Contents

1 Neural network model design

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

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

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

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

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

1.1 Model objectives

The accuracy of our early detection system should be equal or similar to the one of the human observers, no matter if it is operating in daytime or nighttime. Although the system will be placed on the paths that are regularly traversed by elephants, they are not the only possible objects that can appear on taken thermal images. Humans and various livestock, such as goats and cows, could also be photographed. Reporting false positives should be avoided, which means that the system should not incorrectly label a human or a livestock animal as an elephant. At the same time, false negatives also need to be avoided, as an elephant could pass the system undetected. These kinds of mistakes could undermine the community's confidence in the early detection system and defeat the purpose. This means that besides elephant detection, we should also focus on correctly classifying humans and livestock while providing a nature/random class for all other unexpected objects or simply images of nature.

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

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

To summarize:

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

keep it under 1 second.

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

1.2 Tools and development environment

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

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

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

1.3 Creating the dataset

As mentioned in section ??, major part of thermal image dataset was provided by Arribada Initiative [1] [2]. Images in the dataset come from two different locations: Assam, India, and ZSL Whipsnade Zoo, United Kingdom.

Assam served as a testing ground. Arribada team positioned two camera traps on two locations that overlook paths commonly used by elephants. Cameras were built out of Raspberry Pi, PIR sensor, FLIR Lepton 2.5 camera, and batteries, all of which were enclosed in a plastic housing. Insides of the camera and an example of a deployed camera can be seen in Figure 1.1.

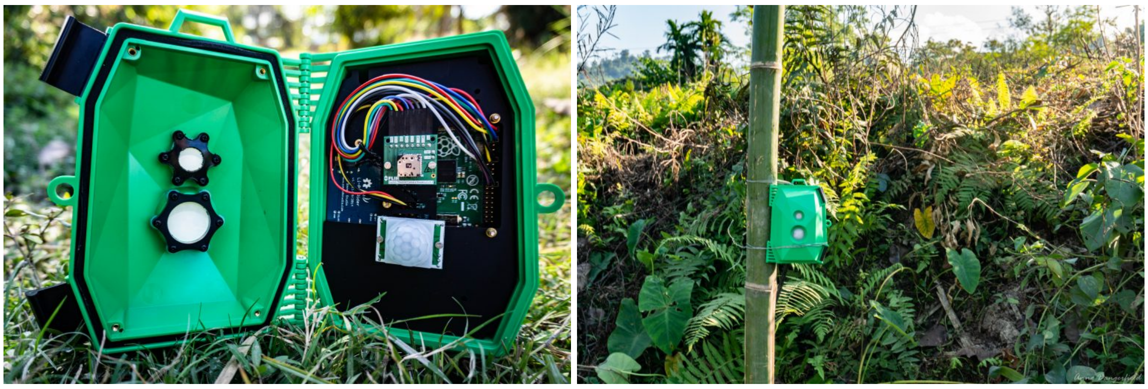


Figure 1.1: Camera trap used in Assam, India. Image source: Arribada Initiative [2]

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

Images of elephants that came from both locations can be seen in Figure 1.2.

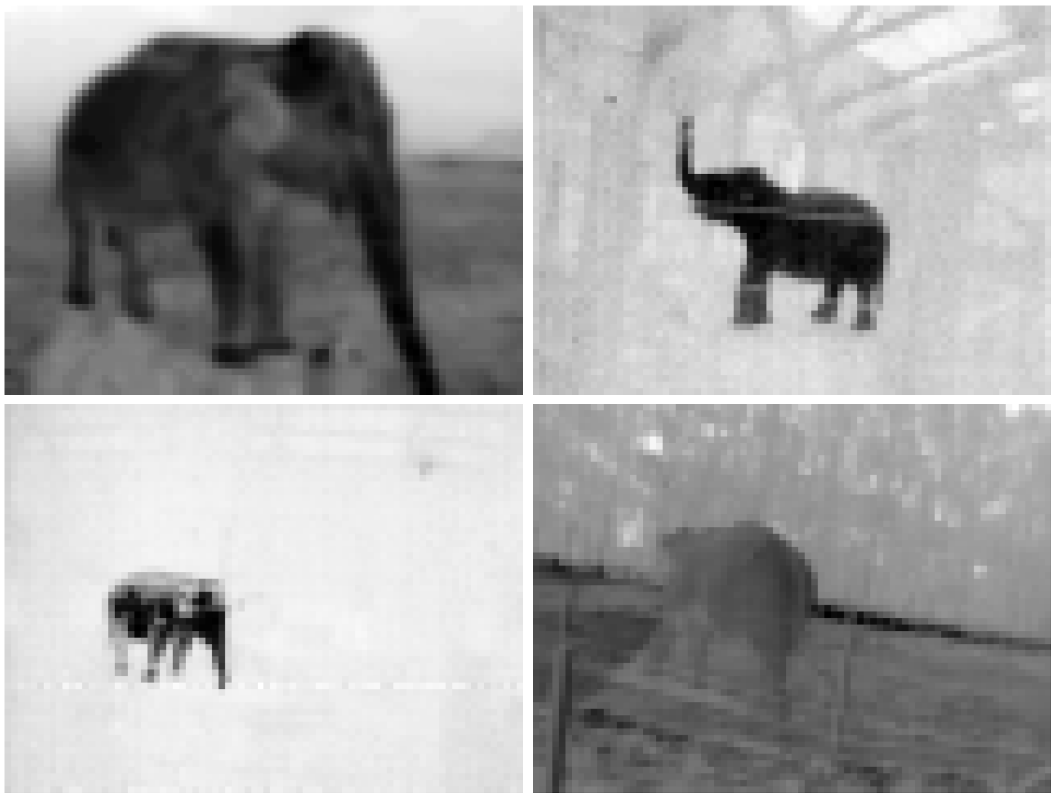


Figure 1.2: Thermal images of elephants from dataset.

Small part of thermal image dataset was provided by us. This was done because the number of images of cows was low compared to the number of human and elephant images and because we also did not have any images that could be used for nature/random class. We wanted to gather images as quickly and efficiently as possible so we build a prototype camera made out of FLIR Lepton 2.5 breakout board, Raspberry Pi Zero, and power bank. We used an open-source library [4] 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 1.3.

