Contents

1 Theoretical description of system building blocks

1.1 Machine learning

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

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

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

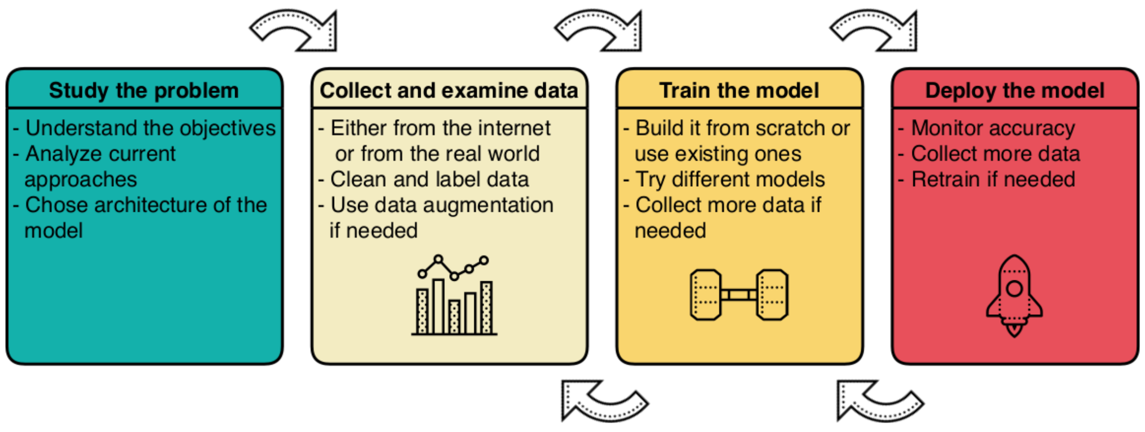
Algorithms like k-Means, Expectation Maximization, Principal Component Analysis fall into the category of unsupervised learning algorithms. Here training data is unlabeled, algorithms are trying to find similarities in data by itself [?]. There exist other categories such as semi-supervised learning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

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

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

1.1.1 General machine learning workflow

There are several steps in ML workflow that need to happen in order to get from an idea to a working ML based system, this is represented on Figure 1.1.



Study the problem
- Understand the objectives
- Analyze current approaches
- Chose architecture of the model
Collect and examine data
- Either from the internet or from the real world
- Clean and label data
- Use data augmentation if needed
Train the model
- Build it from scratch or use existing ones
- Try different models
- Collect more data if needed
Deploy the model
- Monitor accuracy
- Collect more data
- Retrain if needed

Figure 1.1: Workflow of solving a generic machine learning problem. Icons source: www.icons8.com

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

1.1.2 Machine learning on embedded devices

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

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

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

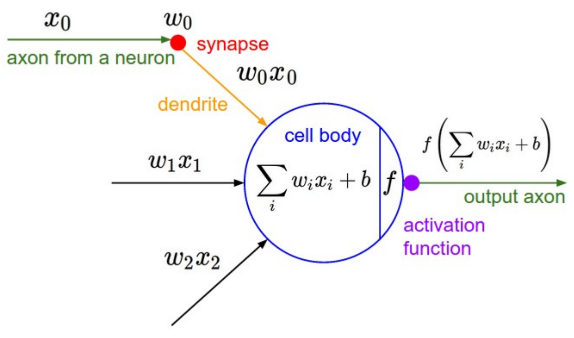
Such benefits do come with some drawbacks. Embedded devices are a more resource constrained environment when compared to personal computers or servers. Because of limited processing power, it is not feasible to train ML models directly on microcontrollers. Also it is not feasible to do online learning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

Usual workflow, while developing machine learning models for microcontrollers, is to train a model on training data on a computer. When we are satisfied with the accuracy of the model we quantize it and convert it into a format understandable to our microcontroller. This is further described in section **??**.

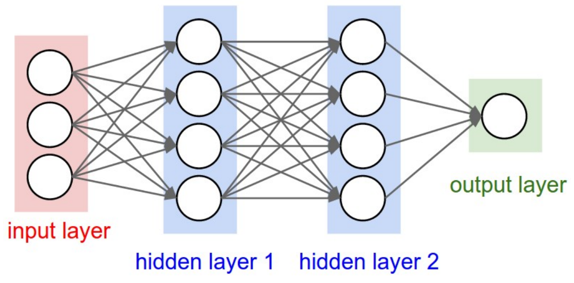
1.2 Neural networks

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

To learn how NNs work we can refer to Figure 1.2a, which shows a generic version of an artificial neuron.



(a) Artificial neuron



(b) 3-layer neural network

Figure 1.2: (a) Mathematical model of an artificial neuron, similarities with biological neurons can be seen. (b) Fully connected 3-layer neural network. Image source: [?]

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

NNs consist of many neurons, which are organized into **layers**. Neurons inside the same layer do not share any connections, but they connect to layers before and after them. First layer is known as **input** layer and last one is known as **output** layer. Any layers between are said to be **hidden**. On Figure 1.2b we can see neural network with an input layer with three inputs, two hidden layers with four neurons each and a output layer with just one neuron. If all inputs of neurons in one layer are connected to all outputs from previous layer, we say that a layer is **fully connected** or **dense**, Figure 1.2b is an example of one. NNs with many hidden layers fall into category of deep neural networks (DNN).

1.2.1 Activation functions

Activation functions introduce non-linearity to chain of otherwise linear transformations, which enables ANNs to approximate any continuous function [**?**]. There are many different kinds of activation functions as seen on Figure 1.3, such as sigmoid function and rectified linear activation function (ReLu). Sigmoid function was commonly used in the past, as it was seen as a good model for a firing rate of a biological neuron: 0 when not firing at all and 1 when fully saturated and firing at maximal frequency [**?**]. It basically takes a real number and squeezes it into range between 0 and 1. It was later shown that training NNs with sigmoid activation function often hinders training process as saturated outputs cut of parts of networks, thus preventing training algorithm reaching all neurons and correctly configuring the weights [**?**]. It has since fallen out of practice and is nowadays replaced by ReLu or some other activation function.

