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, May 2020

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, May 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 program:	Second Cycle Bologna Study Programme of Electrical Engineering
Module of study program:	Electronics
Mentor:	doc. dr. Iztok Kramberger univ. dipl. inž. el.
Co-mentor:	Mag. nekaj nekaj Vojislav Dragan Milivojević

Zahvala

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

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

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

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

Acknowledgement

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

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

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

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

Energetsko učinkovit sistem za detekcijo slonov s pomočjo strojnega učenja

Ključne besede: strojno učenje, mikrokrmilnik, sklepanje na napravi, termalna kamera, sistem z majhno porabo, **UKD:** XXXXX

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 strukturo modela. 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. 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

TODO

6. Povzetek

TODO

Energy efficient system for detection of elephants with Machine Learning

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

UKD: XXXXX

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 it. 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 TensorFlow library and created our own build system, suitable for libopencm3 platform. We also implemented reporting of inference results over LoRaWAN network and described possible server-size solution. We finally 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
AGC	Automatic Gain Control
SYS	System Information
VID	Video Processing Control
OEM	Original Equipment Manufacturer
RAD	Radiometry
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 Measurements and results

1.1 Comparison of models

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

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

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

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 in Edge Impulse Studio, former with a setup comparable to the ours, latter with Transfer Learning technique. Tables that are shown below list one of the metrics as accuracy. With accuracy we mean validation accuracy, it tells us how well model performed on a validation dataset.

1.1.1 Hyperparameter search space and results analysis

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

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

Possible values of hyperparameters (also known as hyperparameter search space) are specified in Table 1.1.

Table 1.1: First hyperparameter search space

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

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

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

The training lasted for about 12 hours. After it was done we compiled a list of all 300 trained models and their different hyperparameter values, number of parameters and achieved accuracies Part of it can be seen in Table 1.2.

After analyzing results we came to several conclusions:

1. We saw that almost all trained models, except for the last one, achieved accuracy above 90 %. This proved that the general architecture of the model was appropriate for the problem.
2. We could not see any visible correlation between a specific choice of a certain hyperparameter and accuracy. This showed that selection of hyperparameters is a non-heuristic task, at least for our particular problem.
3. Filter of size 3 x 4 did not perform significantly better compared to one with size 3 x 3.
4. There is a weak correlation between the number of parameters (model's complexity) and accuracy, however, models with small size and great accuracy exist, model *2a* is an example of that.
5. First 200 models cover an accuracy range of 0.62 %. However inside of this

Table 1.2: Partial results of first random search of hyperparameters

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

range model number of parameters varies hugely, for example, model *192a* has more than 8 times fewer parameters than model *96a*, although the difference in accuracy (0.27 %) is negligible.

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

This time we lowered the maximum and the minimum numbers of filters and size of the dense layer. We decreased all steps from 8 to 2, thus increasing the number of possible configurations. We decided to lower the bottom boundary of `DropoutRate` from 0.2 to 0.0, which means that some models will not be using dropout layer at all. We expected that training without dropout layer would produce suboptimal results, however, we wanted to test it. Redefined search space for second random search can be seen in Table 1.3 We increased the number of MAX_TRIALS from 300 to 500, as we were expecting that more models will end up underfitting and also because there would be more possible options because of smaller step size. Partial table of results of random hyperparameter search can be seen in Table 1.4.

Table 1.3: Second hyperparameter search space

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

Table 1.4: Partial results of second random search of hyperparameters

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

Some observations:

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

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

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

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

1.1.2 Comparison of selected, re-trained models

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

For comparison and later on device performance testing we decided to pick and retrain[1]6 models: *0a*, *2a*, *0b*, *172b*, *338b* and *460b*, their properties are listed in Table 1.5.

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

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

Table 1.5: Properties of selected models

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

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

When analysing the performance of a model on an imbalanced dataset it is more appropriate to use precision and recall metrics[2]. They can give us a better idea of how well the model is performing on data of specific classes. Calculated metrics can be seen in Table 1.6, we abbreviated precision to P and recall to R for clarity. We also colour coded each table row, bright green shows the highest value in the row, red shows the lowest, light-green and orange colours show values in between.

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

[2]Precision tells us what percentage of data points in a specific predicted class fall into that class. Recall tells us what percentage of data points inside a certain class were actually predicted correctly [1].

Table 1.6: 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

Another way to compare models performance is to look at a confusion matrix. Figure 1.1 shows comparison between confusion matrices of *0a* model on the left and *460b* model on the right. In the case of *0a*, 19 elephant images were not classified correctly, and 17 images were wrongly classified as elephants. This is not ideal, however is much better compared to performance of *460b*, where 53 elephants were wrongly classified and 63 of images classified as elephants were not actually elephants.

