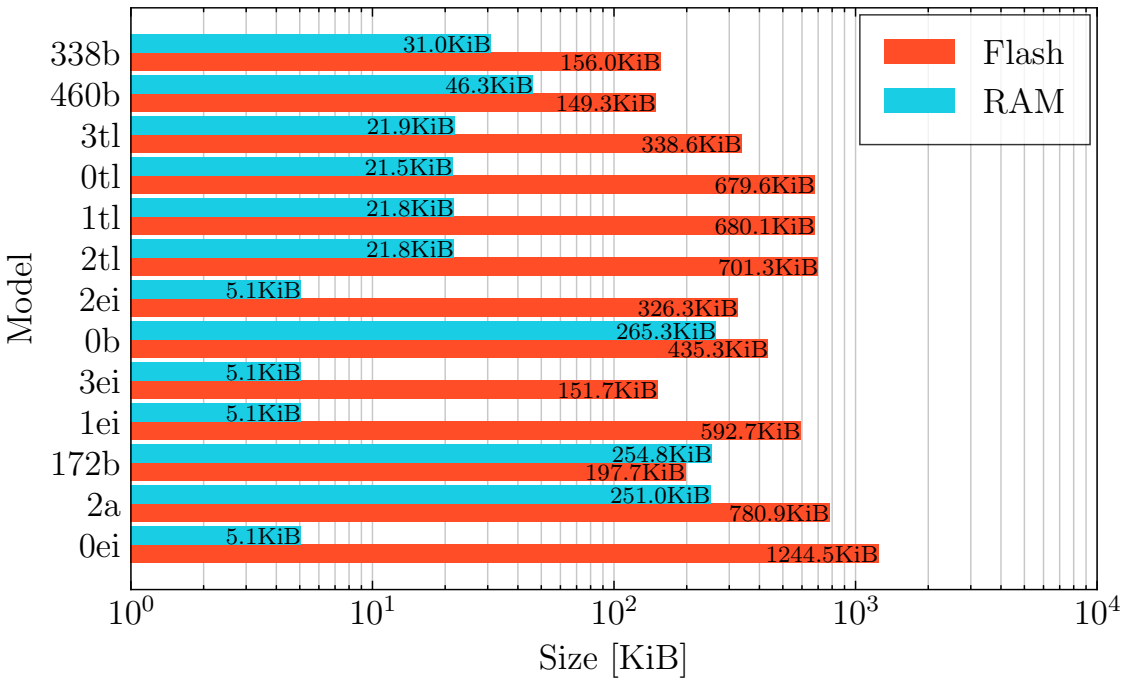


Figure 5.3: Comparison of Flash and RAM size of compiled example models.

We can see that all of our models generally use more RAM then Edge Impulse models. This is due to how the inference is executed. TFLite Micro uses a generic interpreter approach, where the model is loaded at runtime. Edge Impulse uses a compiled approach, which they named The EON™ Compiler [21]. The EON™ Compiler still uses TFLite Micro, however, it does not use its interpreter, but calls operation kernels directly. This means that the linker knows exactly which operations are used and more data can be moved into Flash, thus eliminating unneeded code size [21].

5.2.2 Comparison of different optimisations

Some extra amount of work and research was required to be able to run the ML inference at maximum possible efficiency under MicroML. Figure 5.4 shows the reductions in inference time of the *0b* model while using different optimisation methods.

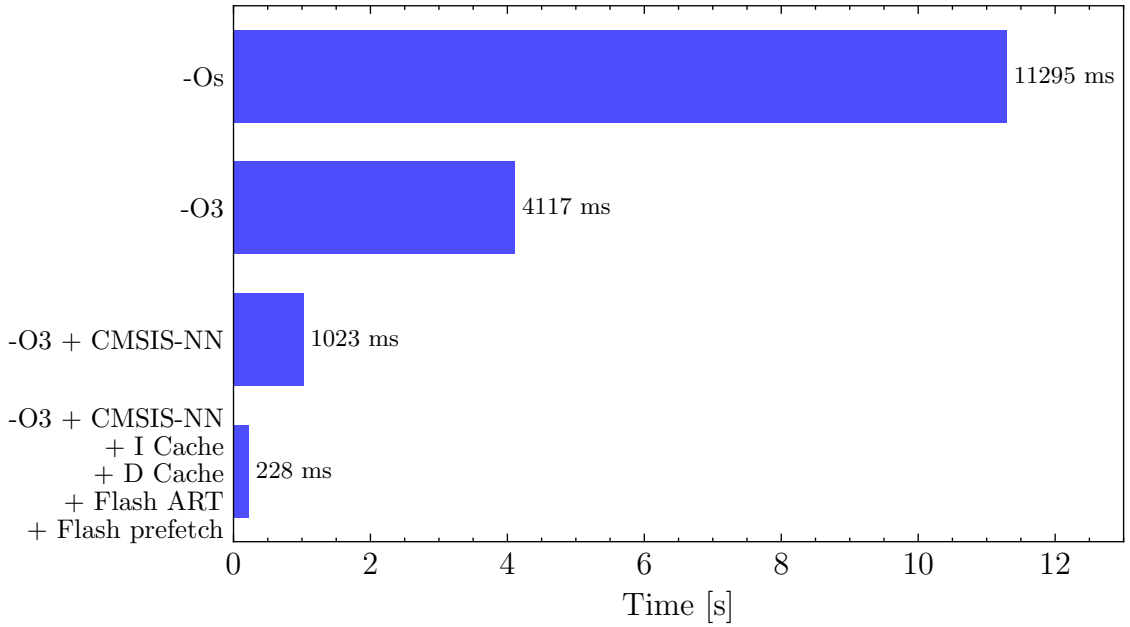


Figure 5.4: Inference time of the *0b* model using different optimisations.