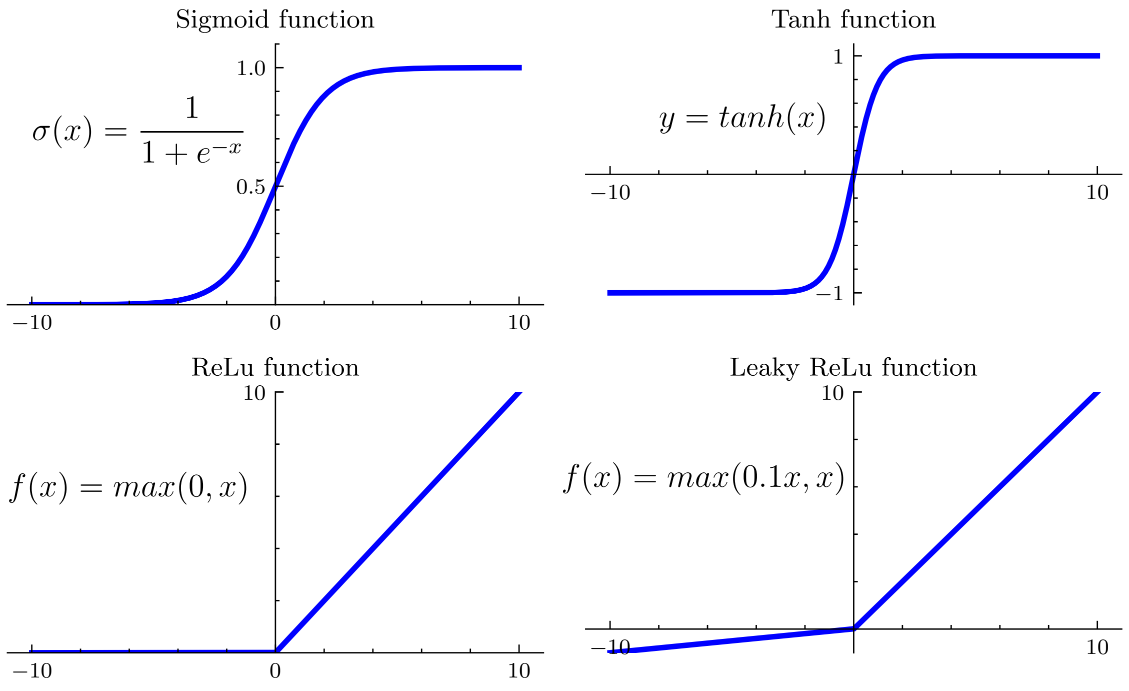


Figure 1.3: Different activation functions and their equations.

1.2.2 Backpropagation

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

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

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

1.2.3 Convolutional neural networks

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

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

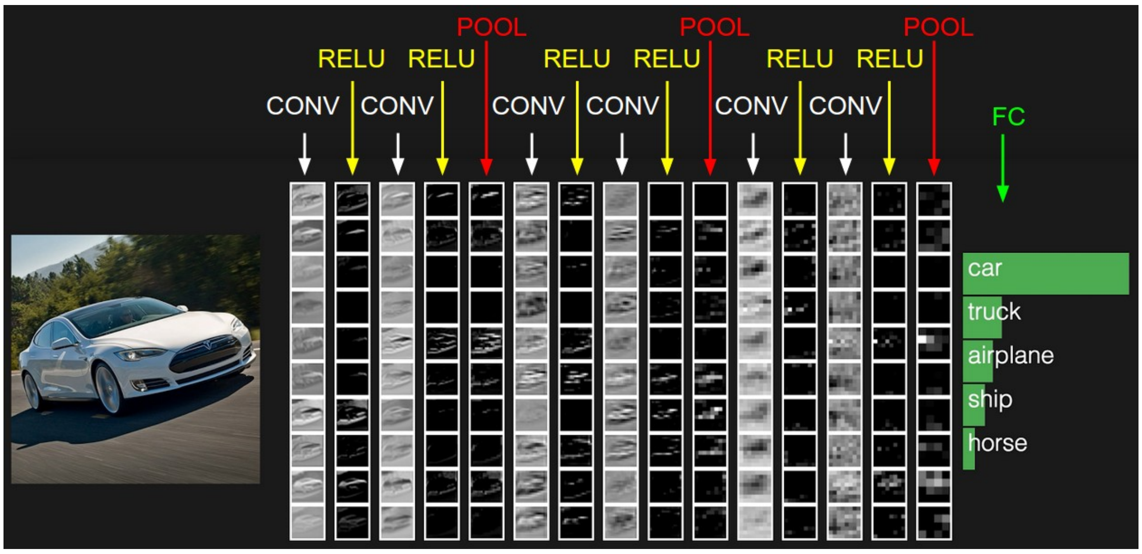


Figure 1.4: CNNs usually consist of alternating convolutional layers and pooling layers. Last polling layer is flattened out and feed into fully connected NN. Image source: [**?**]

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

[1]Scientists Weisel and Hubel showed that different cells in primary visual cortex of a cat responded differently to different visual stimuli [**?**]. Some were activated when shown a horizontal line in specific location, some were activated by vertical lines. More complex cells responded to boxes, circles and so on. CNNs also detect simpler shapes first and use them to detect more complex ones later.

probability results in given classes.

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

1.2.3.1 Convolutional layers

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

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

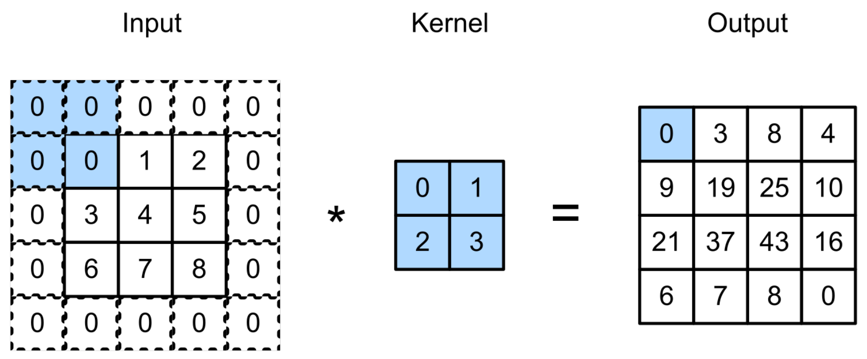


Figure 1.5: Dot product operation between filter and zero-padded input matrix. Image sources: [?]

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

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

$$V_o = (V_i - F + 2P)/S + 1 \tag{1.1}$$

Where:

V_i - Input volume size, only width or height

V_o - Output volume size, only width or height

F - Filter or receptive field size

P - Amount of zero padding used on the border

S - Stride length

If we examine example on Figure **??** we can see that input with a size 3 x 3, stride 1, padding 1 and filter with a size 2 x 2 produces output with size a 4 x 4.

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

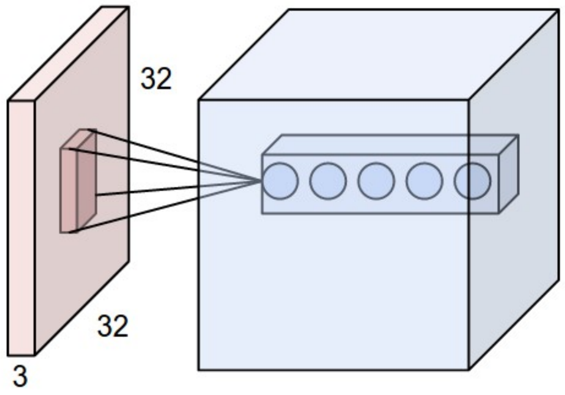
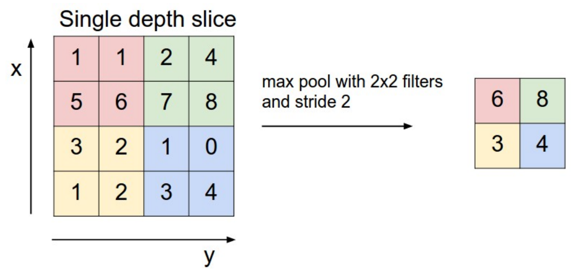


Figure 1.6: Convolutional layer with five different filters. Image sources: [?] [?]