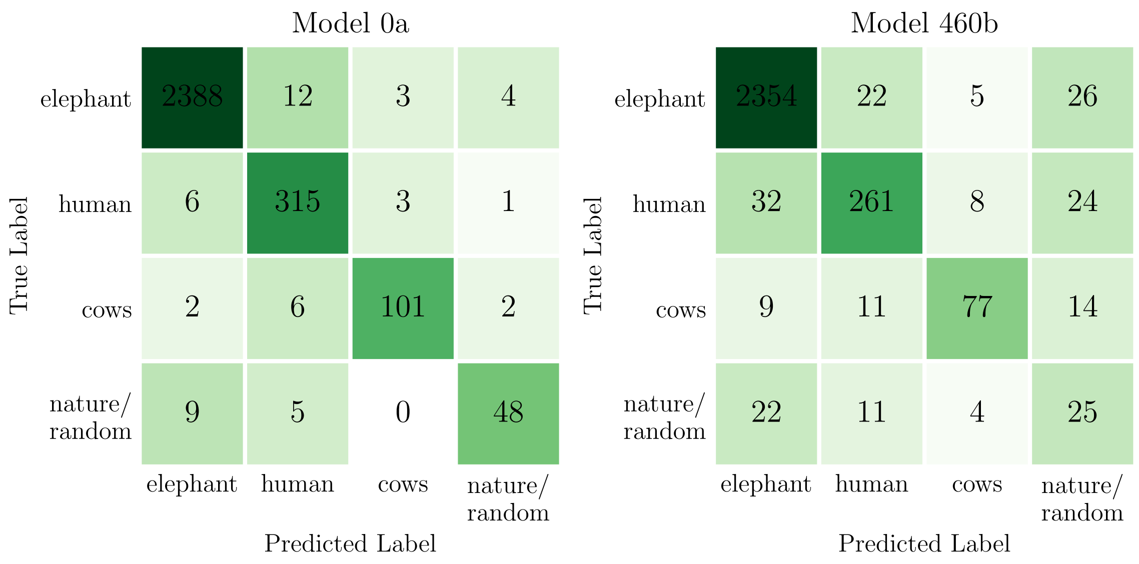

Figure 1.1: Confusion matrices of *0a* model (left) and *460b* model (right).

1.1.3 Comparison of Edge Impulse models

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

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

Tables 1.7 and 1.8 show properties of Edge Impulse models using CNN architecture and Transfer Learning technique, respectively. Table 1.9 shows calculated precision and recall values of Edge Impulse models using both approaches.

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

[3]MobileNetV2 is a efficient, lightweight NN architecture, designed for image recognition tasks, suitable for mobile applications [1]. MobileNetV2 contains width multiplier hyperparameter, which scales up or down the total number of parameters, thus providing a trade-off between accuracy and computation complexity. Edge Impulse offers three different width multiplier options: 0.35, 0.1 and 0.05.

Table 1.7: Properties of Edge Impulse models using CNN architecture.

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

Table 1.8: Properties of Edge Impulse models using Transfer Learning technique

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

Table 1.9: Precision and recall metrics of trained Edge Impulse models

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

Some observations:

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

1.2 On-device performance testing

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

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

Edge Impulse provides examples for testing out of the box, so not much work is needed to get the first-order approximation of performance. For profiling, the code execution Mbed API was used, which uses timer interrupts to track elapsed time. Figure 1.2 shows models ranked from the fastest to the slowest with marked corresponding accuracies and inference times.

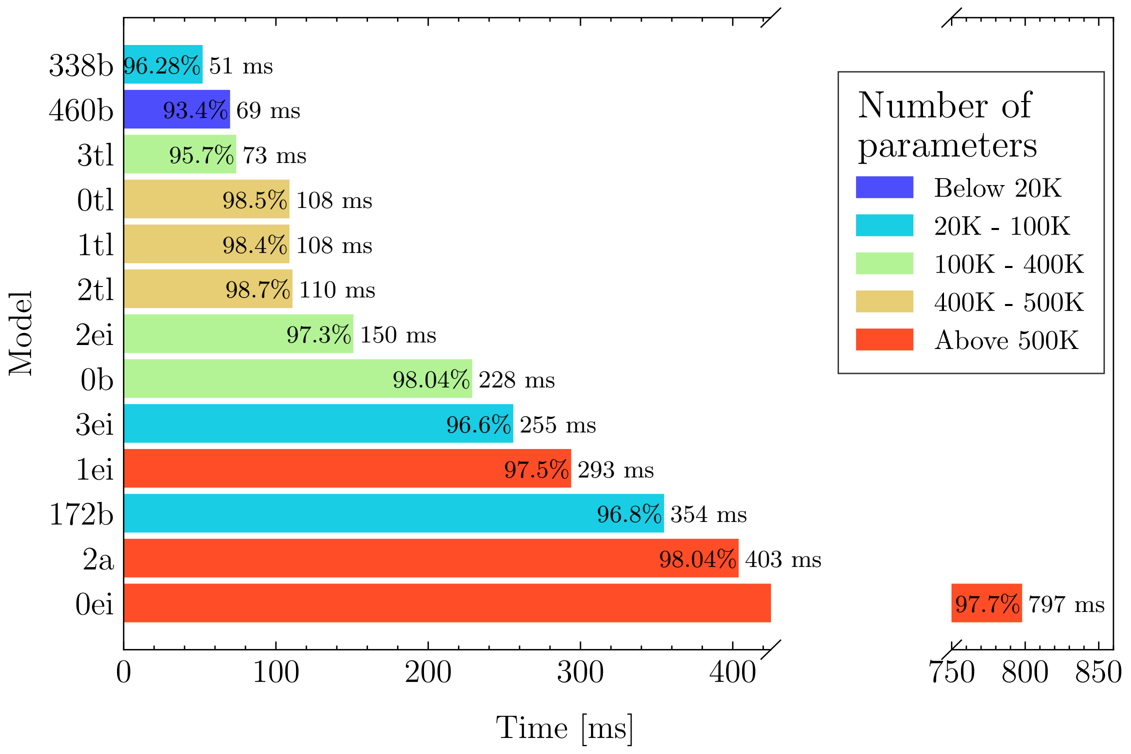

Figure 1.2: Comparison of time of inference of different models.

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

We also discovered some unexpected trend in results. We assumed that inference time is proportional to the number of parameters if the general structure of the model remains the same. As can be seen in Figure 1.2 there are few exceptions to this rule, model *338b* executed inference faster than *460b* although it has more parameters (20K versus 13K). Model *172b* was slower than *0b*, although it has five times less parameters. This behaviour is not exclusive only to our models, but it can be seen in Edge Impulse models as well, for example, models *2ei* and *1ei*.

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

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

1.2.1 Comparison of code sizes

We also wanted to inspect Flash and RAM sizes of binaries, that we compiled for on-device testing. For this task we used `arm-none-eabi-size` command line tool which returns sizes of `text`, `data`, `bss` sections in bytes, example of output can be seen in Figure 1.1. To compute used Flash we need simply add bytes from `text` and `data` sections, and to compute used RAM we add together `data` and `bss` sections[4].