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

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

Changing compiler optimisation flags could not lower the time of inference any further, so other approaches were needed. While reading through the TensorFlow documentation we saw that it supports the CMSIS-NN library for ARM microcontrollers. CMSIS-NN is a collection of efficient Neural Network kernels, that intends to maximise performance and lower code size of NN models implemented on ARM

microcontrollers. TensorFlow provides wrappers for some of these kernels, such as convolutional, fully connected, pooling layers and others. Not much work was needed to use these highly efficient kernels, as we only needed to specify in our Makefile that we wanted to compile CMSIS-NN kernels and not compile generic TensorFlow kernels. Time of inference dropped by about 3 seconds, down to 1023 ms.

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

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

5.2.3 Scoring trained models

Choosing the best model for on-field deployment is a hard task due to many different metrics: Precision and recall values, time of inference, and code size. To make this job easier we devised a scoring system: Each metric was going to be normalised and multiplied with some weight value. All products would then be summed up, and the result would represent the final score. The sum of the weights was equal to 100, which means that the possible maximum score was also 100. We decided to allocate 50 weight points to all precision and recall values, 30 points to the time of inference value, 5 points for Flash size and 15 points for RAM size. As we cared more about

the precision and recall values of the elephant, human and cow classes, we gave them 7 weight points each, while the nature/random class received only 4. We valued Flash size less than RAM, as most of the microcontrollers have much less RAM than they do Flash, thus, we gave 15 weight points to RAM size and only 5 to Flash size.

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

Scoring is described mathematically in 5.2, while the final results can be seen in Figure 5.5.

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

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

$$F(X, i) = \frac{(X[i] - MAX(X))}{MIN(X) - MAX(X)} \tag{5.2}$$

Where:

$Score$ - Vector of calculated scores

i - ith model

P_j - Vector of precision values of the jth class

R_j - Vector of recall values of the jth class

ToI - Vector of Time of Inference values

$Flash$ - Vector of Flash sizes

RAM - Vector of RAM sizes

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

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

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

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

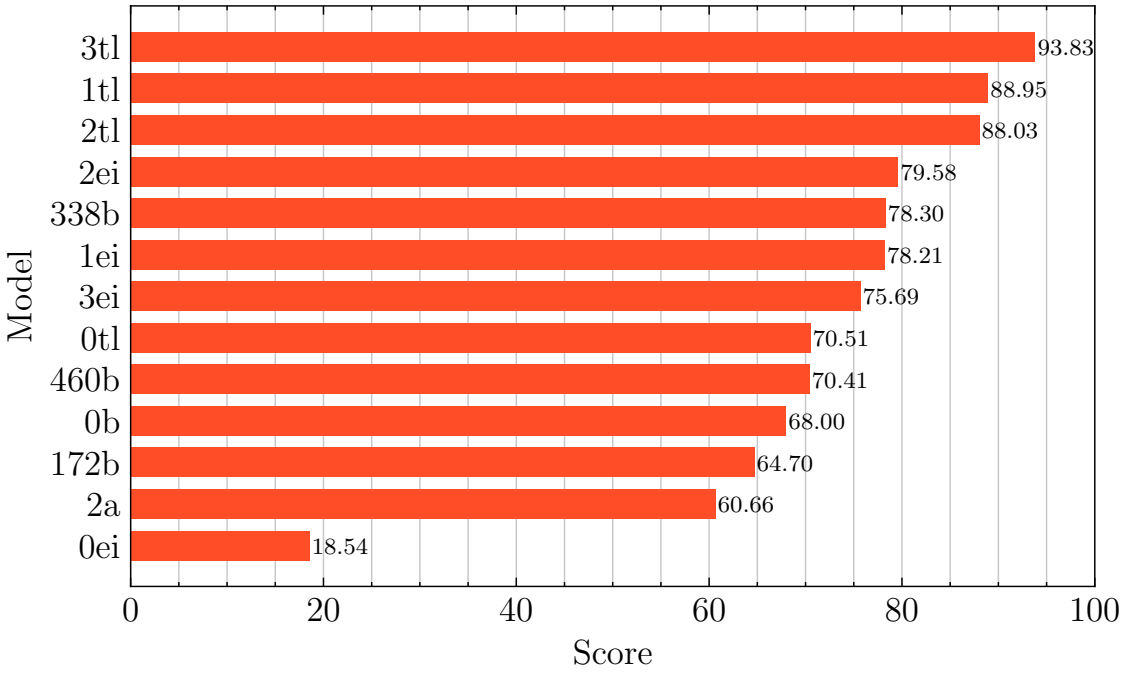


Figure 5.5: Score comparison of different models

5.3 Summary of model testing

As we saw in Figure 5.5 model *3tl* received the highest score, and models *1tl*, *2tl* followed. This should not be a surprise, because models trained with Transfer Learning achieved high accuracies and executed inference in about 100 ms. Additionally, the compiled approach for computing Neural Networks keeps the used Flash and RAM sizes to a minimum. We saw that using a number of different optimisations was critical to achieving low inference times, thus making ML on the embedded device viable.

5.4 Power profiling of an embedded early warning system

5.4.1 Measuring setup

To measure power consumption we used a product called Otii Arc (Otii), which can be seen in Figure 5.6. Otii, made by the company Qoitech, is a small portable box, that contains a power supply, a current and voltage measurement unit and a data acquisition module. It connects to a computer over a USB cable, and can be powered through it or with an external charger.

It can provide output voltage between 0.5 V and 4.55 V, and has accuracy of ±(0.1 % + 50 nA) when measuring current. It is a perfect tool for evaluating low-power systems.