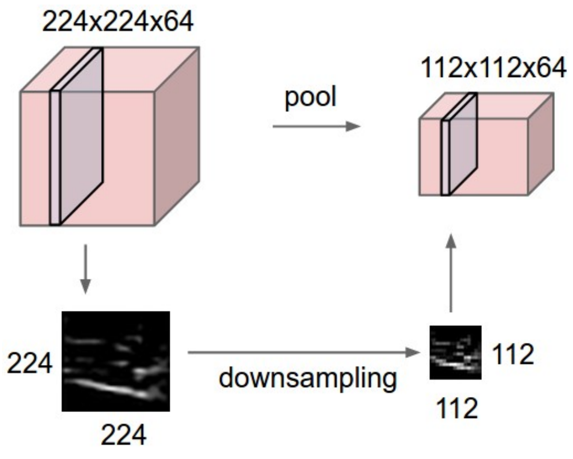
In the end of convolutional layer output volume is fed into neuron similar to one described in section 1.2. All elements in same depth are affected by a same bias term and fed into activation function. In this step, size of volume is preserved.

1.2.3.2 Pooling layers

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



(a) Max pooling operation



(b) Effect of polling on input dimensions

Figure 1.7: Polling layer. Image source: [?]

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

1.3 TensorFlow

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

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

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

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

1.3.1 TensorFlow Lite for Microcontrollers

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

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

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

Is important to know that TFLite is just an extension of existing TensorFlow project. General steps for creating a trained ML model are still the same as seen in Figure 1.1, although we have to be aware of some details. Figure ?? shows all steps that are needed to prepare a ML model for running on a microcontroller.

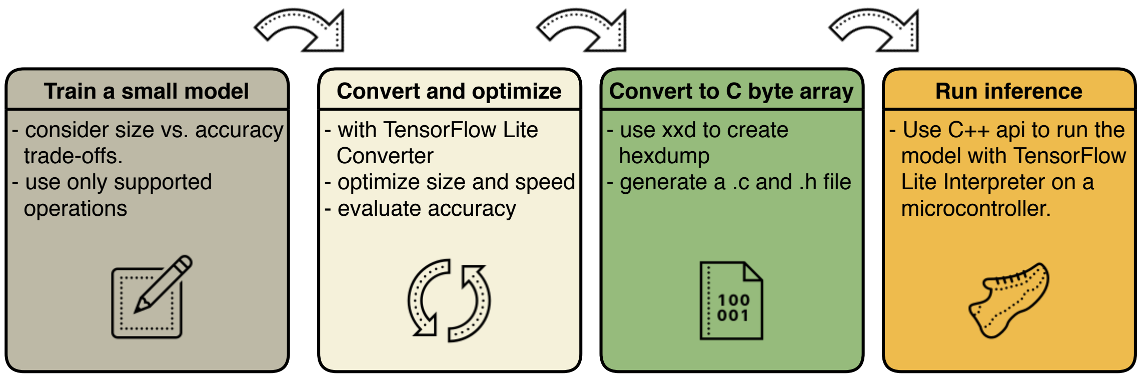

Train a small model
- consider size vs. accuracy trade-offs.
- use only supported operations
Convert and optimize
- with TensorFlow Lite Converter
- optimize size and speed
- evaluate accuracy
Convert to C byte array
- use xxd to create hexdump
- generate a .c and .h file
100
001
Run inference
- Use C++ api to run the model with TensorFlow Lite Interpreter on a microcontroller.

Figure 1.8: Workflow of preparing a ML model for an inference on a microcontroller. Icons source: www.icons8.com

We start with a small but inaccurate model that can still accomplish basic criteria that our objective demands. When the end of this workflow process is reached and we made sure that the model can fit into a flash memory area of our target microcontroller, we can start adding more layers in order to increase accuracy. We are allowed to use only operations that have supported implementations on microcontrollers. This is usually not a restriction as many of them are supported. Model that we created is usually saved in Hierarchical Data Format with extension h5. Model in this shape is usually very big and needs to be converted with TensorFlow Lite Converter tool. This tool can be either used as a command line tool or as a function in a Python script. We input model in .h5 file, choose if and which specific optimisation we want to use and as an output we receive our model in a .tflite format. After we created and evaluated our .tflite model we need to convert it into a format that is understandable to a microcontroller. This is done with the **xxd**, a Linux command line tool which creates a hex dump out of any input file. As a input parameter we give xxd our .tflite model, set the -i flag and save the output into a .c file. Xxd tool will then create a hex dump of our model and format it as a char array in C programming language. We then create a .h file with a declaration of char array which we can later call from our application code. Model is then ready to be executed on a microcontroller, we can run it and process the results. Accuracy will be the same as compared to running the same .tflite model on a personal computer, but execution time will naturally be different. If needed we can tweak the model parameters, train a new model and repeat described workflow again.

1.3.1.1 Post-training quantization

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

1.4 IoT and wireless technologies

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

Important part of IoT system is a wireless network that is used to transport data from edge devices to gateways or directly to the internet. Choice of a wireless network is highly depended on a type of a problem a IoT solution is trying to solve. Factors such as required battery life, amount of data being sent, distance that data has to travel and environment conditions of edge device itself are important.

Because our early detection system demands a decent battery life of several months and needs to send a small amount of data over one or two kilometers we will focus on wireless technologies such as NB-IoT, Sigfox and LoRa.

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

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

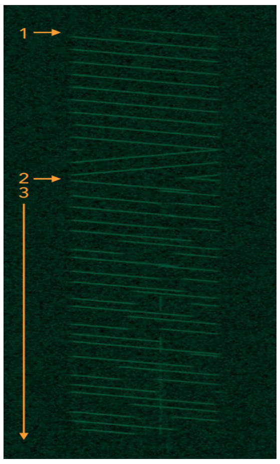
This leaves us with Lora protocol which covers our use case from points of long range, low power consumption and ability to setup our own network.