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

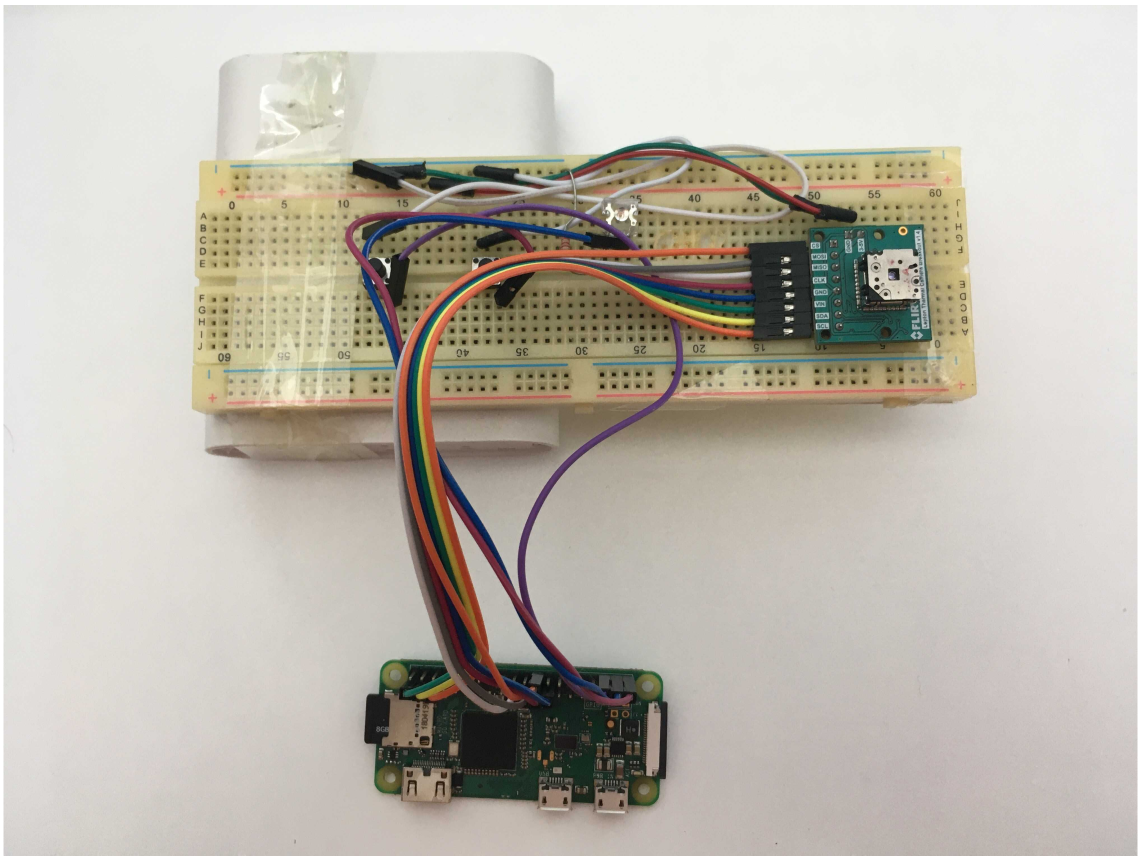


Figure 1.3: Camera setup used for taking thermal images with FLIR Lepton 2.5.

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

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

1.4 Exploring the dataset

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

After examining the folder, we came to several conclusions.

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

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

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

2. We found out that all images were documented in a large Excel database. For each image, there was a row in a database that connected the image file name with the information where the image was taken and with what sensor. This enabled us to generate a graph seen in Figure 1.4.

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

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

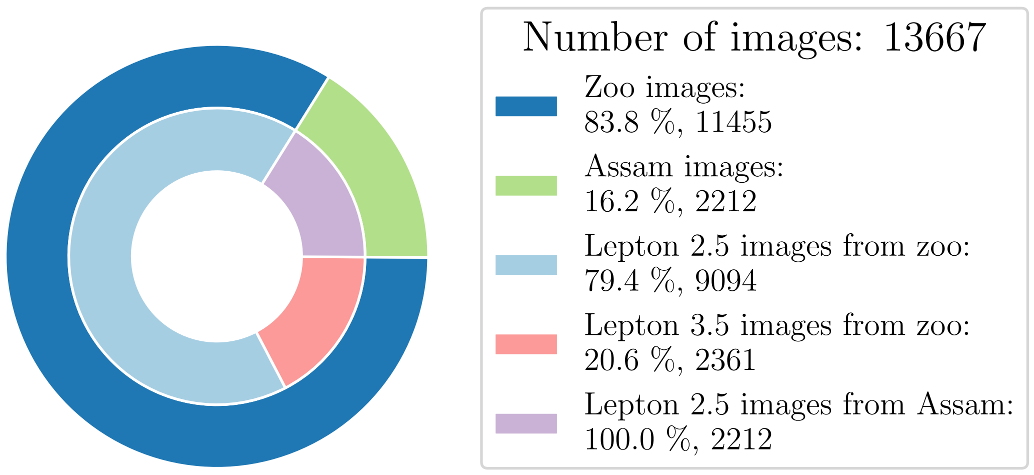


Figure 1.4: Distribution of used images from thermal dataset depending on image location and type of sensor.

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

4. We also realized that there was a large class imbalance, as seen in Figure 1.5 in favor of elephant class.

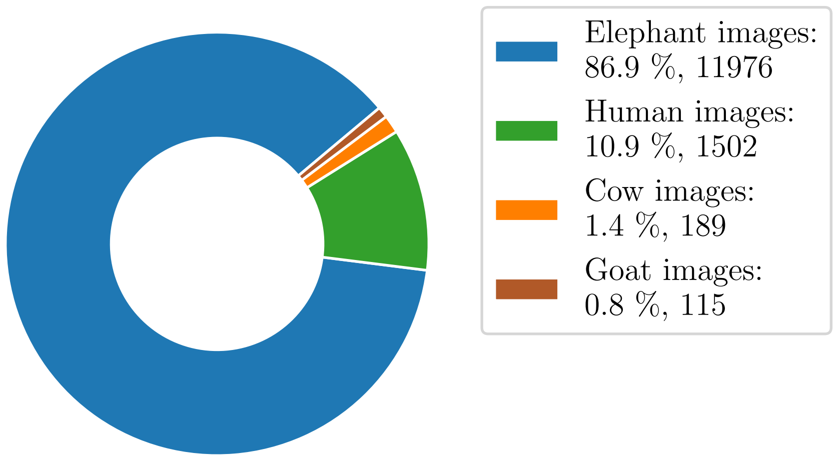


Figure 1.5: Class distribution of thermal images.

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

1.5 Image preprocessing

The image preprocessing phase is a pipeline process that differs from project to project. Our process can be seen on Figure 1.6.

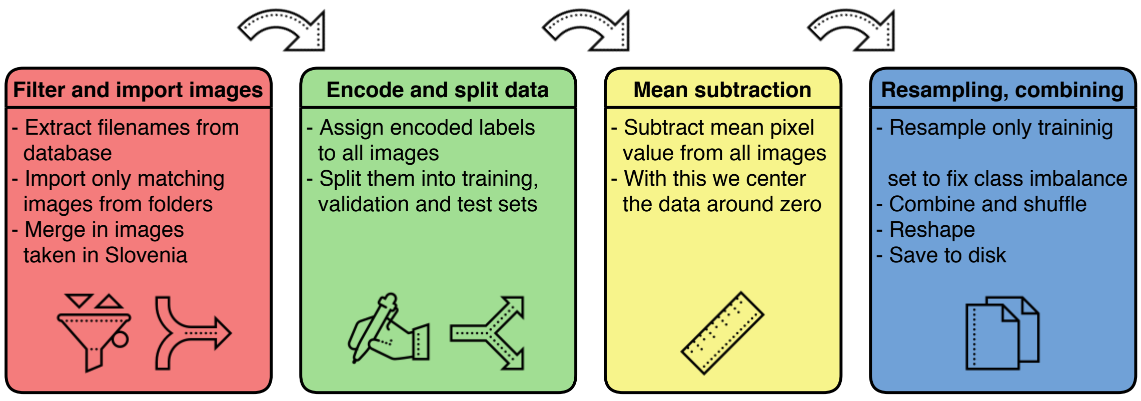

Filter and import images
- Extract filenames from database
- Import only matching images from folders
- Merge in images taken in Slovenia
Encode and split data
- Assign encoded labels to all images
- Split them into training, validation and test sets
Mean subtraction
- Subtract mean pixel value from all images
- With this we center the data around zero
Resampling, combining
- Resample only traininig set to fix class imbalance
- Combine and shuffle
- Reshape
- Save to disk

Figure 1.6: Image preprocessing pipeline. Icons source: [5]