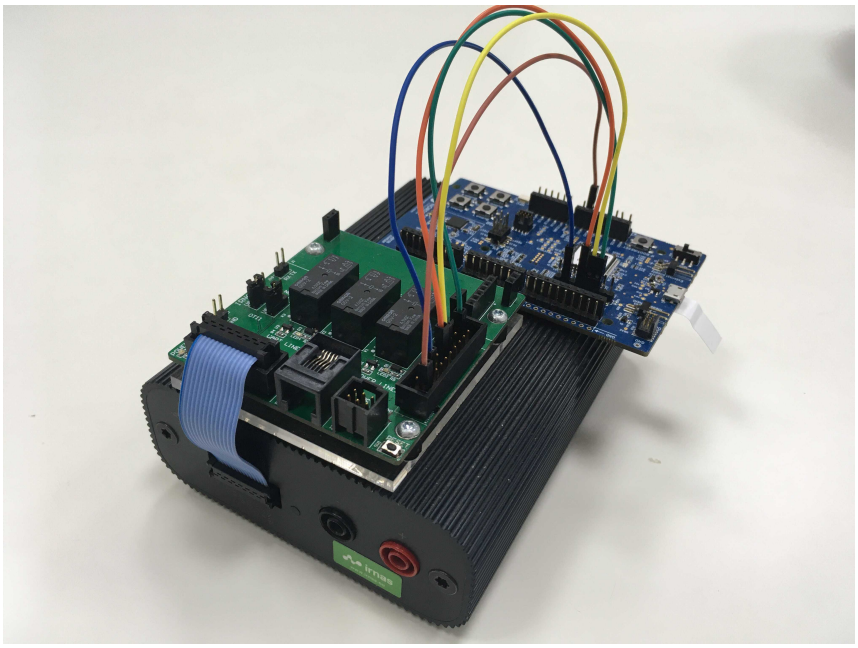


Figure 5.6: Otii Arc with nRF52832 DK and added measurement board made by IRNAS.

Measurements analysis is done with a desktop application, example of it is seen in Figure 5.7. The application enables users to select a part of the measurement, for which it computes minimum, maximum and average values automatically. To present our results we exported the current measurements in CVS format and plotted them

with Matplotlib.

In the following Section we evaluated the power consumption of our embedded early warning system. We first measured the power consumption of the nRF52 microcontroller in a low-power state, then we connected the LR1110 evaluation shield to the nRF52840 DK board and repeated the measurement. We then connected Nucelo-F767ZI and the FLIR camera, and measured power consumption of the whole detection sequence.

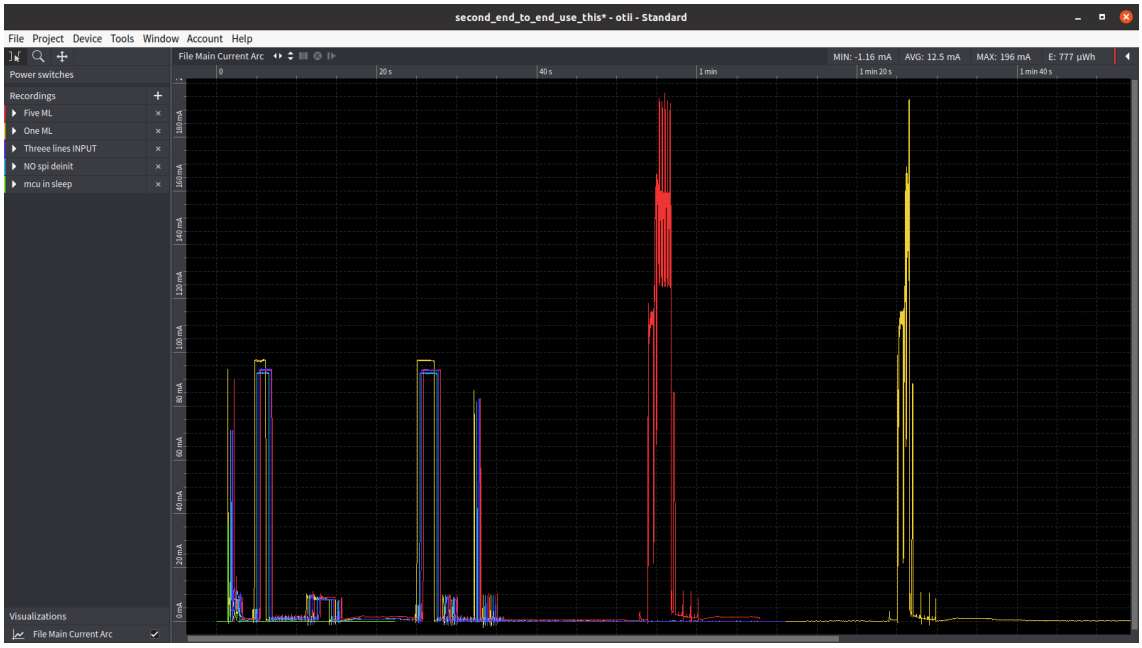


Figure 5.7: Screenshot of Otii user interface.

5.4.2 Current measurements

We conducted all our measurements with Otii's output voltage set to 3.3 V. Before measuring the current consumption of the whole image processing sequence we wanted to evaluate the current consumption of the nRF52 microcontroller in the low-power state. In a Zephyr kernel such procedure is relatively simple; type of sleep mode is configured with a Kconfig file and the microcontroller transitions into it whenever it enters the lowest idle thread. Peripherals require special attention, as it needs to be specified explicitly which one needs to be turned off. We turned off both of the UART peripherals and the SPI peripheral, while keeping GPIO active,

as we needed a GPIO interrupt to wake nRF52 up from a low-power state. We also had to make sure that the nRF52 microcontroller was completely disconnected from the on board J-Link debug probe to avoid any unnecessary current leaks. Luckily, the nRF52840 development board has an analogue switch, which does exactly that. To measure current consumption we simply connected the voltage output of Otii to the external power pins of the nRF52840 DK board. Figure 5.8 shows the current consumption in the low-power state.