1.4.1 LoRa and LoRaWAN

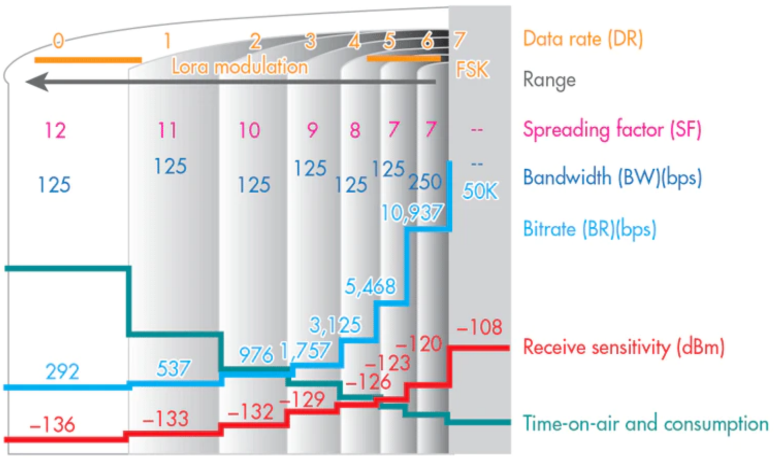
LoRa (Long Range) is a physical layer protocol that defines how information is modulated and transmitted over the air [?] [?]. Protocol is proprietary and owned by a semiconductor company Semtech, who is as sole designer and manufacturer Lora radio chips in the world. LoRa protocol uses a modulation similar to chirp spread spectrum modulation [?]. As the protocol is proprietary exact details of it are not known, however it was reverse engineered by radio frequency specialist [?]. Example of a LoRa signal that was captured with a software defined radio can be seen on Figure ??. Each symbol is modulated into a radio signal whose frequency is either increasing or decreasing with constant rate inside of a specified bandwidth. When bandwidth boundary is reached, signal ”wraps around” and appears at the other boundary. Although frequency is always changing with constant rate, it is not continuous inside the bandwidth window, but it can immediately change to a different frequency and continue from there.

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

Size of a bandwidth window, rate of frequency change (also known as a spreading factor) and transmitting power further define LoRa signal. With these factors we can influence range, power consumption and bit rate of a LoRa signal. For example,



(a) LoRa signal



(b) Properties of LoRa signal

Figure 1.9: Lora signal (left) and different properties of LoRa with their effects on range, bit rate, receiver sensitivity, time on air and consumption (right). Image sources: [?] [?]

as seen on Figure ??, by increasing spreading factor we increase time on air thus giving the receiver more time to sample signal, which leads to better sensitivity but increases power consumption.

While LoRa defines the physical layer, LoRaWAN defines media access control protocol for wide area networks, which are built on top of LoRa [?]. Its specification is open, so any one can implement it. LoRaWAN takes care of communication between end devices and gateways and manages communication frequency bands, data rate and transmitting power.

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

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

1.5 Thermal cameras

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

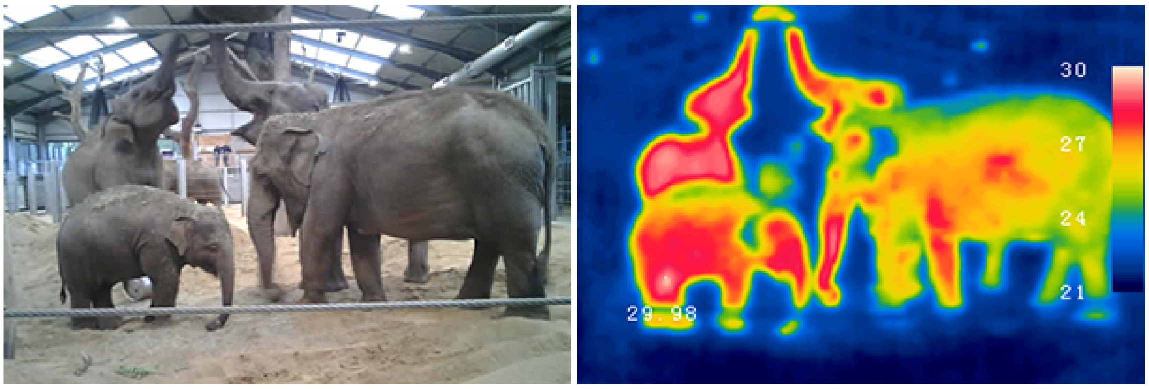
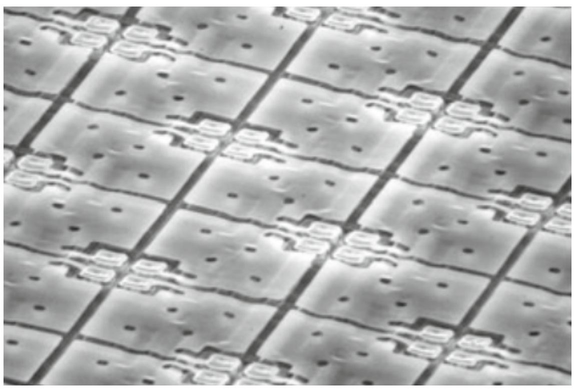
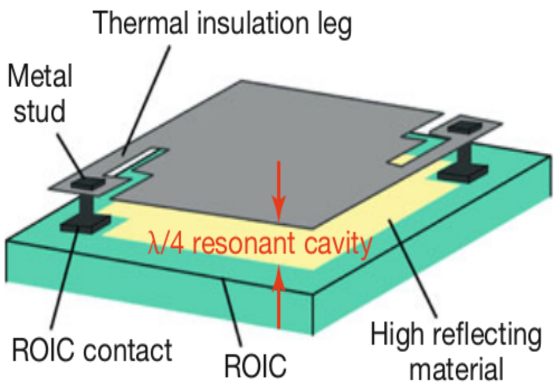


Figure 1.10: Comparison between a picture taken with a normal camera (left) and image taken with a low resolution thermal camera FLIR Lepton 2.5 (right). Image source: Arribada Initiative [?]

Common examples of thermal detectors are thermopiles and microbolometers. Thermopiles are composed of several thermocouples. Thermocouples consists of two different metals joined at one end, which is known as hot junction. Other two ends of the metals are known as cold junctions. When there is a temperature difference between the hot and cold junctions, voltage proportional to that difference is generated on open ends of the metals. To increase voltage responsivity, several thermocouples are connected in series to form a thermopile [?]. Thermopiles have lower responsivity when compared to microbolometers, but they do not require temperature stabilization [?].

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


(a) Focal point array



(b) Bolometer

Figure 1.11: (a) Focal point array under electronic microscope. (b) Bolometer with $\lambda/4$ resonant cavity. Image source: Vollmer, Möllmann [?]

Absorber material is made either out of metals such as gold, platinum, titanium or more commonly out of semiconductors such as vanadium-oxide (VOx) [?]. Important property of absorber materials is that electrical resistance changes proportionally with material's temperature [?]. When IR radiation hits absorber material, it is converted into thermal energy, which raises absorber's temperature, thus changing its resistance. To detect change in resistance, ROIC applies steady-state bias current to absorber material, while measuring voltage over conductive legs [?].