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

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

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

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

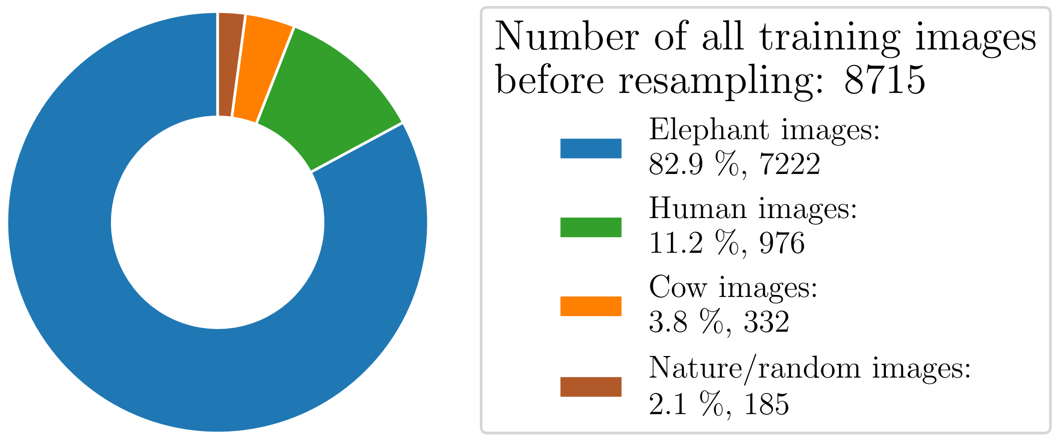
We next applied the simplest form of normalization to all images, a mean subtraction. We calculated a two-dimensional matrix that held mean values of pixels averaged over the whole training set, which we subtracted from all images, essentially zero centering the data. This is a common preprocessing step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

We achieved this by resampling the human, cow, and nature/random classes. The human class was resampled 5 times, while both cow and nature/random classes were resampled 8 times. Figure 1.7 shows the distribution of training images before and after resampling.

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

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

[1]Standardization scales the whole range of input pixel values into -1 and 1 interval. This is only needed if different input values have widely different ranges [6]. Because images that were created with FLIR camera were all 8-bit encoded, therefore had the same range, this was not needed.



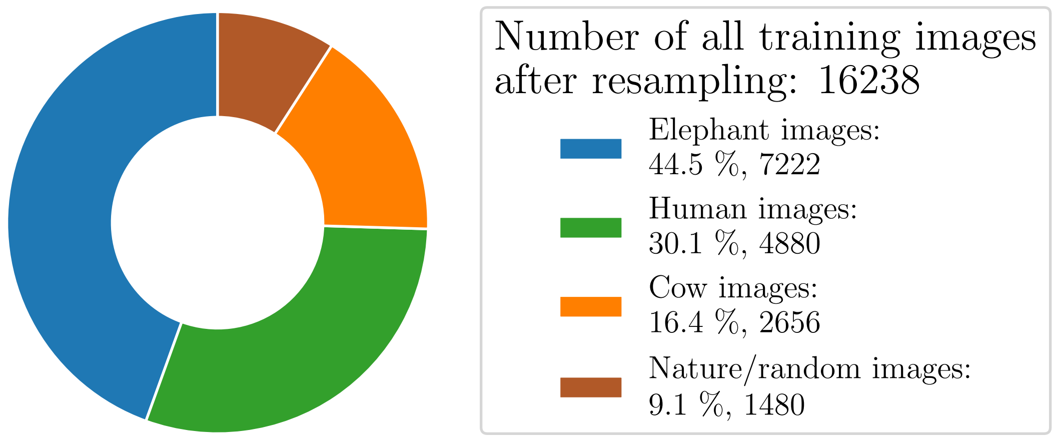


Figure 1.7: Distribution of training images before and after resampling.

1.6 Model creation and training

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

For a model architecture, we decided to use a simplified version of a common CNN architecture that was shown in Figure ??. The best way to present the model is by inspecting the Sequential API code that creates it, code is shown in Figure 1.8.

The model consisted of two pairs of convolutional and max-pooling layers, followed by a final convolutional layer. For activation function ReLu was chosen, as it is currently the most effective and popular option [6] [7]. The padding option was set to same, which meant that a spatial dimension of a volume would not change before and after a convolutional layer. Polling layer kernel size was set to 2 x 2, with a default stride of 2.

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

The last dense layer was a final output layer with only 4 neurons, each one representing one class. Softmax activation was used to calculate class probabilities. Model was set to use Adam optimizer and sparse categorical crossentropy loss function. Adam is an upgraded version of gradient descent method, which automatically adapts learning rate to decaying gradients [7]. It is generally easier to use than gradient descent as it requires less tuning or learning rate hyperparameter. Sparse categorical crossentropy loss function is used when building a multi-class classifier.

Above set hyperparameters follow general rules of thumb and serve as a good starting point when building CNNs [6]. However, hyperparameters such as the number of filters, filter size, size of a hidden dense layer, dropout rate, and learning rate are specific to each dataset and can not be chosen heuristically.

To find hyperparameters that would yield the highest accuracy we used the Keras Tuner module. 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.

The created model was then passed to a `RandomSearch` class, with few other pa-

[2]Dropout layer decides with probability p in each training step how many activations from the previous layer will be passed on to the next layer. It is active only during the training phase, during the testing phase activations are multiplied with $(1-p)$ factor to compensate. It is a very popular type of regularization technique, which makes models more robust to the input data [7].

rameters such as batch size, number of epochs and maximum number of trials. Hyperparameter search could then be started, Keras Tuner would randomly pick a set of hyperparameters and train a model with them, this process would be repeated for a trial number of times. Accuracy and hyperparameters that were used while training every module were saved to a log file for later analysis.

By providing accuracy metric, Keras tuner automatically sorts trained models in descending accuracy, returning the best model. This is in many use cases sufficient, however in our case, we were also interested in best performing models that had small size. As size can not be specified as one of the possible metrics, some manual training was required after the hyperparameter search to determine the most efficient model.

Comparison of trained models is presented and discussed in section TODO ADD REFERENCE.

```
        model = models.Sequential()

        model.add(Conv2D(filter_num_1, filter_size,
                         activation='relu',
                         padding="same",
                         input_shape=(60,80, 1)))

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

        model.add(Conv2D(filter_num_2, filter_size,
                         activation='relu',
                         padding="same")

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

        model.add(Conv2D(filter_num_3, filter_size,
                         activation='relu',
                         padding="same")

        model.add(Flatten())

        model.add(Dense(dense_size, activation='relu'))
        model.add(Dropout(dropout_rate))
        model.add(Dense(4), activation='softmax')

```

Figure 1.8: CNN architecture written in Python using Keras Sequential API.

1.7 Model optimization

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

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

- **Non-quantized tflite model,** no quantization, just basic conversion from .h5 to .tflite format is done.
- **float16 model,** weights are quantized from 32-bit to 16-bit floating-point values. The model size is split in half and the accuracy decrease is minimal, but there is no boost in execution speed.
- **dynamic model,** weights are quantized as 8-bit values, but operations are still done in floating-point math. Models are 4 times smaller and execution speed is faster when compared to float16 optimization but slower from full integer optimization.
- **Full integer model,** weights, biases, and math operations are quantized, execution speed is increased. It requires a representative dataset at conversion time.