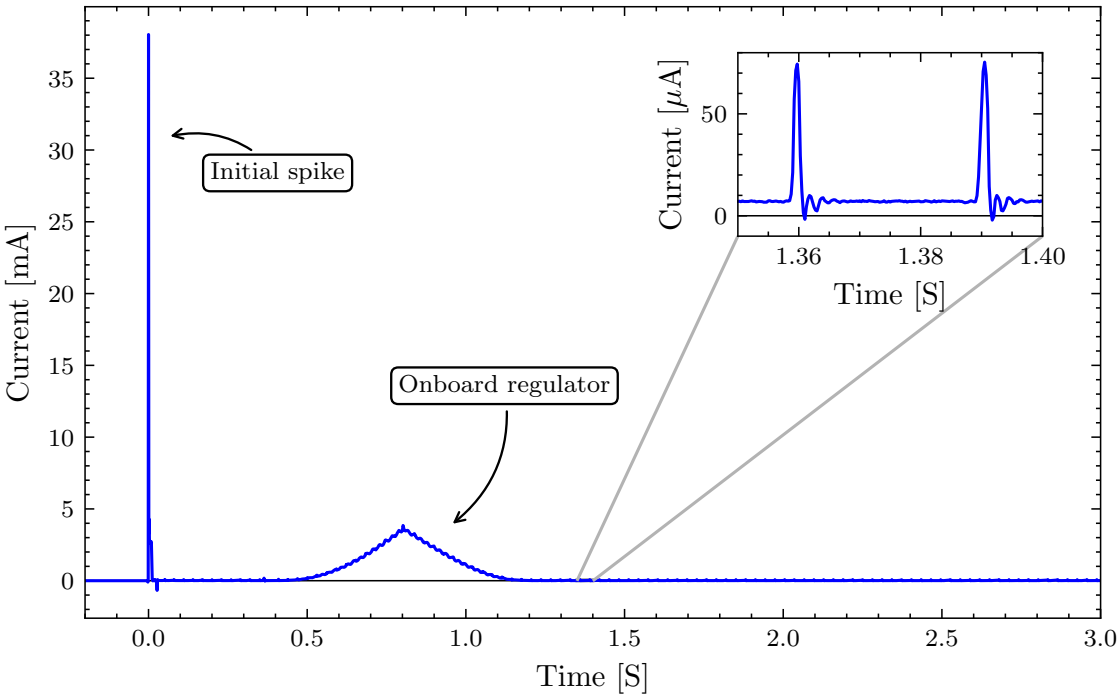


Figure 5.8: Current consumption of nRF52840 microcontroller in low-power state.

The initial spike at the beginning of the graph happens due to the many decoupling capacitors on the board. Due to the sudden change in voltage their impedance is low, therefore, more current is drawn. The pyramid looking shape between 0.5 and 1 second happens due to the onboard regulators turning on.

The smaller graph inside Figure 5.8 shows a close up view of the current consumption. The peaks reached 70.3 µA, steady state was at 6.9 µA, while the average current was 9.1 µA. Peaks were repeating at a frequency of 33.3 Hz.

The measured average current was higher than expected, according to the nRF52's datasheet [40] current consumption should be 3.16 µA.

In the next measurement we connected the LR1110 shield to the nRF52840 DK board, and observed the current consumption of the initial LoRaWAN join sequence. The current profile of it can be seen in Figure 5.9.

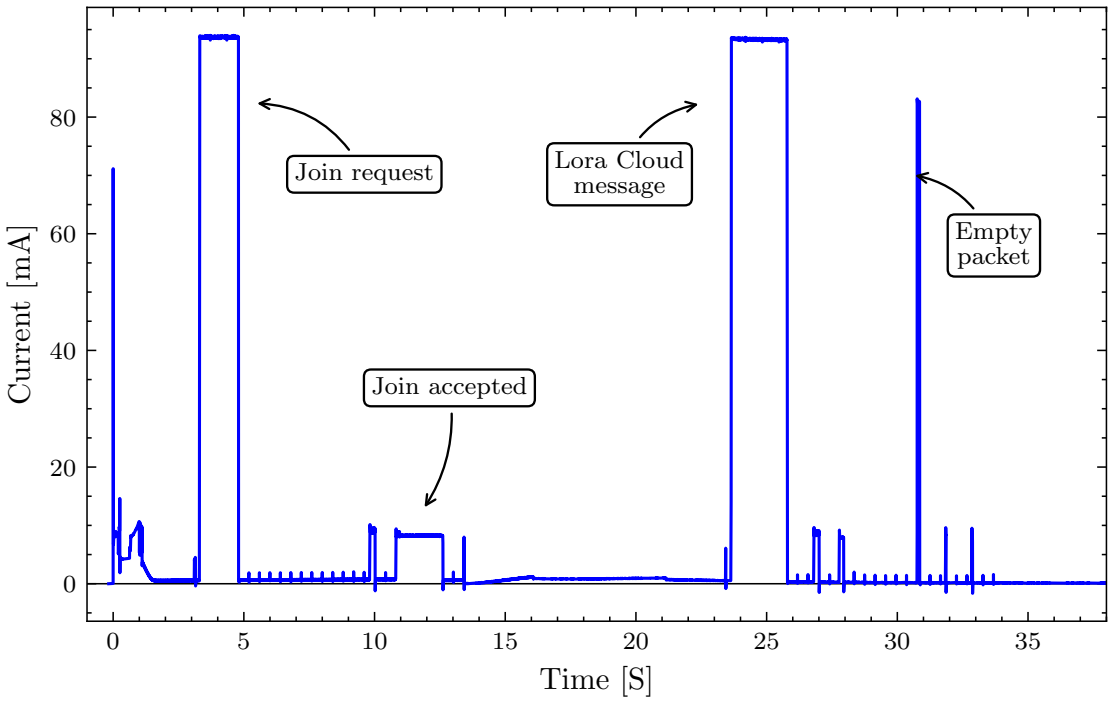


Figure 5.9: Current profile of the LoRaWAN join sequence.