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

Listing 1.1: Example output of arm-none-eabi-size command.

Code sizes for all models are presented in Figure 1.3, models are ordered the same way as they have been in Figure 1.2.

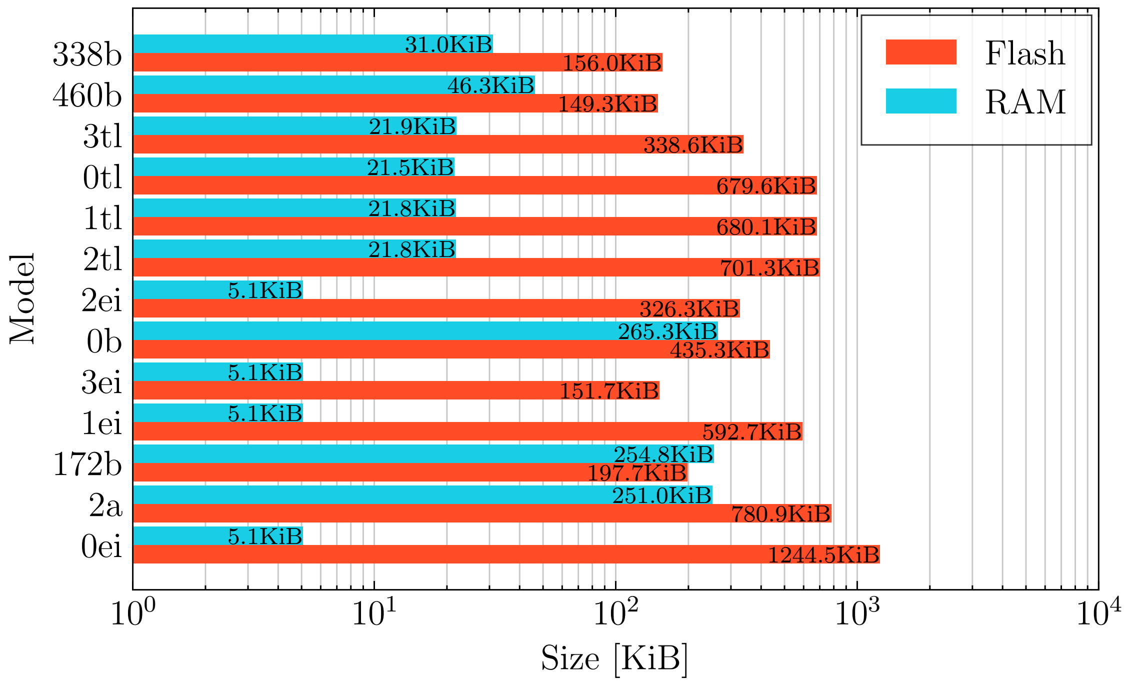

Figure 1.3: Comparison of Flash and RAM size of compiled example models.

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

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 [2]. The EON™ Compiler still uses TFLite Micro, however, it does not use its interpreter, but directly calls operation kernels. This means that linker knows exactly which operations are used and more data can be moved into Flash, thus eliminating unneeded code size [2].

1.2.2 Comparison of different optimisations

To be able to run the ML inference at maximum possible efficiency under MicroML some extra amount of work and research was required. Figure 1.4 shows reductions in inference time of *0b* model while using different optimisation methods.

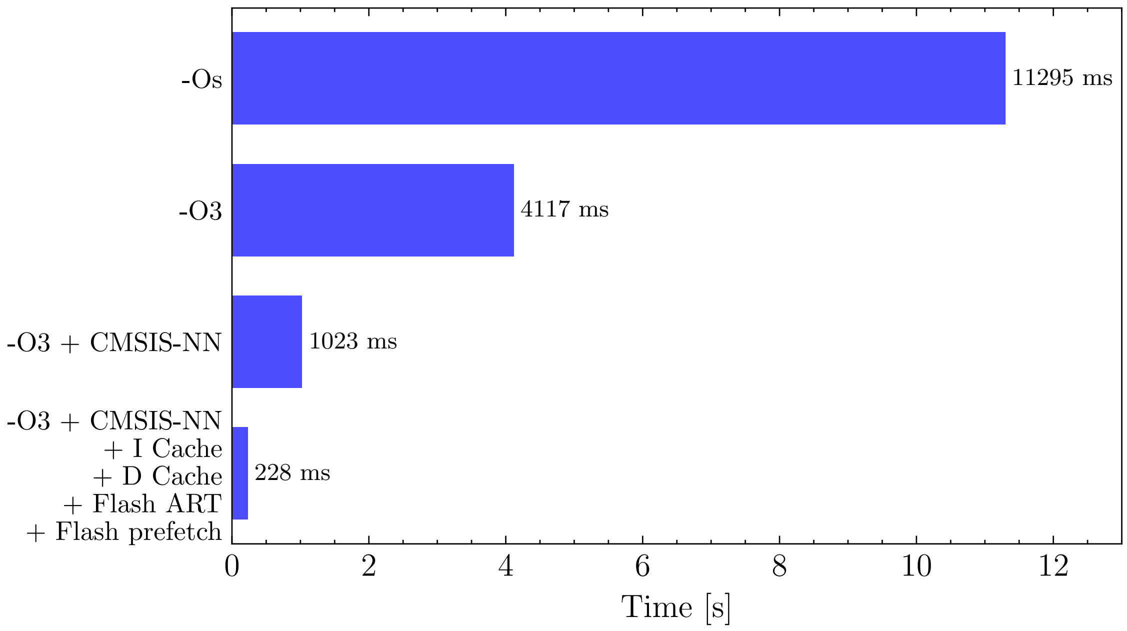

Figure 1.4: Inference time of *0b* model using different optimisations.