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

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

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

1.8 Neural network model design in Edge Impulse Studio

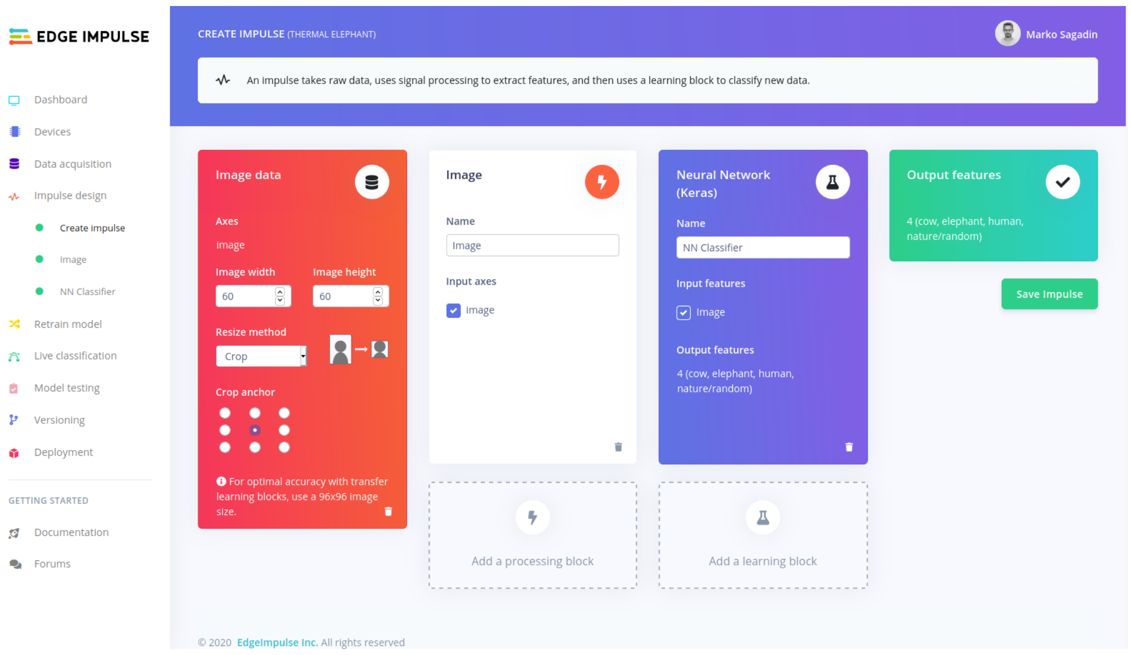
Designing a neural network with Edge Impulse is a much less involved process than the one we described above, as many steps of image preprocessing are automated. To start with NN design, we first had to upload our image data to the Edge Impulse Studio project. This can be done either by connecting an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

After the data was uploaded, the rest of NN design was done through Edge Impulse web interface. In Figure 1.9 we can see the so-called Impulse Design tab, where we design by selecting different blocks. With input block, we tell what kind of data are we inputting, either image or time-series data, and with processing block we decide how are we going to extract features. With learning block, we can choose to use a neural network provided by Keras, an anomaly detection algorithm, or a pre-trained model for transfer learning.

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

Neural network block is configurable, we could either define our network with differ-

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



CREATE IMPULSE (THERMAL ELEPHANT)

An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Image data
Axes: image
Image width: 60
Image height: 60
Resize method: Crop
Crop anchor
For optimal accuracy with transfer learning blocks, use a 96x96 image size.

Image
Name: Image
Input axes: [✓] image

Neural Network (Keras)
Name: NN Classifier
Input features: [✓] Image
Output features: 4 (cow, elephant, human, nature/random)

Output features
4 (cow, elephant, human, nature/random)

© 2020 EdgeImpulse Inc. All rights reserved

Figure 1.9: Creating a neural network in Edge Impulse Studio.

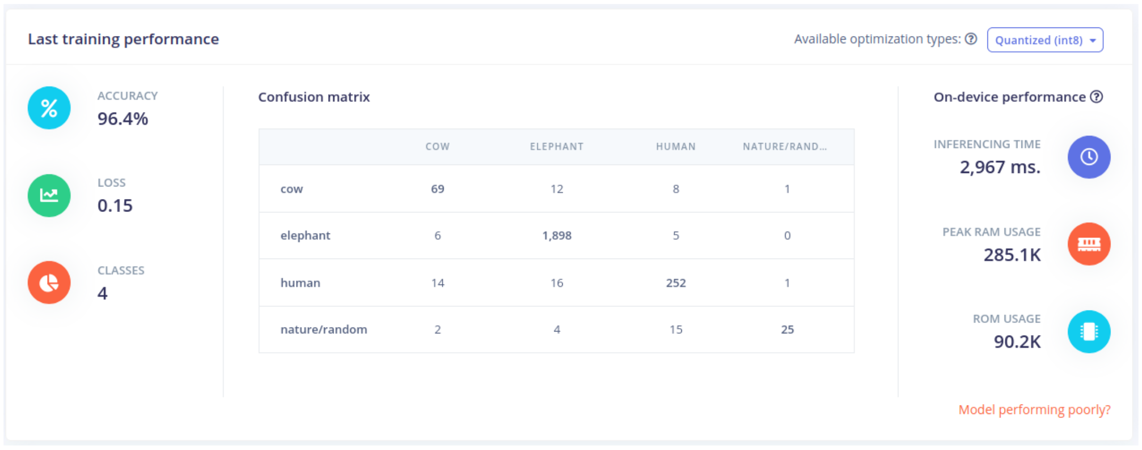
ent blocks representing layers or switch to the text editor with Keras Sequential API code, where we could do our adjustments. Settings such as learning rate, number of epochs, and confidence rating are also available regardless of the option we chose. Training of neural networks inside Edge Impulse Studio is done in a cloud, so as users we do not have to worry about settings up a development environment. We only had to start it and wait for it to finish.

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

An example of the output is presented in Figure 1.10, inferencing time is estimated for a Cortex-M4 microcontroller, running at 80 MHz. Edge Impulse also does conversion to an optimized full-integer model automatically.

The final step was deploying the trained model to the microcontroller. This step is fairly simple, Edge Impulse provides few example projects on their GitHub for different platforms that it supports. As we wanted to compare the performance of the

amount of data in the cloud.



Last training performance

Available optimization types: Quantized (int8)

ACCURACY 96.4%

LOSS 0.15

CLASSES 4

Confusion matrix

	COW	ELEPHANT	HUMAN	NATURE/RAND...
cow	69	12	8	1
elephant	6	1,898	5	0
human	14	16	252	1
nature/random	2	4	15	25

On-device performance

INFERENCING TIME 2,967 ms.

PEAK RAM USAGE 285.1K

ROM USAGE 90.2K

Model performing poorly?

Figure 1.10: After training training performance report.

models on ab STM32f767ZI, we chose the Mbed platform. We copied the example Mbed project from GitHub and in Edge Impulse Studio we selected to generate an inferencing library with our model for the Mbed platform. We extracted the library, which consisted of C++ files, into an example project and compiled it. An example project just continuously runs the inference on one image and outputs results over the serial port. A performance comparison between this example project and our implementation is done in section TODO ADD REFERENCE.

2 Planning and design of early warning system

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

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

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

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

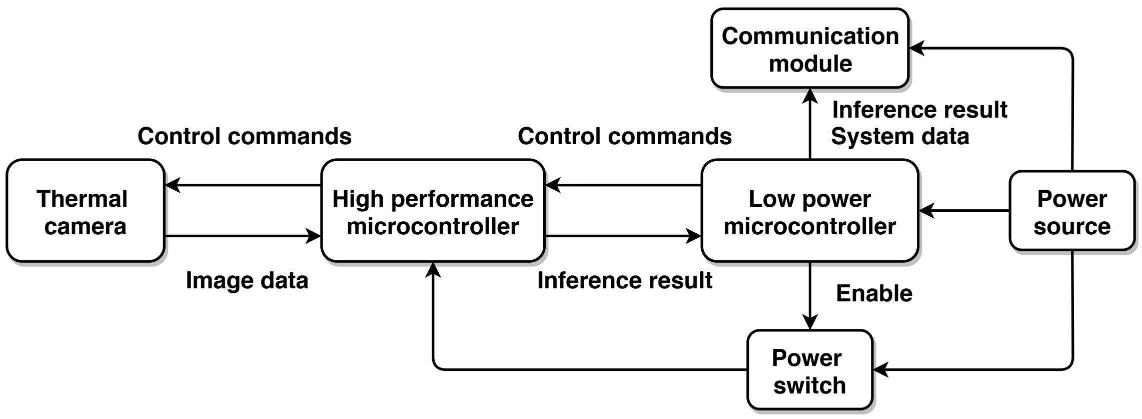


Figure 2.1: General block diagram of an embedded system

Embedded system will consist of two different microcontrollers with two distinct tasks, a thermal camera, PIR sensor, wireless communication module, power switch and battery.