Besides the initial spike, we can see an additional four pulses afterwards, with some smaller spikes in between. As we are using Over-The-Air Activation (OTAA), the LR1110 first has to negotiate for a set of keys with the server before it can start transmitting. This happens in first the two pulses; LR1110 first sends a join request and then listens for a response. After receiving a response it confirms it. In the third pulse LR1110 sends a message that is a part of the LoRa Cloud service. This message is specific to LR1110, and it cannot be disabled completely. The last, thin pulse belongs to an empty payload message, which LR1110 always transmits in the beginning. The average current consumption of the LoRaWan join procedure is 11.4 mA and lasts for about 34 seconds. Average current consumption in sleep state

increased to 76.8 µA.

In our next test we connected the nRF52 to a boost converter circuit, Nucleo-F767ZI and FLIR Lepton camera, setup can be seen in Figure 5.10. We did not use a PIR sensor as a wakeup source, as we saw that its detection was too sensitive to its surroundings and we could not control it completely. Instead we used a button on the nRF52840 DK as a wakeup interrupt. To account for PIR sensor current consumption in later calculations we measured it separately, and we saw that it drew 130 µA.

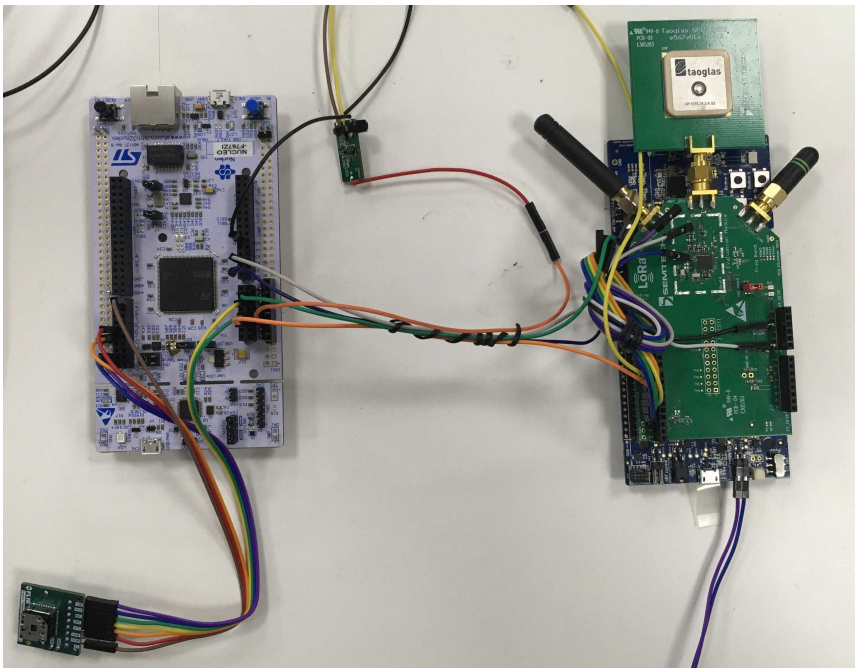


Figure 5.10: Device under test: nRF52840 DK with attached LR1110 shield, boost converter breakout board, Nucleo-F767ZI and FLIR Lepton camera.

Since we wrote our firmware with libopencm3 in mind, we could not use the best performing model *3tl*, as Edge Impulse models could only run on the Mbed platform. We used our model *460b* instead, as it was most similar to *3tl* in terms of inference time (69 ms compared to 73 ms). We captured two different inference procedures; in the first image capture and inference were done once, in the second one they were repeated 5 times. Procedures can be seen on Figures 5.11 and 5.12 respectively. Both procedures were followed by a LoRaWAN message that reported results to the server.

In the case where image capture and inferencing happened once, we measured the total time of the whole detection procedure to be about 1480 ms, not including the time needed for the LoRaWAN message. Average current consumption for this period was 114 mA. The measured average current of the whole event, shown in Figure 5.11 between 0 and 6 seconds, was 30 mA.

In the case where image capture and inferencing happened five times, we measured the total time of the whole detection procedure to be about 2,960 ms, not including the time needed for the LoRaWAN message. Average current consumption for this period was 131 mA. If we add the transmission of the LoRaWAN message to the measured current consumption, thus increasing the time of the whole detection event to 8 seconds, we measure 51.9 mA.