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

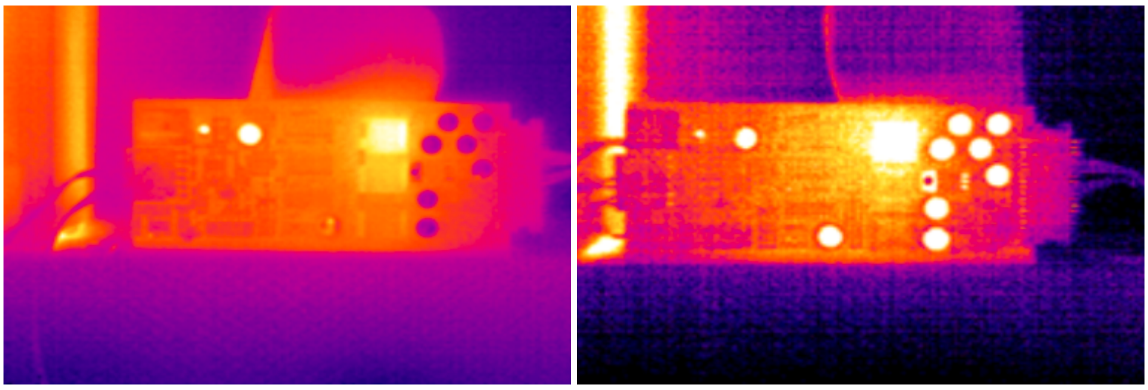

(a) NETD is 60 mK (b) NETD is 80 mK

Figure 1.12: Comparison of images of the same object taken with cameras with different NETD values. Low NETD values are more appropriate for object recognition. Image source: MoviTherm [?]

1.5.1 Choosing the thermal camera

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

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

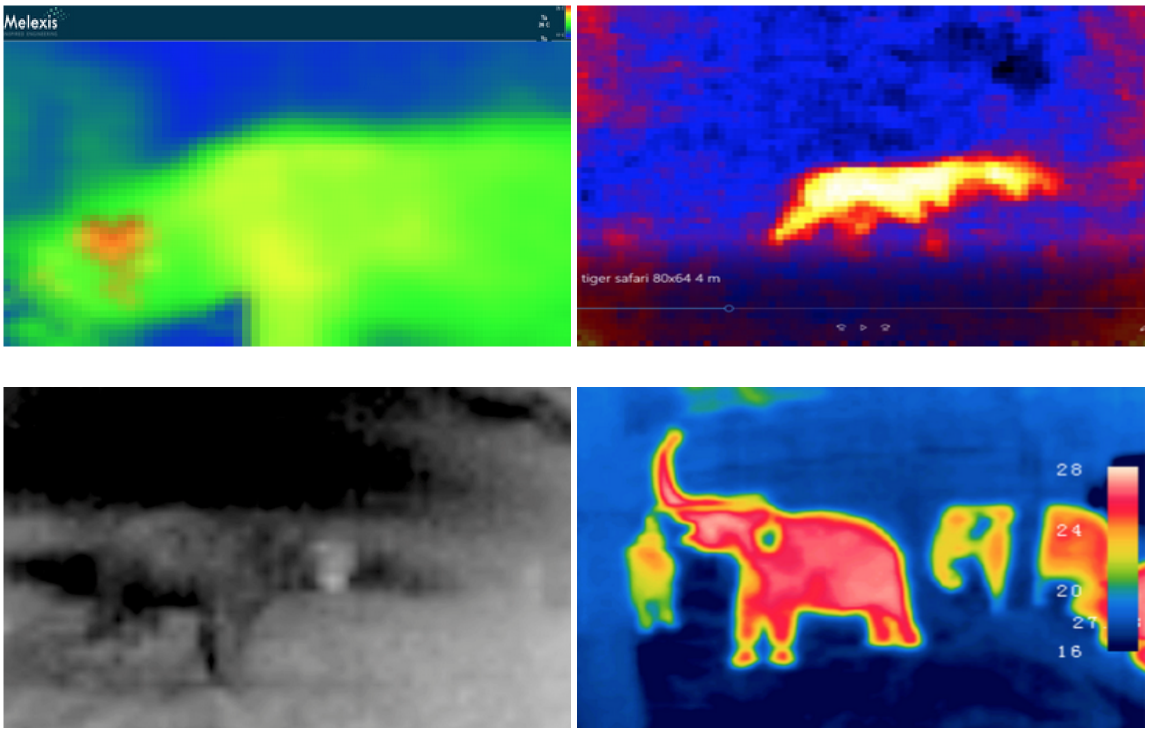


Figure 1.13: , MELEXIS MLX90640 (top left), Heimann 80x64 (top right), ULIS Micro80 Gen2 (bottom left) and FLIR Lepton 2.5 (bottom right). Image source: Arribada Initiative [?]

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

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

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

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

2 Neural network model design

In this chapter we will describe the design of convolutional neural network that will be able to process thermal images and predict what object they contain. Workflow that we will follow will largely be a combination of workflows presented in Figures 1.1 and **??**.

We will first set concrete objectives, which will dictate what exactly we want to accomplish, while keeping in consideration various constraints. We will then explore the dataset provided by Arribada Initiative, analyze different class representations and decide if it is appropriate for accomplishing objectives that we set earlier. We will decribe tools and development eviroment that we used in the process.

In image preprocessing phase parsed a large excel table and connected images with correct metadata. Image preprocessing such as mean normalization and augmentation was then done. We then designed and trained few CNN models with varying complexities and observed how well they behave on thermal image dataset. We then optimized models with TFLite tools and compared accuracies and model sizes. Models were then converted into a C char array format, ready to be tested on a actual microcontroller.

We will finish this chapter by describing the same workflow, but by using tools that Edge Impulse provides to the point where we will have our model ready to run on a microcontroller. Using Edge Impulse will generally require less knowledge, less time and will lead to better results when compared to our setup.

2.1 Objectives

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

It would be beneficial if thermal camera can take several pictures of the same object, thus increasing the confidence of computed label of the object. However this is constrained by the image processing time and camera's field of view. Thermal camera FLIR Lepton has a horizontal field of view of 57 degrees. The closer elephant passes by the thermal camera the quicker he will traverse the camera's field of view, thus giving the camera less time for capture. This problem can be solved by optimizing the execution time of the ML model or by placing the early detection system on far enough position from expected elephant's path. As 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 will be deployed on a microcontroller and not on a computer or a server we have to keep it simple and small. Extra model complexity that might bring us few percents of accuracy will not matter much, if our model would be too large to fit on a microcontroller or too slow 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 possible 4 categories: elephant, human, cows and nature/random.
- Total image processing time should be as short as possible, we should try to keep it under 2 seconds.
- Model should be small enough to fit on a microcontroller of our choice, while still leaving some space for application code. Microcontroller of our choice (STM32F767) has 2 MB of flash memory so model size should be smaller than that.

2.2 Exploring the dataset

As mentioned in section **??** thermal image dataset was provided by Arribada Initiative [?] [?]. Images in dataset came 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 otu from 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 deployed camera can be seen on Figure **??**.