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

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

Thermal camera only communicates with high performance microcontroller, which configures it and requests image data. That data is then inputted into neural network algorithm and an probability results are then returned to a low power microcontroller. low power microcontroller then packs the data and sends it over radio through wireless communication module. Power source to high performance microcontroller and thermal camera is then turned of to conserve power. Diagram of described procedure can also be seen on Figure 2.2.

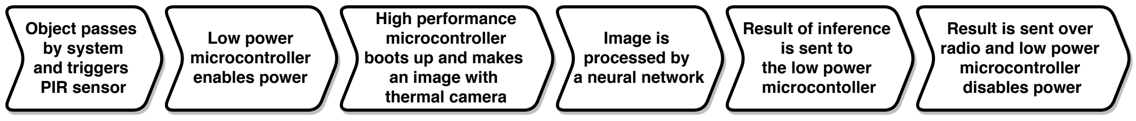
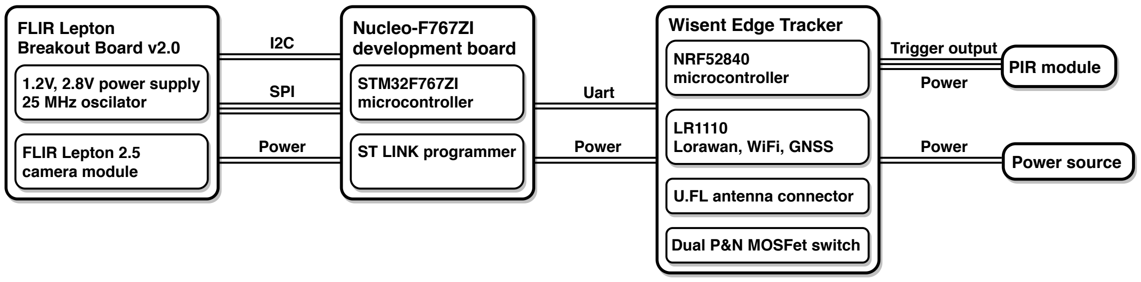

Object passes by system and triggers PIR sensor
Low power microcontroller enables power
High performance microcontroller boots up and makes an image with thermal camera
Image is processed by a neural network
Result of inference is sent to the low power microcontoller
Result is sent over radio and low power microcontroller disables power

Figure 2.2: Diagram describing behavior of embedded early detection system

2.1 Hardware

In this section we present concrete components that we used to implement the embedded part of the early detection system. Hardware version of embedded system diagram is presented on the Figure 2.3. It should be noted that we did not include specific power source into the diagram. Wisent Edge tracker board is general enough to work with different power sources, such as non-chargeable or chargeable batteries and or solar cells.



FLIR Lepton Breakout Board v2.0
1.2V, 2.8V power supply 25 MHz oscilator
FLIR Lepton 2.5 camera module
I2C
SPI
Power
Nucleo-F767ZI development board
STM32F767ZI microcontroller
ST LINK programmer
Uart
Power
Wisent Edge Tracker
NRF52840 microcontroller
LR1110 Lorawan, WiFi, GNSS
U.FL antenna connector
Dual P&N MOSFet switch
Trigger output
Power
PIR module
Power
Power source

Figure 2.3: Hardware diagram of embedded early detection system

2.1.1 Nucleo-F767ZI

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

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

2.1.2 Wisent Edge tracker

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

[1]As a part of OpenCollar project, the design of Wisent Edge is open-source and available on GitHub [8], alongside other hardware and firmware tracker projects.

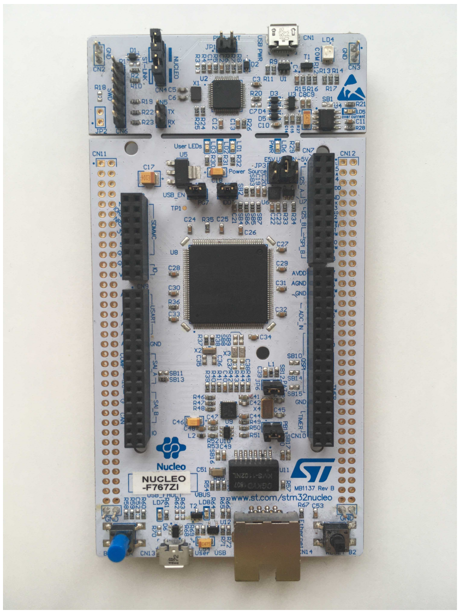


Figure 2.4: Nucleo-F767ZI development board

poaching and animal conservation areas.

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

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

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

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

[2]Reason for two GNSS modules is that although LR1110 chip can provide extremely power efficient location information, it's accuracy is smaller when compared to ZOE-M8G and it can only be resolved after sending it to an application server [9].

channel p and n MOSFET, circuit can be seen on Figure 2.5. Circuit functions as a high side switch, with microcontroller pin driving enable line. When enable line is low, n MOSFET is closed, therefore p MOSFET is also closed, as it is pulled high by resistor R1. When enable line is high, n MOSFET is opened, therefore gate of p MOSFET is grounded, which opens the MOSFET.