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

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

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

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

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

1.2.3 Scoring trained models

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

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

Scoring is described mathematically in 1.1, while the final results can be seen in Figure 1.5

$$\begin{aligned} Score[i] &= 7K(P_{elephant}, i) + 7K(P_{human}, i) + 7K(P_{human}, i) + 4K(P_{ntr/rnd}, i) \\ &\quad + 7K(R_{elephant}, i) + 7K(R_{human}, i) + 7K(R_{human}, i) + 4K(R_{ntr/rnd}, i) \\ &\quad + 30\,F(ToI, i) + 5\,F(Flash, i) + 15\,F(RAM, i) \\ K(X, i) &= \frac{(X[i] - MIN(X))}{MAX(X) - MIN(X)} \\ F(X, i) &= \frac{(X[i] - MAX(X))}{MIN(X) - MAX(X)} \end{aligned} \tag{1.1}$$

Where:

$Score$ - Vector of calculated scores

i - ith model

P_j - Vector of precision values of jth class

R_j - Vector of recall values of jth class

ToI - Vector of Time of Inference values

$Flash$ - Vector of Flash sizes

RAM - Vector of RAM sizes

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

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

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

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

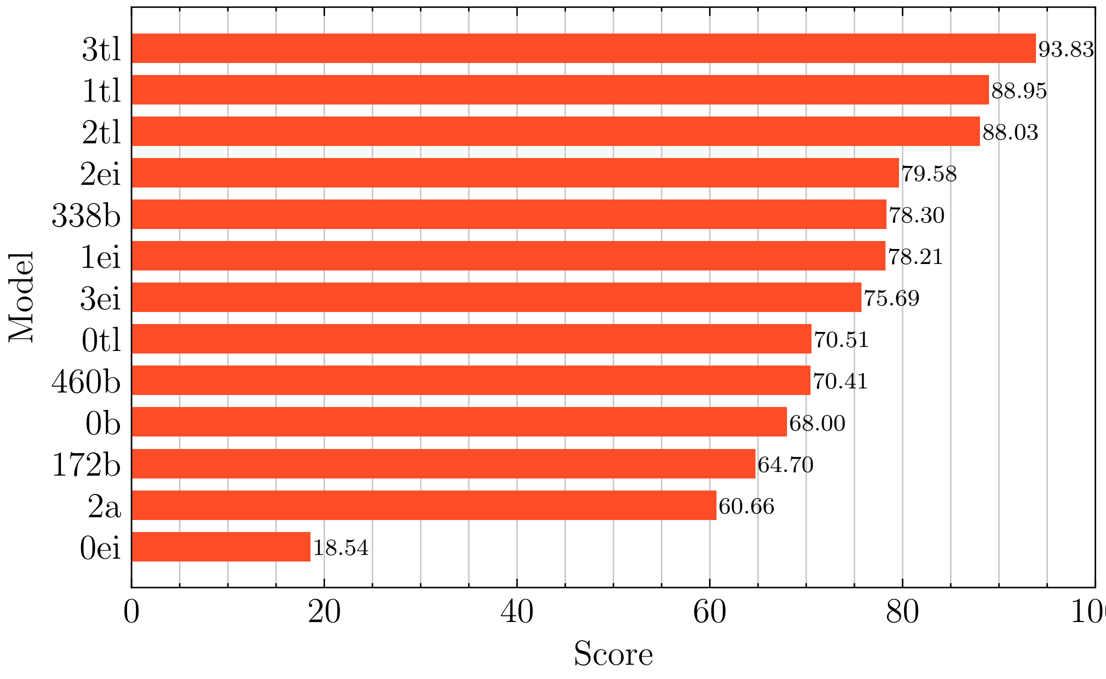

Figure 1.5: Score comparison of different models

1.3 Summary of model testing

As we saw in Figure 1.5 model *3tl* received the highest score, models *1tl*, *2tl* follow. This should not be a surprise, models trained with Transfer Learning achieved high accuracies and executed inference in about 100 ms. Additionally, compiled approach for computing Neural Networks keeps used Flash and RAM sizes to a minimum. We saw that using a number of different optimisations is critical to achieving low inference times, thus making ML on embedded device viable.

1.4 Power profiling of an embedded early warning system

1.4.1 Measuring setup

To measure power consumption we used product called Otii Arc (Otii), which can be seen in Figure 1.6. Otii, made by a 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 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.

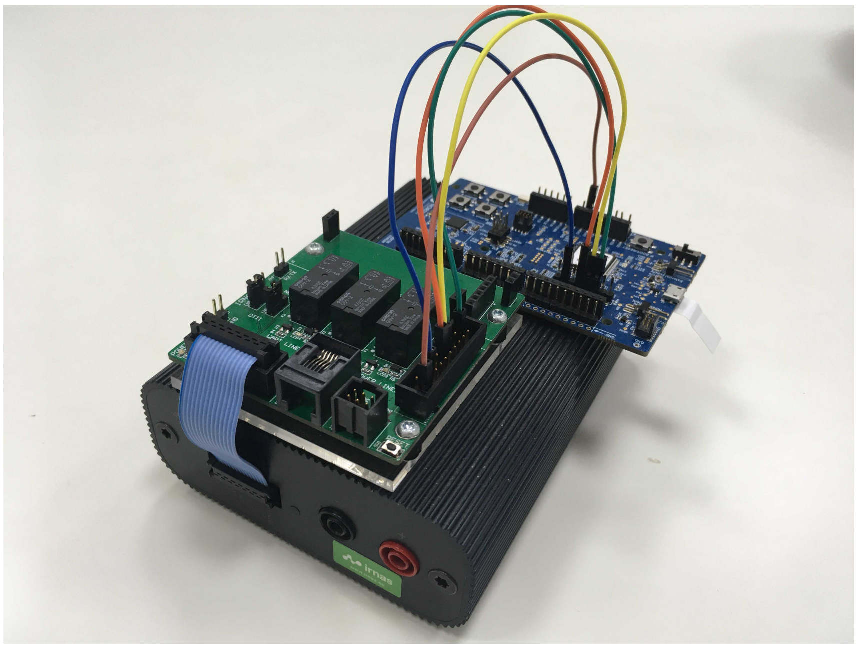

Figure 1.6: Otii Arc with nRF52832 DK and added measurement board made by IRNAS.