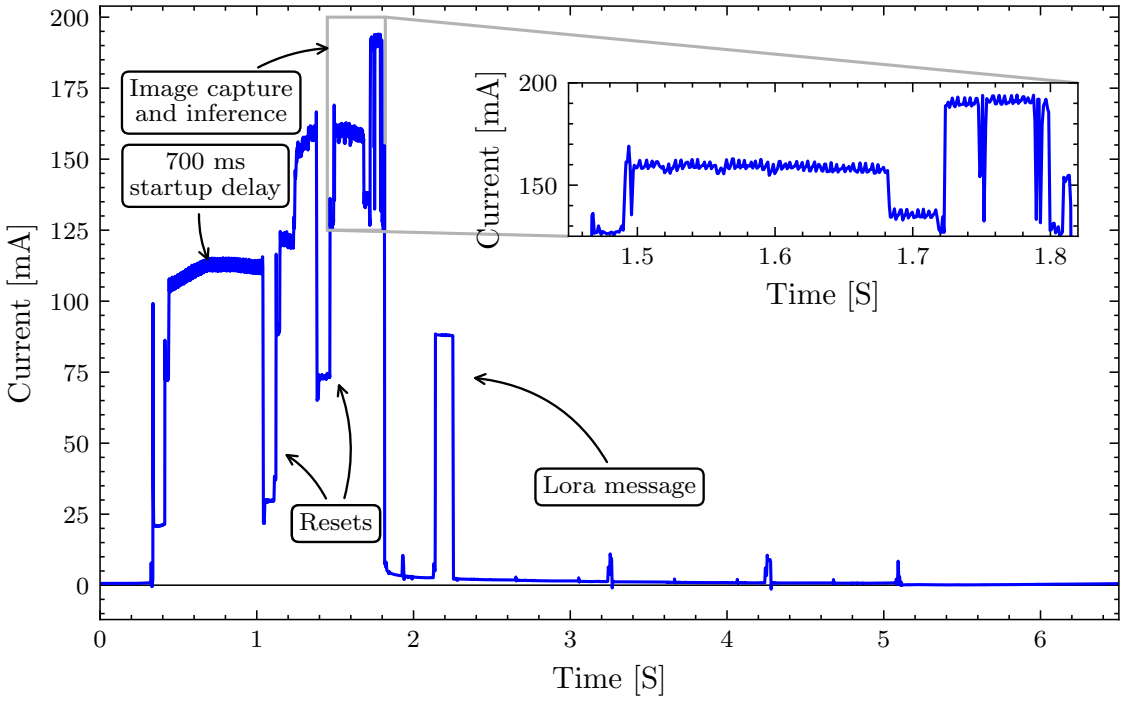


Figure 5.11: Current profile of image capture and inference procedure.

5.4.3 Commentary of the current measurement results

The measured low-power state of nRF52, visible in Figure 5.8, was higher than expected. nRF52's datasheet [40] specifies current consumptions in many different conditions, which depend on: Type of sleep mode (System ON or System OFF, the

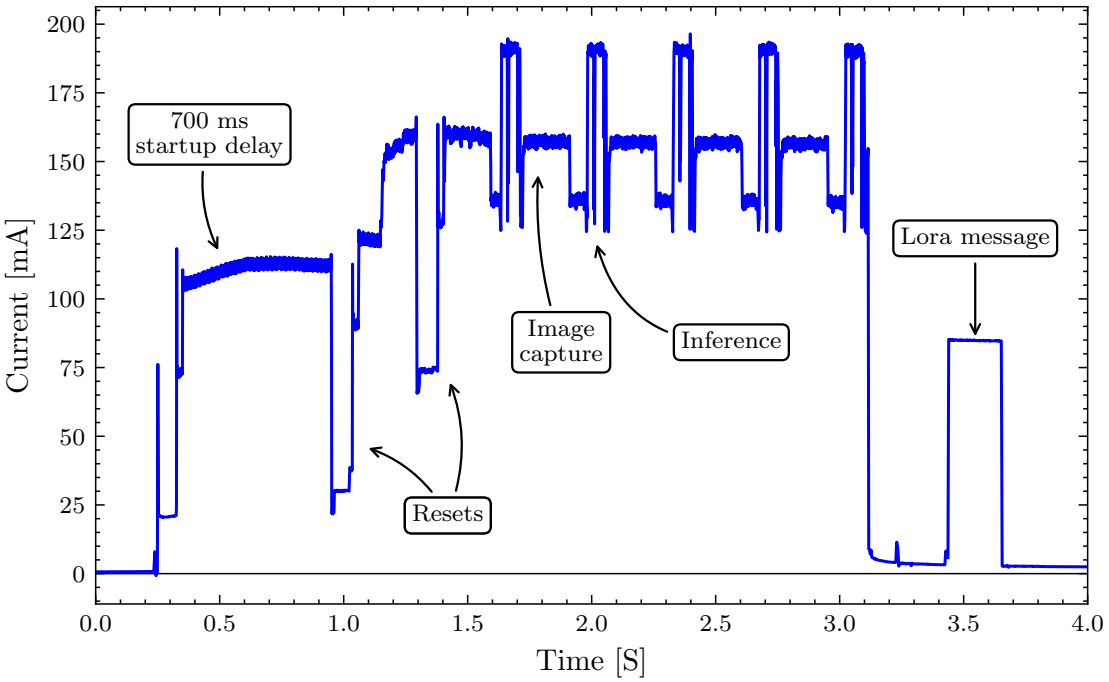


Figure 5.12: Current profile of image capture and inference procedure repeated 5 times.

latter loses execution context at wakeup), the amount of RAM retention and type of wakeup event. Consumption can range from 0.95 µA to 17.37 µA. Because Zephyr provides only abstract interface to power management, implementation of which is platform dependent, further research is needed to determine which exact nRF52 sleep mode Zephyr uses. We assumed an expected current consumption of 3.16 µA as this is the specified current consumption in System ON mode with full RAM retention and LFRC set as a wakeup event. Another reason for increased power consumption could also be inadequate support circuitry of the nRF52840 DK board.

When we connected the LR1110 shield to the nRF52840 DK board we saw that average current consumption increased to 76.8 µA, which was much more than expected. LR1110 enters sleep mode automatically when it is finished with communication; according to its datasheet [41], current consumption should be around 1.85 µA. This current consumption is plausible, as we observed it in other IRNAS's products that use the LR1110 chip. We suspect that the incorrect state of common GPIO connections between LR1110 and nRF52 was the reason for the increased current consumption, although, we were not able to fix the problem.

We mentioned that the time required for detection was about 1480ms, if we captured one image and processed it or 2960 ms, if done five times. We can see that both detection procedures started with 700 ms of delay. According to the FLIR Lepton's datasheet [37], a delay is required before we can start communicating with the camera over the TWI interface. Two microcontroller resets are visible, during our testing we saw that we could not communicate with the camera properly, if we did not reset the STM32 twice before that. To accomplish this we simply connected the reset pin of STM32 with one of available nRF52 GPIO pins.