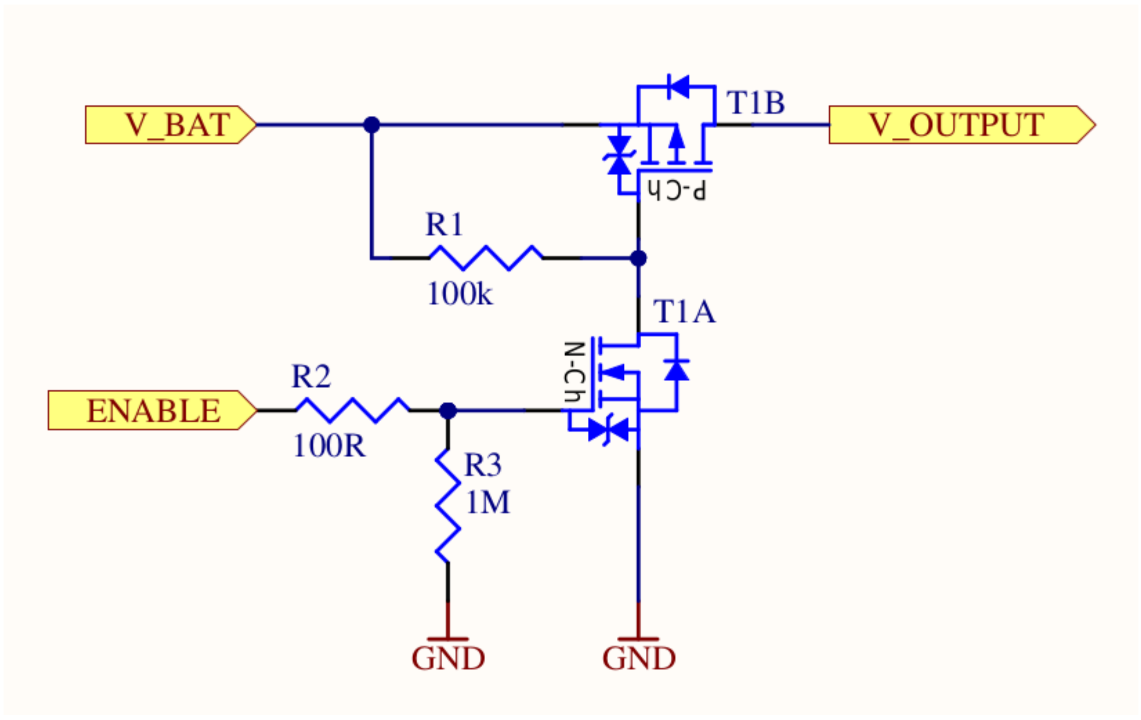


Figure 2.5: Dual P and N MOSFET configuration for power switching

2.1.3 Flir Lepton 2.5 camera module and Lepton breakout board

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

FLIR Lepton camera needs to be powered from two different sources, 1.2 V and 2.8 V, as well it needs a reference clock of 25 MHz. All of this is provided by Lepton breakout board, which can be seen on the Figure 2.6. Front side of the breakout board contains a FLIR module socket and back side has two voltage regulators and a oscillator. Breakout board can be powered from 3.3 to 5 V and also conveniently breaks out all communication pins in form of headers.

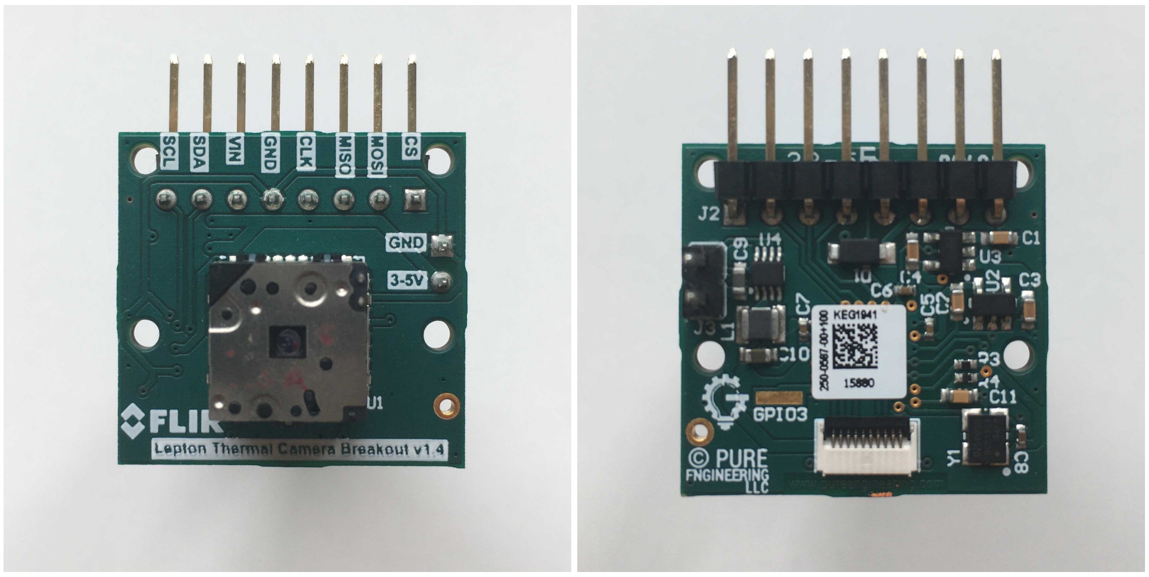


Figure 2.6: Front and back side of Flir Lepton breakout board with thermal camera module inserted.

FLIR Lepton module itself conatins five different subsystems that work together and can be configured:

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

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

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

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

Internal structure of Lepton's control block can be seen on the Figure 2.7. Whenever we are communicating with FLIR camera we have to specify which subsystem are we addressing, what type of action we want to do (get, set or run), length of data and data itself.

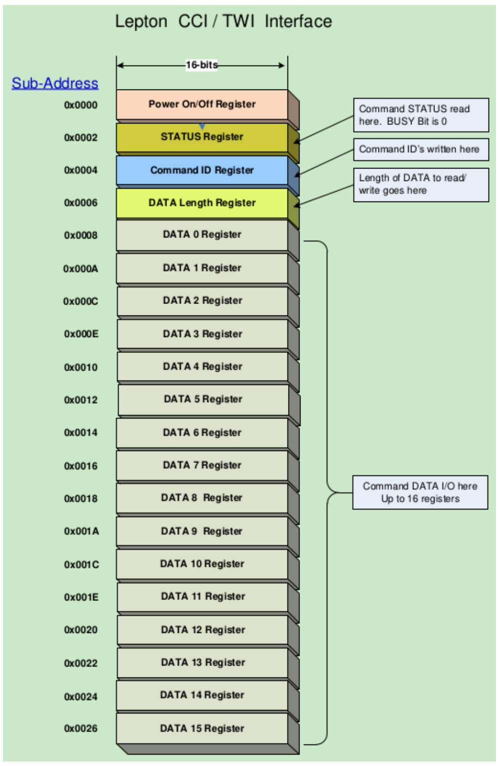


Figure 2.7: Command and control interface of FLIR Lepton camera.

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

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

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

2.1.4 PIR Sensor

2.2 Firmware

2.2.1 Tools and development environment

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

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

2.2.1.1 Development environment for STM32f767ZI

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

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

As libopencm3 does not provide `printf` functionality out of the box we used excellent library by GitHub user mpaland [10]

2.2.1.2 Development environment for NRF52840

To develop firmware for NRF52840 we decided to use The Zephyr OS, which is a small kernel, designed for IoT embedded systems. Besides usual RTOS functionalities such as tasks, mutexes, semaphores it also provides common driver API for supported microprocessors.

2.2.2 Architecture design

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

As seen on Figure 2.8 we split the firmware into two hardware and application modules.

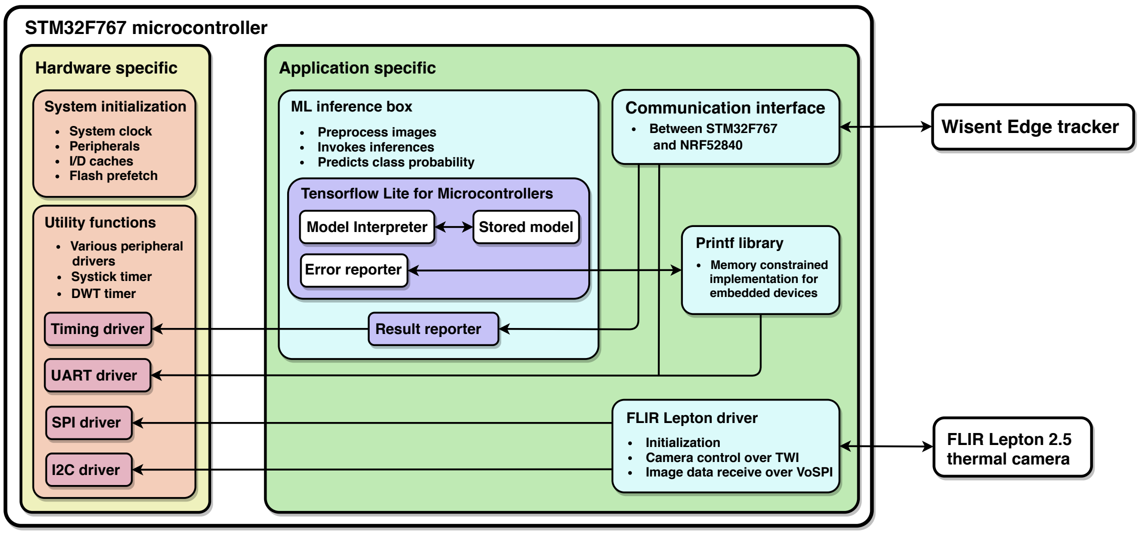


Figure 2.8: Architecture diagram of the firmware that is running on a STM32F767ZI microcontroller.