Measurements analysis is done through desktop application, example of it is seen in

Figure 1.7. Application enables users to select a part of the measurement, for which it automatically computes minimum, maximum and average values. To present our results we exported current measurements in CVS format and plotted them with Matplotlib.

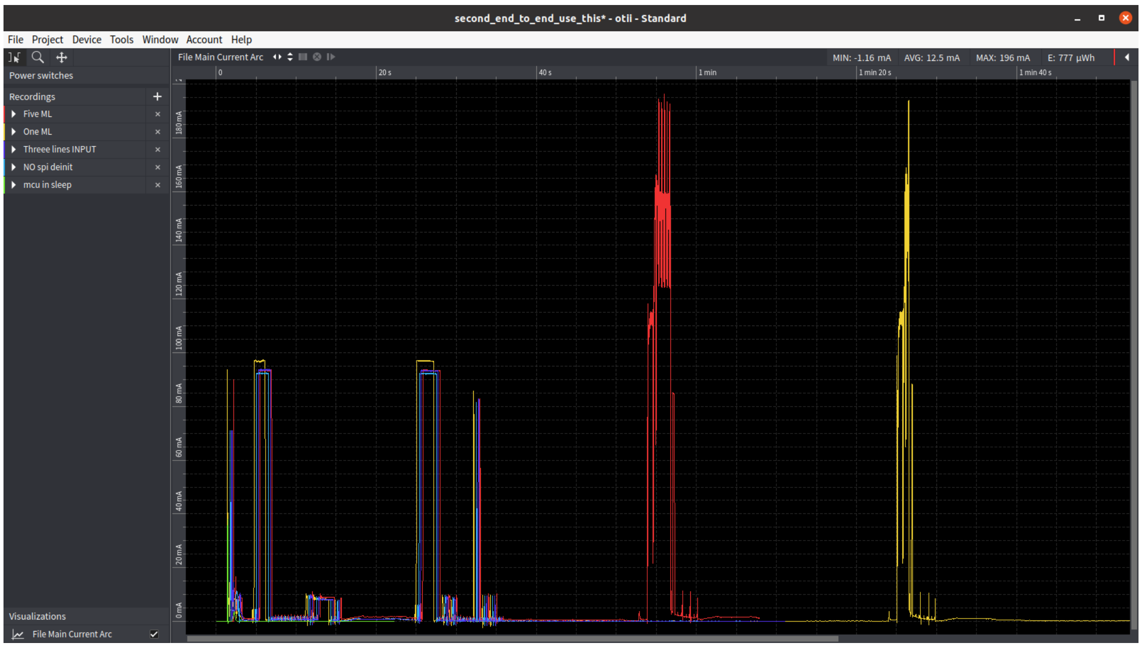

Figure 1.7: Screenshot of Otii user interface.

In the next section we evaluate power consumption of our embedded early warning system. We first measured power consumption of the nRF52 microcontroller in 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 FLIR camera, and measured power consumption of the whole detection sequence.

1.4.2 Current measurements

We conducted all our measurements with Otii's output voltage set to 3.3 V. Before measuring current consumption of the whole image processing sequence we wanted to evaluate current consumption of nRF52 microcontroller in the low-power state. In Zephyr kernel such procedure is relatively simple, type of sleep mode is configured with Kconfig file and microcontroller transitions into it whenever it enters lowest idle thread. Peripherals require special attention, as needs to be explicitly specified which one needs to be turned off. We turned off both of the UART peripherals and SPI peripheral, while keeping GPIO active, as we needed a GPIO interrupt to wakeup nRF52 from low-power state. We also had to make sure that nRF52 microcontroller is completely disconnected from 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 voltage output of Otii to external power pins of nRF52840 DK board. Figure 1.8 shows us the current consumption in the low-power state.

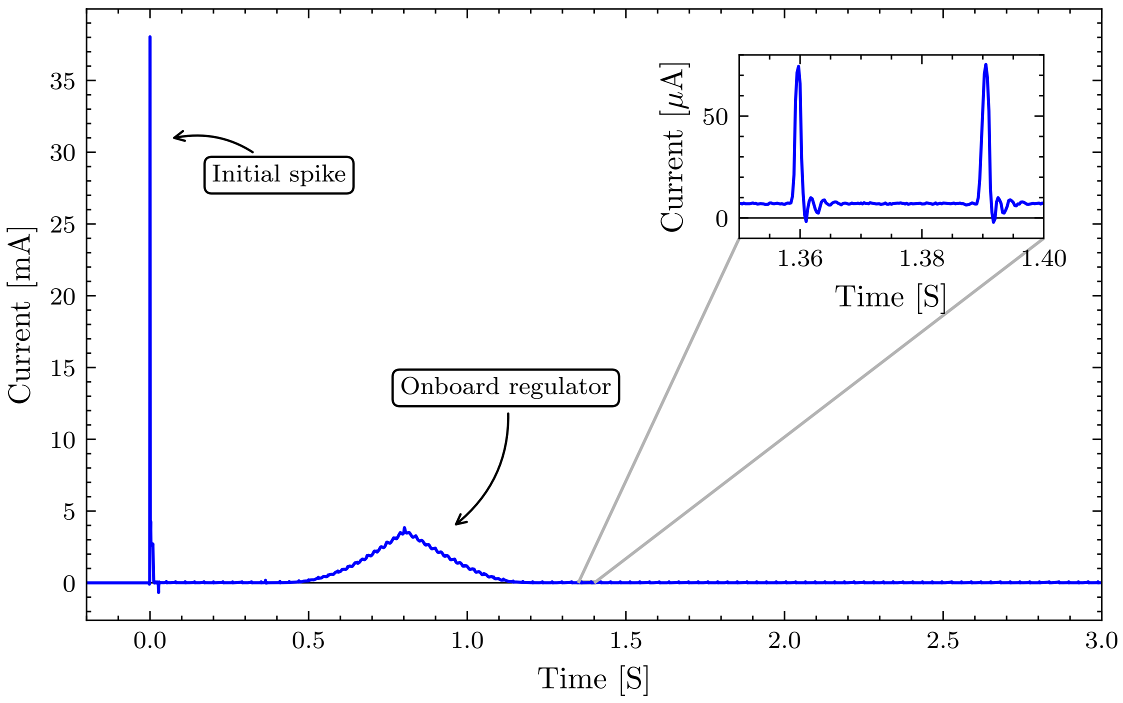

Figure 1.8: Current consumption of nRF52840 microcontroller in low-power state.