5.4.4 Battery life estimations

Table 5.10 shows parameters values that we used in 5.3 to calculate battery lifetime. We defined a detection as a sequence where image capture and inference were repeated 5 times and then followed by a LoRaWAN message. In our calculation we also accounted for one daily LoRaWAN message, which would report system status. We assumed that the system is in low-power mode when it is not performing detection sequence or sending a LoRaWAN message. As a power source we chose a lithium-ion cell battery NCR 18650B of the manufacturer Panasonic. Its properties can be seen in Table 5.10.

Table 5.10: First hyperparameter search space

Event	**Current consumption at 3.3 V**
Low-power state	76.8 µA
PIR	130 µA
PIR and low-power state	206.8 µA
Detection sequence, 8 s	51.9 mA
LoRaWAN message, 230 ms	80 mA
Battery type	**Properties**
Panasonic NCR 18650B	Nominal Voltage: 3.6 V
	Nominal Capacity: 3350 mA h

$$
\begin{aligned}
P_{sleep} &= U_{supplied}\, I_{sleep} \\
P_{detect} &= U_{supplied}\, I_{detec} \\
P_{lora} &= U_{supplied}\, I_{lora} \\
t_{sleep} &= 24h - t_{detect}\, N_{detect} - t_{lora} \\
P_{average} &= \frac{P_{sleep}\, t_{sleep} + P_{detect}\, t_{detect}\, N_{detect} + P_{lora}\, t_{lora}}{24h} \\
t_{lifetime} &= \frac{U_{bat}\, Ah_{bat}\, N_{bat}}{P_{average}}
\end{aligned}
\tag{5.3}
$$

Where:

$U_{supplied}$ - Voltage at which we did our measurements, 3.3 V

U_{bat} - Battery nominal voltage, 3.6 V

Ah_{bat} - Battery nominal capacity, 3350 mA h

I_{lora} - Current consumption when sending a LoRaWAN message

I_{sleep} - Current consumption of low-power state with PIR consumption

I_{detect} - Current consumption of detection sequence

P_{sleep} - Used power of low-power state and PIR

P_{detect} - Used power of detection sequence

P_{lora} - Used power of LoRaWAN message send

$P_{average}$ - Average power needed for the system to operate

t_{detect} - Time spent in detection sequence in hours

t_{sleep} - Time spent in low-power state in hours

t_{lora} - Time spent sending LoRaWAN messages

$t_{lifetime}$ - Battery life time in hours

N_{detec} - Number of detections in a day

N_{bat} - Number of batteries

Because we could only assume how frequently our system had to perform the detection sequence, we repeated our calculation for several different numbers of detections per day. We also varied the number of battery cells. The enclosure that we planned to use had enough space for up to 6 battery cells. The results of the estimations can be

seen in Figure 5.13. The X axis represents the number of battery cells, the Y axis represents the system life time in months, and the colour of the lines represents the number of detections per day, from 100 to 600.

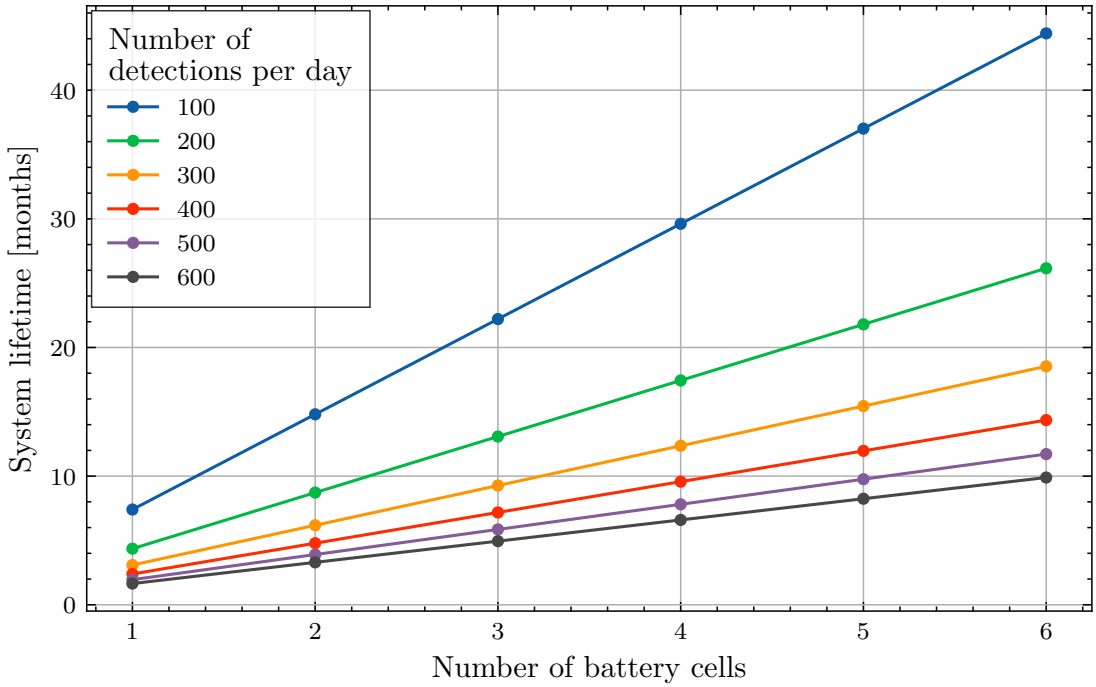


Figure 5.13: Current profile of image capture and inference procedure.