Both modules are lightly coupled, which enables us reuse of the application module on a different hardware sometime in the future. Hardware specific module is mostly using libopencm3 API to set the system clock and initialize peripherals. Small function wrappers had to be written to make use of various peripheral drivers more abstract. To profile execution of our code we first wrote a timer driver based on a Arm's systick timer, however we later decided to use data watch trigger (DWT). DWT does not use interrupts, therefore it does not introduce overhead of calling interrupt routines like systick timer does.

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

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

TODO describe communication interface between THIS AND WISENT EDGE.

The architecture diagram for NRF52840 can be seen on Figure 2.9. For NRF52840 microcontroller, we did not had to write any peripheral drivers, as they were provided by Zephyr itself. Priority was to achieve low power consumption, for which NRF52840 had to spend most of its time in sleep mode, such behavior was easily configurable in Zephyr.

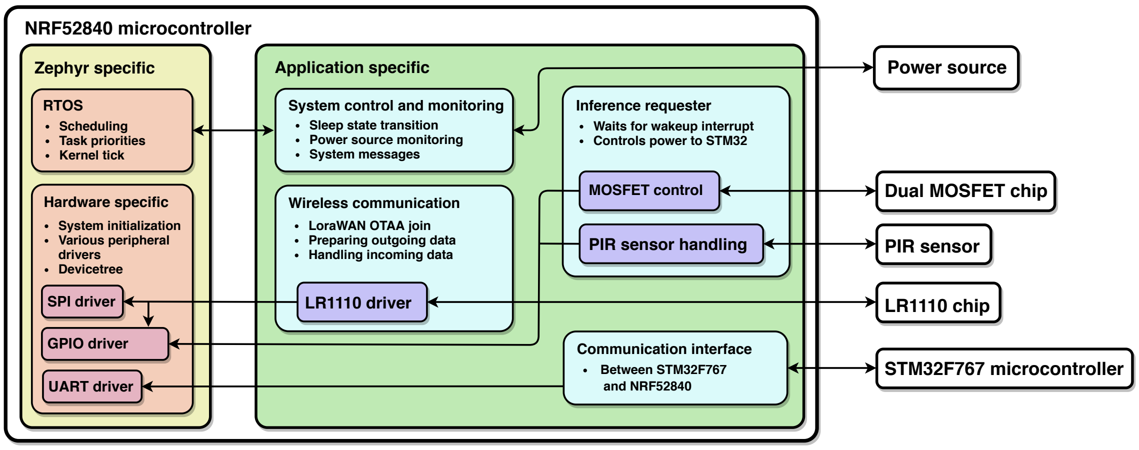


Figure 2.9: Architecture diagram of the firmware that is running on a NRF52840 microcontroller.

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

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

TODO describe communication interface between THIS AND WISENT EDGE.

2.2.3 MicroML and build system

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

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

We learned a useful principle while observing the build process. Each time we would build an example for a new platform, Make would first compile all TensorFlow files, create a static library out of them, compile specific example source files and then link against library in linking stage. If we wanted to build firmware for a different example, Make would only had to compile source files of that example and it could reuse previously made library. As compiling of static library took quite some time, this was an efficient option.

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

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

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

To solve our problem we started developing a small project that we called MicroML[4]. MicroML enables users to develop ML applications on libopencm3 supported microcontrollers. Project's directory structure can be seen on Figure 2.10

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

Figure 2.10: Directory structure of MicroML project.

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

- **project.mk** - It contains information which files need to be compiled inside the project folder. It is a place where we also define which microcontroller are we using and what kind of compiler optimization we would like to set.
- **openocd.cfg** - Configuration file which tells OpenOCD which programmer is used to flash which microcontroller and location of the binary file that has to be flashed.
- **Makefile** - Project's makefile which is copied from project to project. It makes

[4]Project is open-source and publicly available on GitHub [11].

it possible to call `make` directly from projects directory, which eases development process. It does not include any building rules, those are specified in included `rules.mk` file in root directory of the project.

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

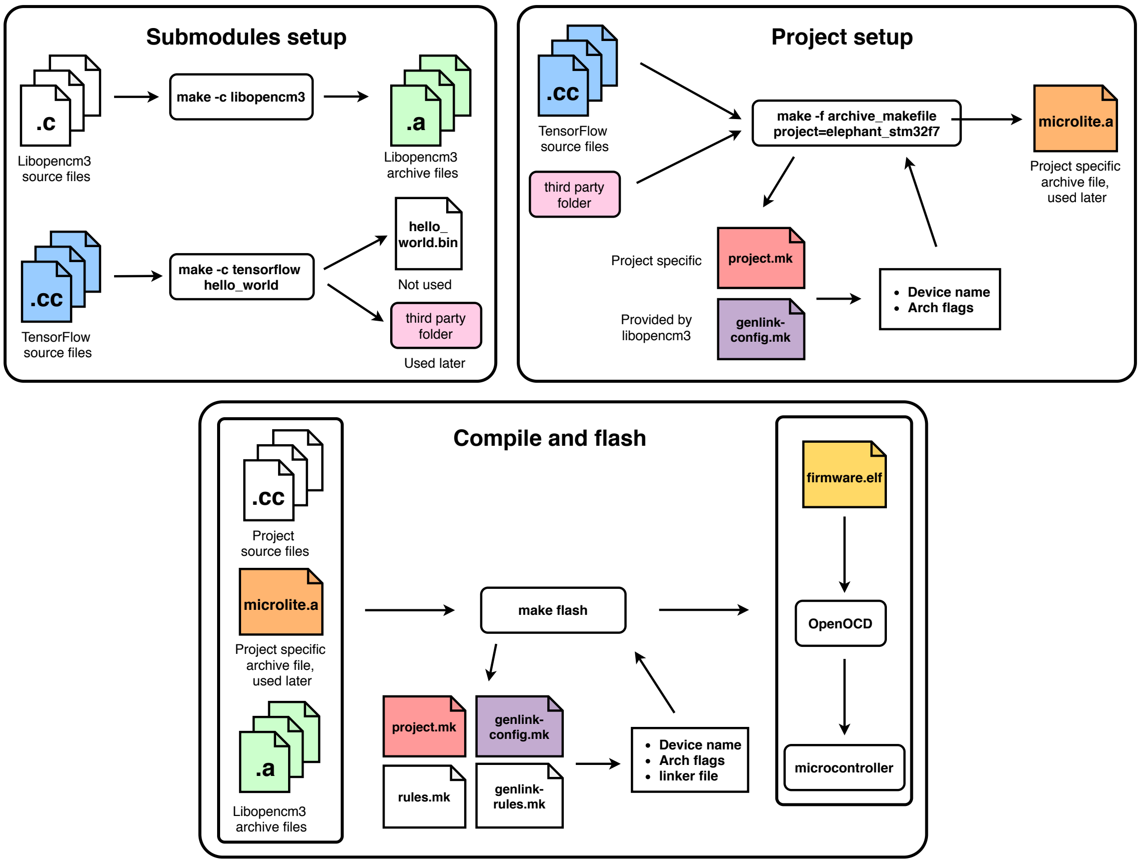


Figure 2.11: Build system of MicroML project.