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

Smaller graph inside Figure 1.8 shows close up view of the current consumption. The peaks reached 70.3 µA, steady state was at 6.9 µA, while the average was 9.1 µA. Peaks were repeating at frequency of 33.3 Hz.

Measured average current was higher than expected, according to the nRF52's datasheet [3] current consumption should be 3.16 µA.

In next measurement we connected LR1110 shield to the nRF52840 DK board and observed current consumption of the initial LoRaWAN join sequence. Current profile of it can be see in Figure 1.9. Besides initial spike we can see additional four pulses afterwards with some smaller spikes in between. As we are using Over-The-Air Activation (OTAA) LR1110 first has to negotiate for a set of keys with the server before it can start transmitting. This happens in first two pulses, in third pulse LR1110 confirms the set of keys and fourth one first message is sent. First LoRa message is sent automatically, as it is a part of the LoRa Cloud service. Although we do not use this service we can not completely disable it. Average current consumption of LoRaWan join procedure is 11.4 mA and lasts for about 34 seconds. Average current consumption in sleep state has increased to 76.8 µA.

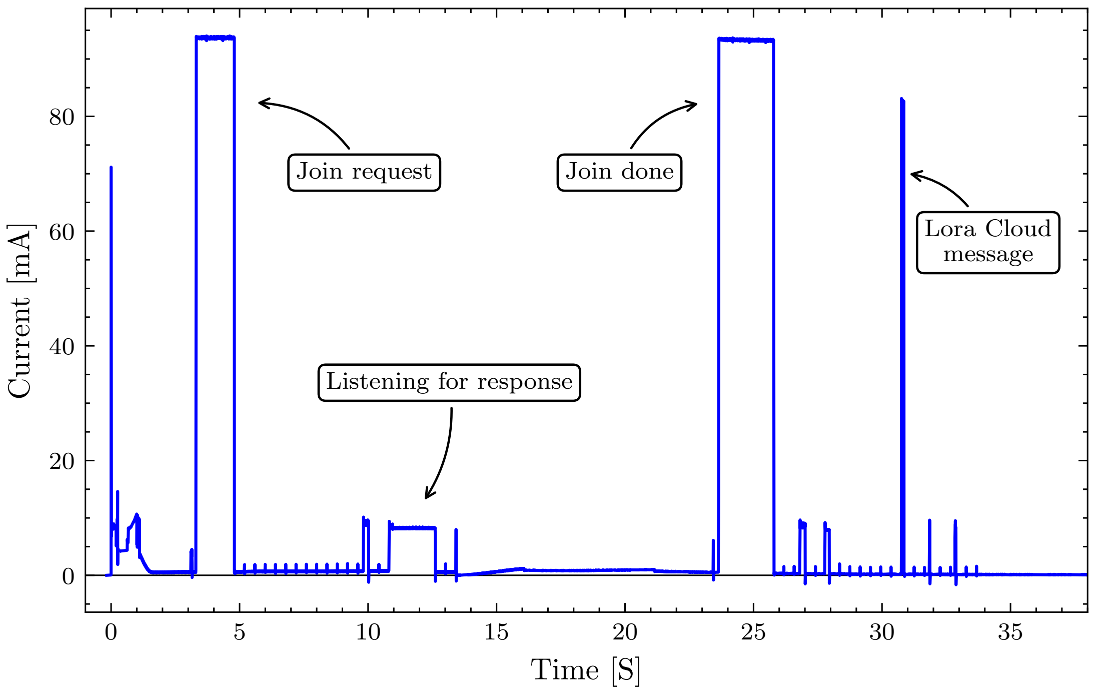

Figure 1.9: Current profile of LoRaWAN join sequence.

In our next test we connected nRF52 to boost converter circuit, Nucleo-F767ZI and FLIR Lepton camera, setup can be seen in Figure 1.10. We did not use PIR sensor as a wakeup source as we saw that its detection was too sensitive to surroundings and we could not completely control it. We instead used a button on nRF52840 DK as a wakeup interrupt. Since we wrote our firmware for libopencm3 in mind we could not use the best performing model *3tl*, as Edge Impulse models only could run on Mbed platform. We instead used our model *460b*, as was most similar to *3tl* in terms of inference time (69 ms compared to 73 ms). In case where image capture and inferencing happened once, we measured total time of the whole detection procedure to be about 1480 ms, not including time needed for LoRaWAN message. Average current consumption for this period was 114 mA. If we measure average current of the whole event shown on Figure 1.11, between 0 and 6 seconds, we get 30 mA.

In case where image capture and inferencing happened five times, we measured total time of the whole detection procedure to be about 2960 ms, not including time needed for LoRaWAN message. Average current consumption for this period was 131 mA. If we add transmission of LoRaWAN message to the measured current consumption, thus increasing the time of the whole detection event to 8 seconds, we measure 51.9 mA.