Results were promising, we can see that most battery configurations lasted more than 6 months, which is a long time, considering the application. If we decide on a 6 battery cell configuration, we see that in the worst case where we are executing 600 detections per day, we can expect a system lifetime of 10 months, which is more than enough. There are some things that we have to take into account. We assumed 100 % battery efficiency, meaning that each battery would provide its complete nominal capacity, in our case 3350 mA h. In practice this is not possible, due to effects such as self-discharge rate, high discharge profiles and temperature influences. On the other hand, we assumed that we would process same amount of detections everyday, which was in the worst case 600. This is unlikely to happen and 600 detection per day is quite high so we expected the actual system lifetime to be longer. Such effects and conditions can drastically change the lifetime of the system and can be hard to estimate.

While calculating the system lifetime, we tried changing the input parameters to see which ones affected the final lifetime of the system the most. We found out that, by decreasing current consumption in the low-power state by a factor of ten (from 206.8 µA to 21 µA), we did not increase the system's lifetime dramatically. With 6 batteries and 600 detections per day the lifetime increased to 10.5 months from 9.9 months. However, we saw a large increase in the system's lifetime if we halved either the current consumption of a detection event or its duration, with same conditions as before the system's lifetime increased from 9.9 to 18.5 months. This means that in order to increase the system's lifetime or to keep it the same with a smaller number of batteries, we should focus on lowering the power consumption of detection events rather than lowering the low-power state current.

6 Conclusion

In this Master's thesis we presented a solution, an early warning system, for minimising Human-Elephant Conflicts. In the beginning we presented a Machine Learning approach for solving classification of thermal images and we outlined its strengths compared to classical problem solving. We described in depth the knowledge required for understanding our work from various sub parts of Machine Learning, to wireless IoT technologies and thermal cameras.

We presented several Machine Learning workflows that we followed during the design and planning of our own Neural Network. We analysed the thermal image dataset and recognized what kind of data it lacked. We compensated for missing data by gathering more of them with our own image capturing setup and by resampling them. We proposed a basic CNN structure that we used later in the model training phase.

As we wanted to execute Machine Learning algorithms on our selected STM32 microcontroller, we had to find a way to make this possible. We ported a TensorFlow Lite for a Microcontrollers library to the libopencm3 platform and in the process created an open-source project MicroML. In the process we familiarised ourselves with the specifics of cross compiling and build systems.

To implement our early warning system on the hardware we decided to use a two microcontroller setup, with an STM32 as a main inferencing processor and nRF52 as a low-power system controller. To communicate with the FLIR Lepton camera we wrote the driver code, and we also wrote the LoRaWAN communication module.

We ran random search algorithms to find the proper hyperparameters for our proposed CNN model. We did an in depth comparison of several trained models and compared

them to the models trained with the help of commercial software.

We saw that we could train CNN models with an accuracy as high as 98.04 % and we saw that models trained with the Transfer Learning technique reached 98.7 % without much additional work. We ran each trained model on the STM32 and measured its inference time. From the results we saw that reaching an inference time below 200 ms is not hard if special care is devoted to the low number of parameters and correct microcontroller optimisations.

We also evaluated the performance of the embedded system from the battery lifetime perspective and we saw promising results. Assuming that our system would have to process 600 events per day, while connected from 6 battery cells, we estimated the lifetime of the system would be around 10 months.

This thesis shows that the field of Machine Learning on embedded devices has reached a point were it is viable enough to use it in real life applications. Running inference directly on the embedded device can provide instant feedback and can extend the device's lifetime, as data do not need to be sent to the server for processing. Many use cases from the Animal Conservation field would benefit from such technology, as there is a need for the embedded devices that require minimal manual care and can provide big value for their cost.

6.1 Future work

Our research into elephant detection with the aid of Machine Learning models yielded promising results. There are several aspects of it that can be improved.

In terms of model performance, we can always improve it by gathering more relevant training data. The dataset that we used contained several thousand images of elephants, but only a couple of thousand images of humans and only a few hundred images of cows. In order to train a more robust and reliable model, we should gather more thermal images of humans and livestock, especially goats.

It would be interesting to further explore models trained with the Transfer Learning technique. We saw that Transfer Learning models reached higher accuracies with

shorter inferencing times when compared to other models. We expect that running a random hyperparameter search with a smaller version of the pre-trained MobileNetV2 model could produce optimal results.

In terms of the system performance, testing our early warning system in the field would give us key insights into what could be improved. With a device deployed in a zoo, we could monitor its performance and see which conditions degrade its performance. We could add an SD card to the system, and save every taken image and the result of its inference.

By observing the performance of the model in the field we would see if extremely low inference times are really needed. It might be feasible to run CNN models on slower, low-power, Cortex-M4 microcontrollers. Although we are expecting longer inference times, we would benefit from a simpler system design and a lower overall price of the embedded system.

In terms of battery life performance, creating a custom printed circuit board specially for our application would give us more control over the current consumption of the systems. Reaching lower low-power state current consumption should be easier. At the same time, we should research how to optimise the detection sequence. Decrease in either current consumption or the duration of detection sequence would provide us with huge improvements in battery life.

6.2 Final words

Machine Learning on the embedded devices is opening doors to various, wonderful applications that were not possible a few years ago. We can see that there is already a demand for intelligence for devices on the "edge" and we expect it to increase as the field matures. It is quite possible that future embedded engineers would require some amount of Machine Learning knowledge to stay competitive and interesting to the market.

As Pete Warden said: "The Future of ML is Tiny and Bright".

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

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

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

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

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

[26] 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].

[27] 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.

[28] 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].

[29] 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].

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

[32] 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.

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

[34] 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].

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

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

[37] FLIR, Lepton Engineering Datasheet. Available on: https://flir.netx.net/file/asset/12411/original/attachment, [29.11.2020].

[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].

[40] Nordic Semiconductor, nRF52840 Product Specification. Available on: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.1.pdf, [28.11.2020].

[41] Semtech, LR1110 Datasheet. Available on: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R000000Q2YY/7hyal8gUewWREN8DSEk5R3Ee8OpqEuVOdsHpiAHb3jo, [29.11.2020].