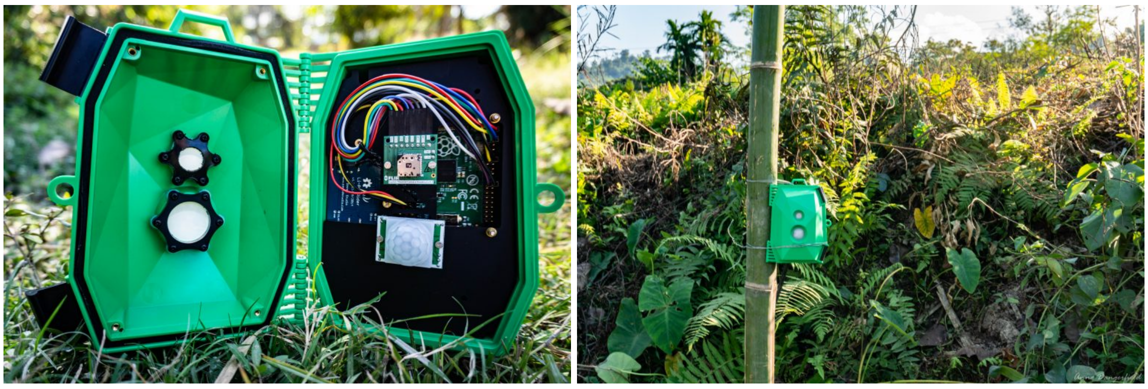


Figure 2.1: Camera trap used in Assam, India. Image source: Arribada Initiative [?]

PIR sensor functioned as photo trigger, whenever an object passed in front of it camera made a picture. This setup provided Arribada with elephant images in real life scenarios, however they could not capture elephants in variety of different conditions, such as different angles and distances.

This was accomplished in ZSL Whipsnade Zoo, where they could take many images of elephants in variety different conditions [?]. 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, with better resolution.

Images of elephants that came from both locations can be seen on Figure ??.

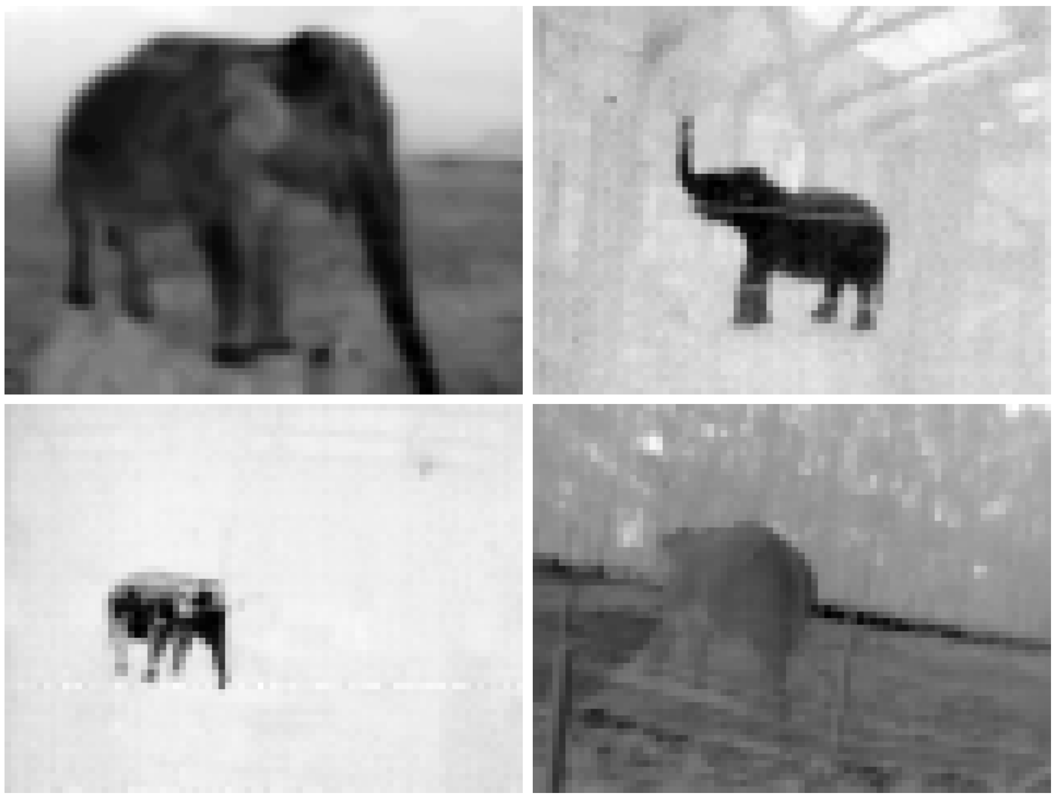


Figure 2.2: Thermal images of elephants from dataset.

Thermal image dataset 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.

Firstly, we saw that the primary focus of Arribada team was to build an object localization model, not an image classification model. In object localization neural network draws bounding boxes around objects that it recognizes and assigns them labels, while image classification model only labels the image as whole. Object localization produces a bigger and more complex model then image classification and it is unsuitable for running on a microcontroller. All major work that was done by Arribada team was contained in one folder where each image had 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 a 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 few folder 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. All folders with elephant pictures we merged into one folder, as we did not care if 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.

Secondly, 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 image file name with information where image was taken and with what sensor. This enabled us to generate a graph seen on Figure **??**. We used a total of 13667 images from 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 2.5 version of the thermal camera.

Thirdly, after manually inspecting the folder with goat images we saw that it mostly contained images of a goat herd, 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 case of an image classification model, this sort of training data is not desirable, as it would be too similar to another separate class, in our case elephant class. We therefore dropped goat images out of our training data entirely.

Additionally, we realised that there was a large class imbalance, as seen on Figure **??** in favor to elephant class. Number of elephant images was more than 4 times

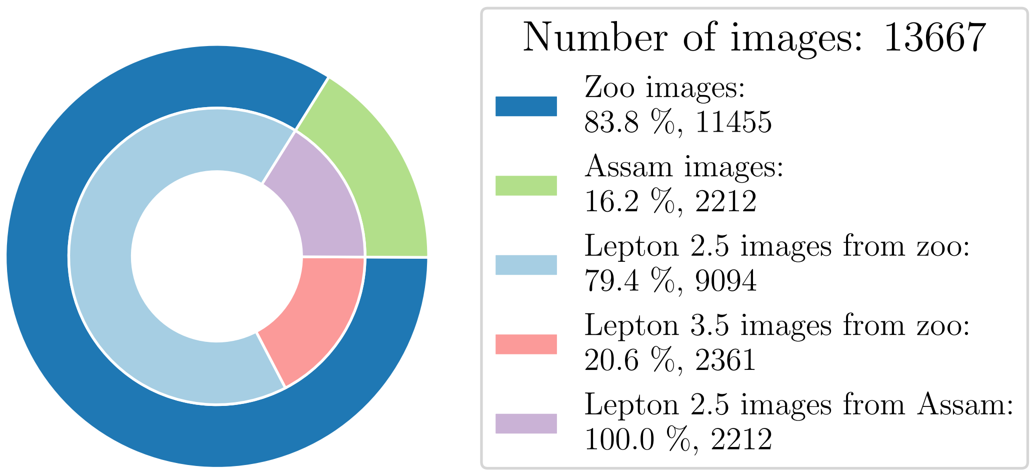


Figure 2.3: Distribution of used images depending on image location and type of sensor.

larger from the number of images of the all other classes combined. This issue was solved with acquiring more images of the minority class, oversampling the minority class and/or undersampling the majority class and additionally augmenting images in preparation phase.