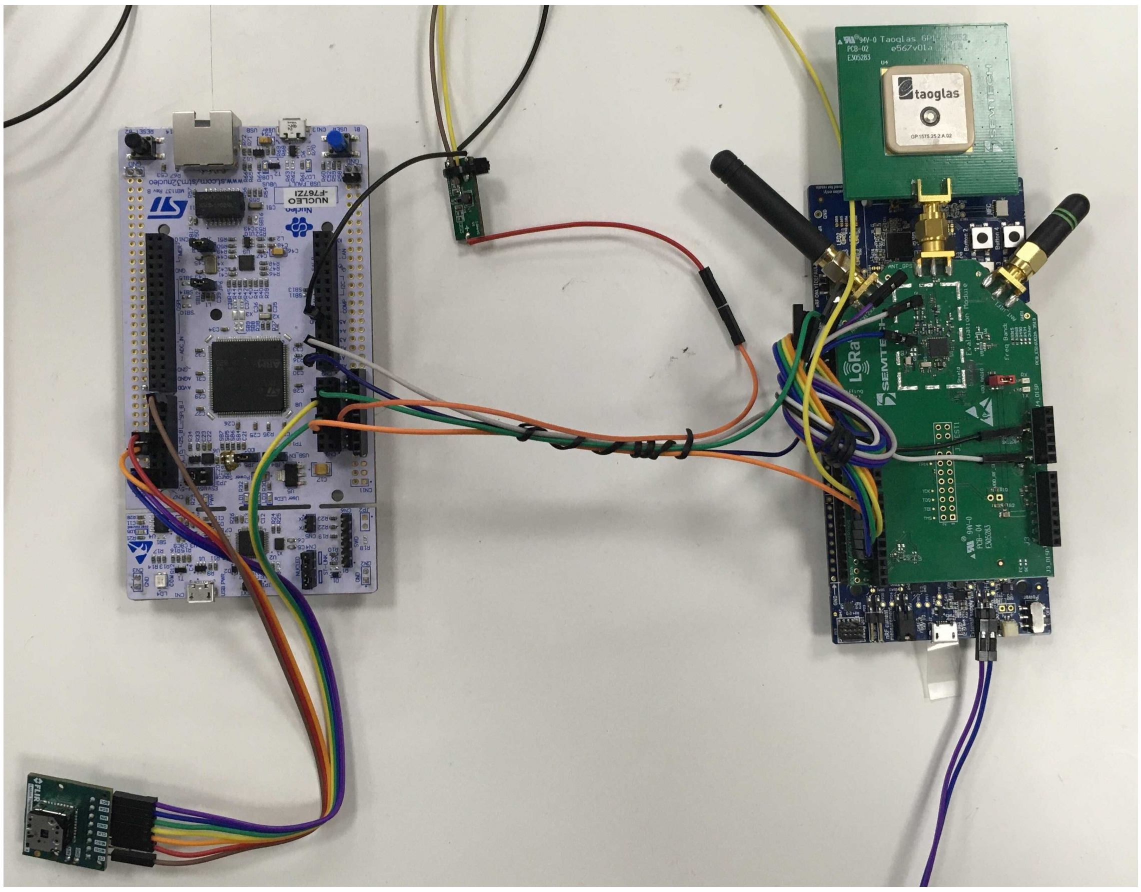

Figure 1.10: Device under test: nRF52840 DK with attached LR1110 shield, boost converter breakout board, Nucleo-F767ZI and FLIR Lepton camera.

We captured two different inference procedures, in first image capture and inference were done once, in second one they were repeated 5 times. Procedures can be seen on Figures 1.11 and 1.12 respectively. Both procedures were followed by a LoRaWAN message that reported results to the server.

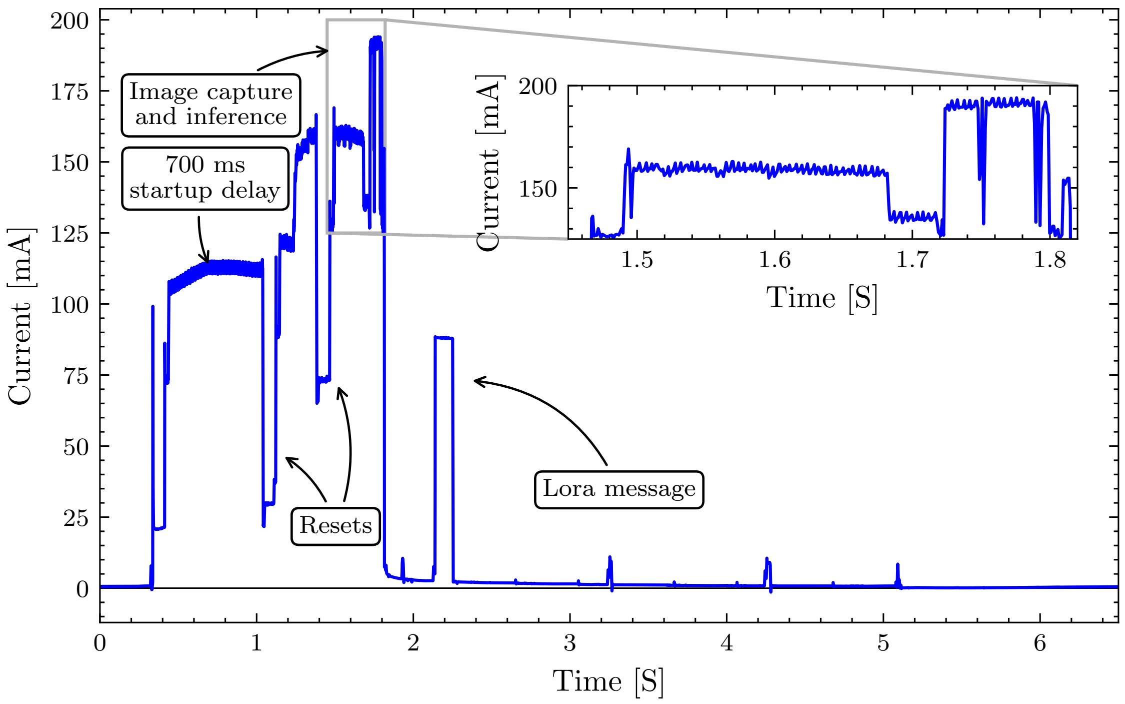

Figure 1.11: Current profile of image capture and inference procedure.