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

Whenever we start with a new project that will use ML algorithms, we need to go through *project setup* stage. From main directory we call make command with `archive_makefile` and define `PROJECT` variable with the name of our project. `Archive_makefile` looks into `project.mk` and extracts `DEVICE` variable. Libopencm3's `genlink-config.py` script then with the help of `DEVICE` variable determines which compile flags[5]are needed. Afterwards all needed TensorFlow source files and third party files are compiled with this flags and a project specific `microlite.a` archive file is created in our project's folder.

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

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

2.2.4 Running inference on a microcontroller

TFLite Micro API is fairly simple to use and general enough that it can be copied from project to project without many modifications. Figure **??** shows a simplified inference code example, copied from our project. As a first step, we need to define size of `tensor_arena` array, which holds memory input, output, and intermediate arrays. Exact size of `tensor_arena` is determined by trail and error: we set it to some big value and then decrease it in steps, until the code does not work anymore.

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

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

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

In lines 16 to 24 we create an operation resolver. One way to do it is to specify each needed operation specifically (which is done in the example) or simply pull in all operation implementations. Latter approach is however not recommended, as it results in a large binary size. To find out exactly which operations are needed we used online tool Netron [12].

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

In line 47 we finally invoke interpreter and run inference on input data. Whole expression is surrounded with timing functions, which are used to keep track of time spend computing inference.

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

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

2.2.5 Wisent board control firmware

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

2.3 Server side components and software

In this section we describe possible server side construction of various frameworks which enable us to receive LoRaWAN message, parse it, store it in a database and present it. For this thesis we did not implement this specific setup as it was not required for testing purposes, however at IRNAS we use this setup all the time for our IoT products and implementing such system would be trivial.

System that we use consists of different components, each one with a distinct task. Tools that we use are The Things Network (TTN), Node-RED, InfluxDB and Grafana. Flow of information and tasks of each tool are presented on Figure 2.13.

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

Node-RED functions as a glue logic that parses packets and shapes them into format that is required by InfluxDB. Node-RED provides a browser-based flow editor, where actual programming can be done graphically. Logic is programmed by choosing different blocks called *nodes* and connecting them together. This is convenient, as Node-RED provides different nodes for communicating with different technologies, such as MQTT, HTTP requests, emails, Twitter accounts and others. In our use case we need to use a combination of nodes seen on Figure 2.14 Node *Elephant Gateway* is connected to a specific application on TTN, which is used for collection of packets from our device in field. Any packet that will appear in that TTN application will also appear in Node-RED. Node *Parse packet* extracts information contained in each packet and stores it in a specific format, which is finally send to node *Elephant Database.*

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

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

Example of Grafana graph can be seen on Figure 2.15.

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

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

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

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

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

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

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

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

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

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

```

Figure 2.12: Example of TensorFlow Lite inference code in C++.

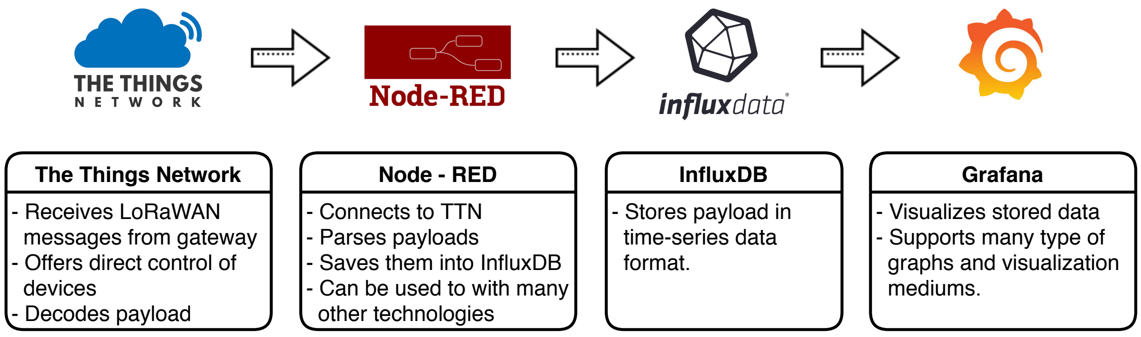

THE THINGS NETWORK
Node-RED
influxdata
The Things Network
- Receives LoRaWAN messages from gateway
- Offers direct control of devices
- Decodes payload
Node - RED
- Connects to TTN
- Parses payloads
- Saves them into InfluxDB
- Can be used to with many other technologies
InfluxDB
- Stores payload in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

Figure 2.13: Server side flow of information. Icons source: [5]

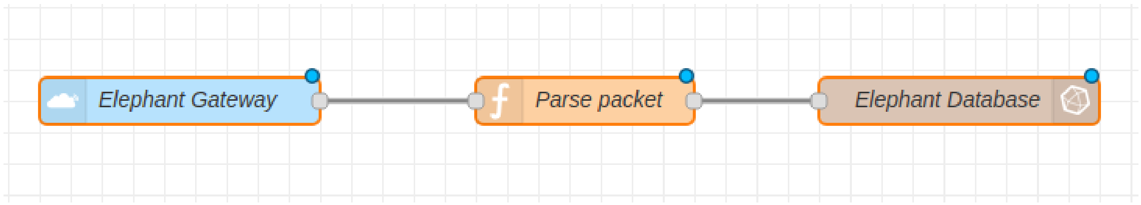


Figure 2.14: Node-RED flow.

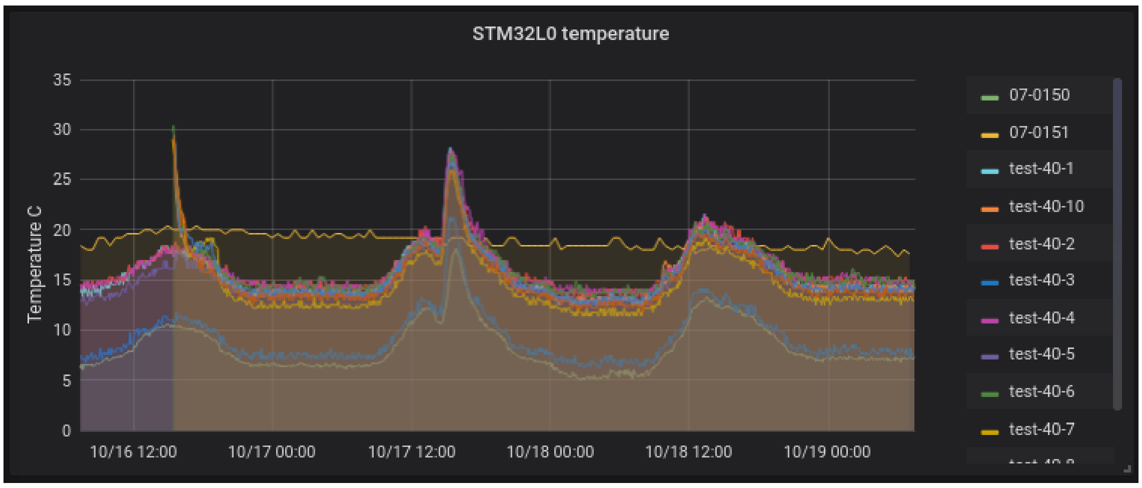


Figure 2.15: Example of Grafana graph.

Bibliography

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

[2] 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].

[3] 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].

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

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

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

[7] Geron, A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems, 2nd edition.*

O'Reilly Media, Sebastopol, CA, 2019.

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

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

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

[11] 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].

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