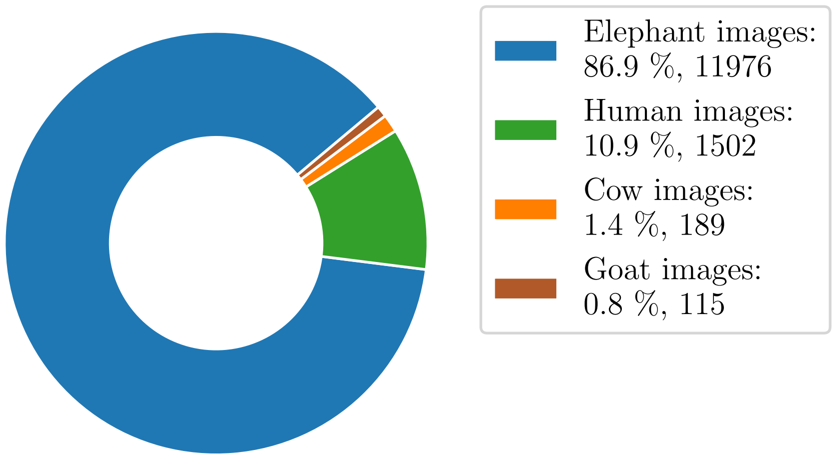


Figure 2.4: Class distribution of thermal images.

2.2.1 Gathering thermal images

As the amount of cow images was low compared to human and elephant classes and because we also did not had any images that would fit into nature/random class, we decided to gather them ourselves. We wanted to do this 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 [?] for 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. Image of the setup can be seen on Figure **??**.

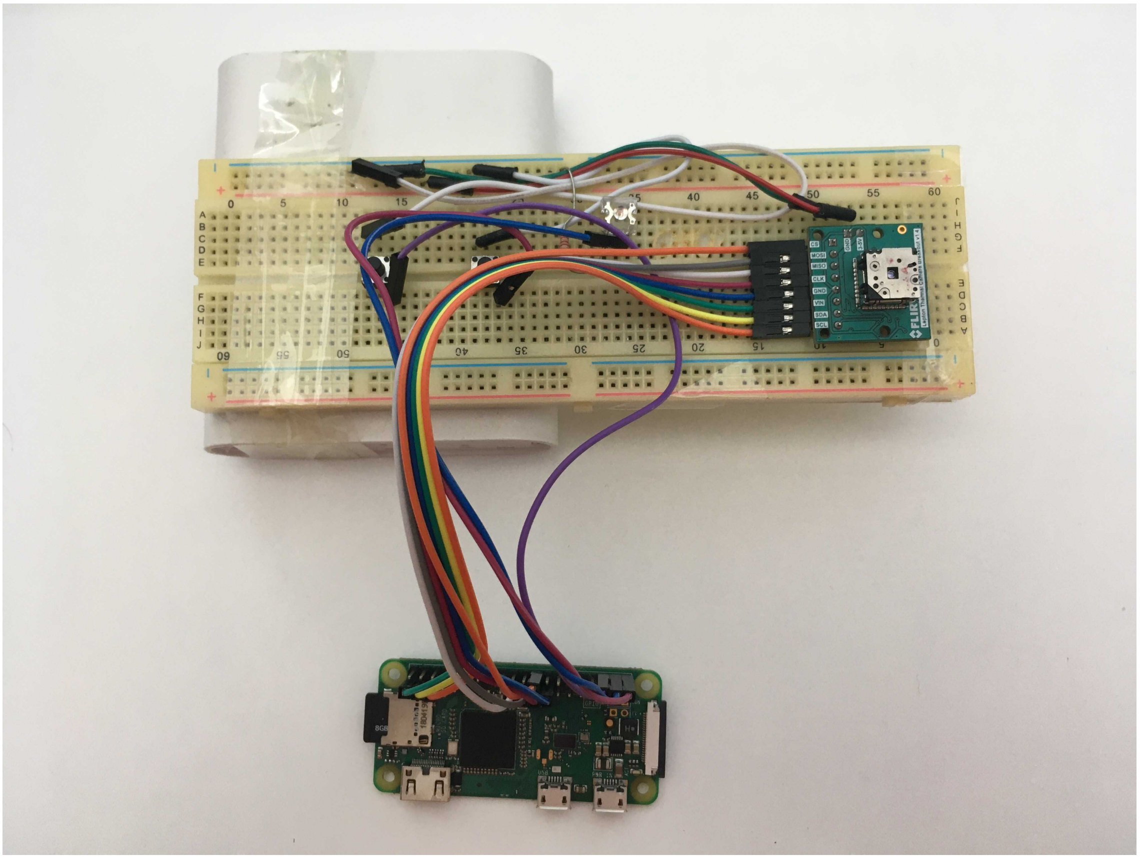


Figure 2.5: Camera setup used for taking thermal images with FLIR Lepton 2.5.

We wrote a simple Python script which executed C program every time we pressed the trigger push-button. Additional shutdown button was added to call the Raspberry Pi shutdown routine, as forcibly removing power from it would corrupt freshly taken thermal images on the Raspberry Pi's SD card.

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

2.3 Tools and development environment

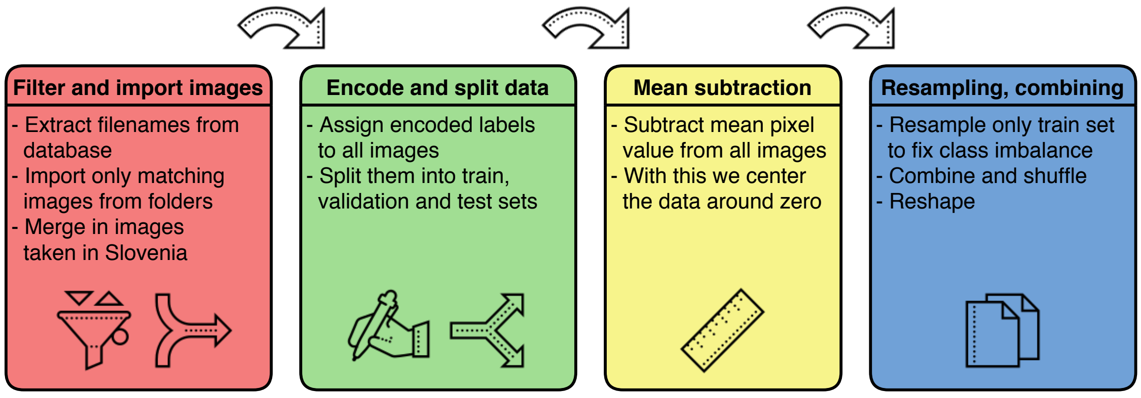
All of the work around image preparation and ML model creation was done in Python 3.6. Python module Numpy was used for image preprocessing, Pandas for Excel database manipulation and Matplotlib for plot generation. Neural networks were designed in TensorFlow 2.4, using Keras high level API.