1.4.3 Commentary of the current measurement results

Measured low-power state of nRF52, visible in Figure 1.8, was higher than expected. nRF52's datasheet [3] specifies current consumptions in many different conditions, which depend on: type of sleep mode (System ON or System OFF, latter loses execution context at wakeup), 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 dependant, we can only guess which exact nRF52 sleep mode Zephyr uses. We assumed 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 nRF52840 DK board.

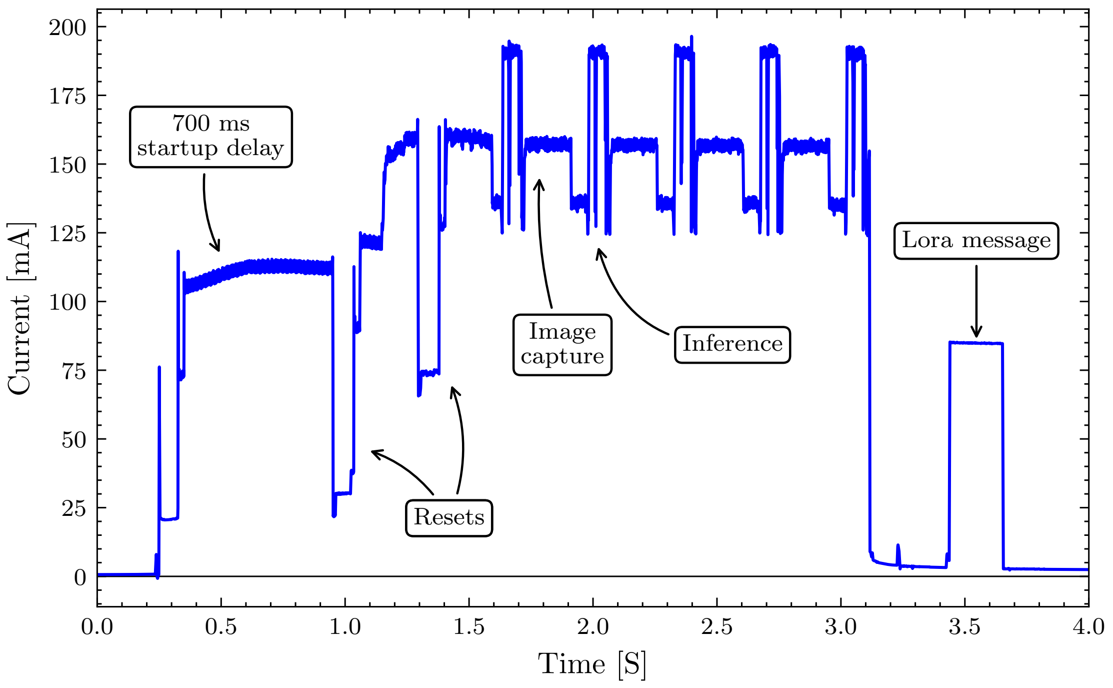

Figure 1.12: Current profile of image capture and inference procedure repeated 5 times.

When we connected LR1110 shield to the nRF52840 DK board we saw that average current consumption increased to 76.8 µA, which is much more than expected. LR1110 automatically enters sleep mode when it is finished with communication, according to its datasheet [4], current consumption should be around 1.85 µA. This current consumption is plausible, as we observed it in other IRNAS's products that use LR1110 chip. We suspect that the incorrect state of common GPIO connections between LR1110 and nRF52 is the cause for increased current consumption, however we were not able to fix the problem.

We mentioned that the time required for detection is about 1480ms if we capture one image and process it or 2960 ms if done five times. We can see that both detection procedures started with a 700 ms of delay, according to FLIR Lepton's datasheet [5], a delay is required before we can start communicating with the camera over TWI interface. Two microcontroller resets are visible, during our testing we saw that we could not properly communicate with the camera, if we did not reset STM32 twice before that. To accomplish this we simply connected a reset pin of STM32 with one of available nRF52 GPIO pins.

1.4.4 Battery life estimations

average currents: sleep 76.8 µA. you should add here PIR sensor consumption. 5ml detection 51.9 mA.

2 Conclusion

TODO SUMMARY

2.1Future work

Our research into elephant detection with the aid of Machine Learning models yielded promising results and should be improved.

In terms of model performance, we can always improve it by gathering more relevant training data. Dataset that we used contained several thousands images of elephants, but only couple thousands images of humans and only few hundreds 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 Transfer Learning technique. We saw that Transfer Learning models reached higher accuracies with shorter inferencing times compared to other models. We expect that running a random hyperparameter search with the smaller version of pre-trained MobileNetV2 model could produce optimal results.

In terms of the system performance, testing our early warning system in field would give us key insights 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 a SD card to the system and save every taken image, and result of its inference.

By observing 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.

TODO preveri če se napisano sklada prejšnjo sekcijo. In terms of battery life performance, we can definitely lower the current consumption of the nRF52 microcontroller in the low-power state, reaching 10µA with LR1110 in sleep mode is completely possible. Early detection system should be implemented on a custom printed circuit board, such approach would give use more control over current consumption than the setup that we created from development boards.

TODO You are probably not finished with this section, you need feedback.

2.2Final words

Machine Learning on the embedded devices is opening doors to various, wonderful applications that were not possible 3 years ago.

Main three sections: summary, results commentary, and looking forward (future work)

, you are summarizing the paper for a reader who had read the introduction and the body of the report already, and should already have a strong sense of key concepts. Your conclusion, then, is for a more informed reader and should look quite different than the introduction.

Sklep je zadnje poglavje zaključnega dela. V njem podamo objektivno oceno rezultatov in jih povežemo s problemom, zastavljenim v uvodu. Če se nam zdi ustrezneje, lahko opis rezultatov in diskusijo podamo v sklepu namesto v glavnem delu. Nakažemo morebitne težave in opažanja, ki so se nam pojavila med delom, ter podamo napotke za nadaljnje delo.

Bibliography

[1] 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.

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

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

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

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