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

Instead of writing Python scripts and executing them through 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 web browser by writing the IP address of the server, followed by the default port that Juptyer Notebook server uses.

2.4 Image preprocessing

Image preparation phase is a pipeline process that differs from project to project. Our process can be seen on Figure **??**.



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 train, 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 train set to fix class imbalance
- Combine and shuffle
- Reshape

Figure 2.6: Image preparation pipeline that was used. Icons source: www.icons8.com.

At the start of the process we compared filenames of each separate folder to the list of filenames found in 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 camera with different resolutions (60 x 80 and 120 x 160), we just downscaled higher resolution directly in the importing process. After this we simply added images that were taken by us in Slovenia. At this point we had four separate Numpy arrays, one for every class, with 3 dimensions: first dimension was a number of different images in that class, second and third dimensions stored image's pixel values (60 and 80 pixels respectively).

Next step was assigning labels to every 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 elephant, 1 for human, 2 for cow and 3 for nature/random class. We shuffled images inside of each class and then split them into training, validation and test sets. 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 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 pipeline, which is usually combined with standardization [?]. 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 [?]. Because images that were created with FLIR camera were all 8-bit encoded, therefore same range, this was not needed.

In the end we had to fix our imbalance problem. We achieved this by resampling the human, cow and nature/random classes. Human class was resampled three times, while both cow and nature/random classes were resampled six times. Figure **??** shows distribution of training images before and after resampling.

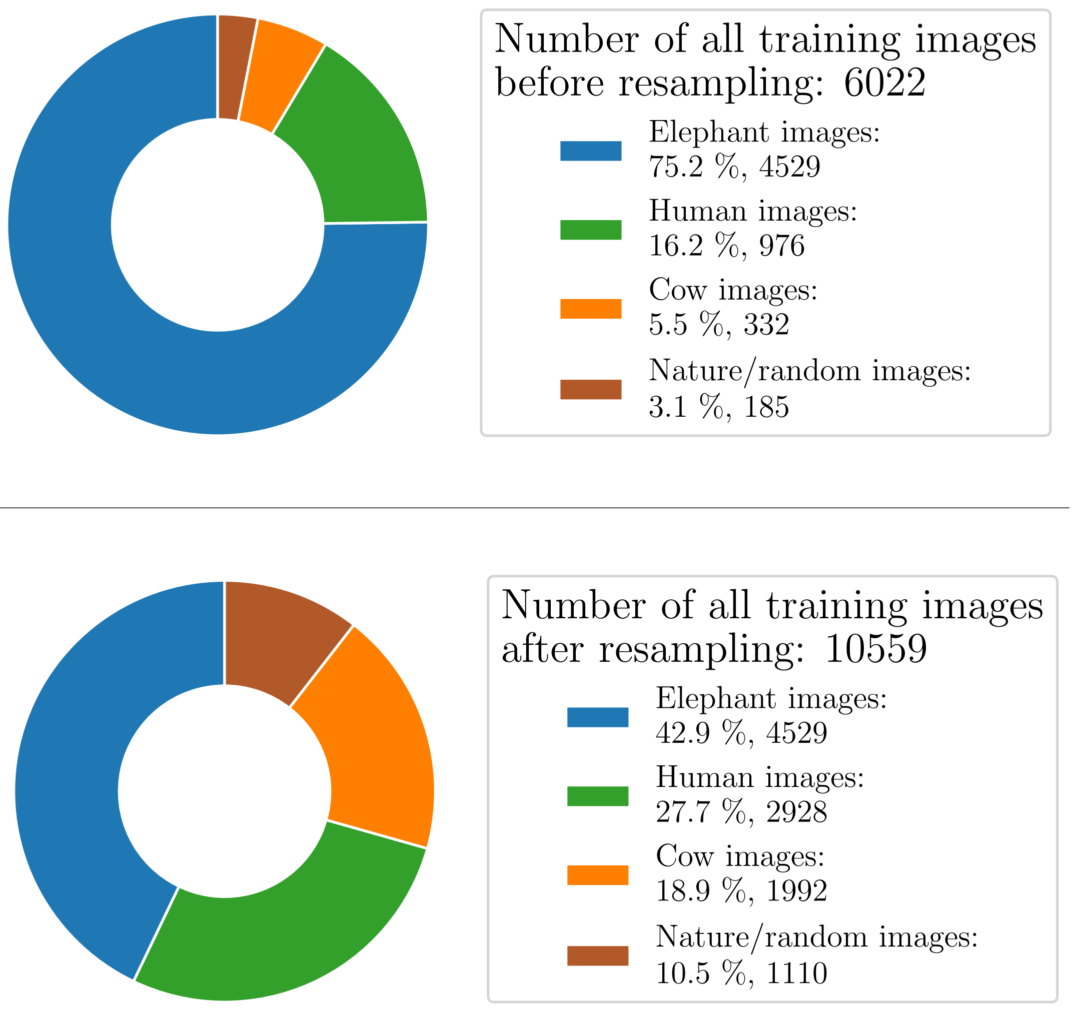


Figure 2.7: Distribution of training images before and after resampling.

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

After resampling we merged and shuffled all data and reshaped it with Numpy from three dimensions to four, so it was ready for model training.

2.5 Model creation and training

For creating CNN models we used Keras Sequential API. With it programmer creates a model and adds layers one by one. When adding layers we are only specifying what type of layer we went, size of it and layer specific features. We do not need to keep track of any connections between or in layers, this is done automatically by Keras. Specific architecture of models that we created can be seen in section TODO ADD REFERENCE, however in Figure **??** we can see Keras code and summary of the model that it created.

Hyperparameter selection procedure, random search? * Each hyperparameter of model is described. * Epoch, batch size, early stopping

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
        input_shape=(60,80, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(4))

# Show the model's architecture
model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 58, 78, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 29, 39, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 27, 37, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 18, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 16, 64)        36928
_________________________________________________________________
```

```
flatten (Flatten)            (None, 11264)            0
_________________________________________________________________
dense (Dense)                (None, 64)               720960
_________________________________________________________________
dense_1 (Dense)              (None, 4)                260
=================================================================
Total params: 776,964
Trainable params: 776,964
Non-trainable params: 0
_________________________________________________________________
```

2.6 Model optimization

For model optimization phase we wrote a Python script that took model in .h5 format and converted it 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. Model size is split in half and 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 a 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 representative dataset at time of optimization.

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 implementations in TFLite Micro. Script also uses tool xxd to create .c and .h files of models in c array format, for later inference on microcontroller.

2.7 Neural network model design in Edge Impulse Studio

Notes and remarks from other ML thesis: * You could rework section development enviroment to be more concise. No need to describe everything in such detail, but you need to mention all tools. * You did not mention data split, what was ratio and why are they used