Contents

1 Theoretical description of system building blocks

1.1 Machine learning

According to Arthur Samuel (qtd. in Geron [1]) 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 the terms "machine learning" and "learning" interchangeably. To learn, these learning algorithms need to be trained on a collection of examples of some phenomenon [2]. These collections are called **datasets** and can be generated artificially or collected in nature.

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 that contains 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 [2]. 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 the 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 the learning algorithm needs in the training process. Algorithms like K-nearest neighbors, 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** [1]. 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 [1]. There exist other categories such as semi-supervised learning which is a combination of the previous two and reinforcement learning, where model acts inside the environment according to learned policies [1].

Neural networks, algorithms inspired by neurons in human brains [1] [3], 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 the field of deep learning, which is a sub-field of machine learning.

Training of ML models is computationally demanding and is usually done on powerful servers or computers with dedicated graphic processing units to speed up training time. After a model has been trained, data can be fed in and prediction is computed. This process is also known as **inference**. The inference is computationally less intensive compared to the training process, so with properly optimized 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 to get from an idea to a working ML based system, this is represented in Figure 1.1.

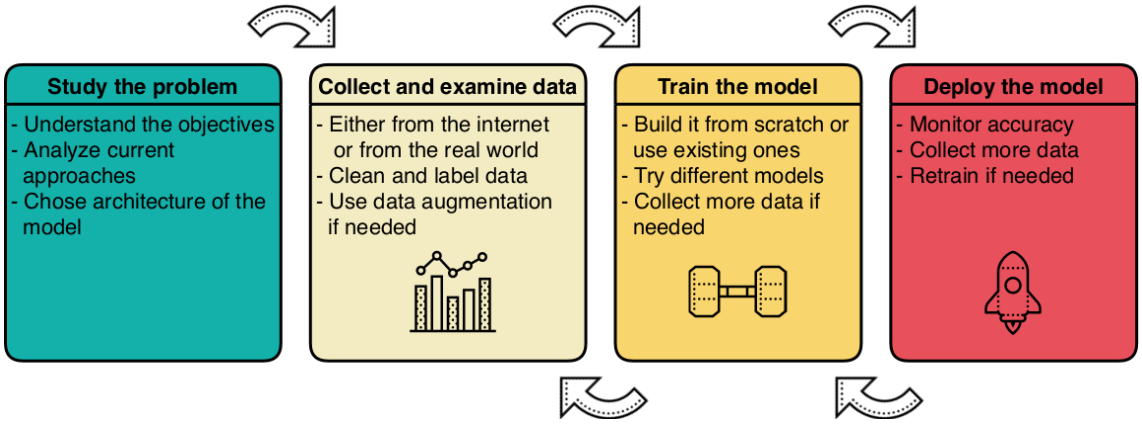


Figure 1.1: Workflow of solving a generic machine learning problem. Icons source: www.icons8.com

First, the 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 the second step we collect and clean up data. We should always strive to collect a large amount of quality and diverse data that represents real world phenomenon. Collecting that kind of data can be hard and expensive, but we can use various tools 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 the 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 the 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 [4] [5] [6].

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 [6]. 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 [6] 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.

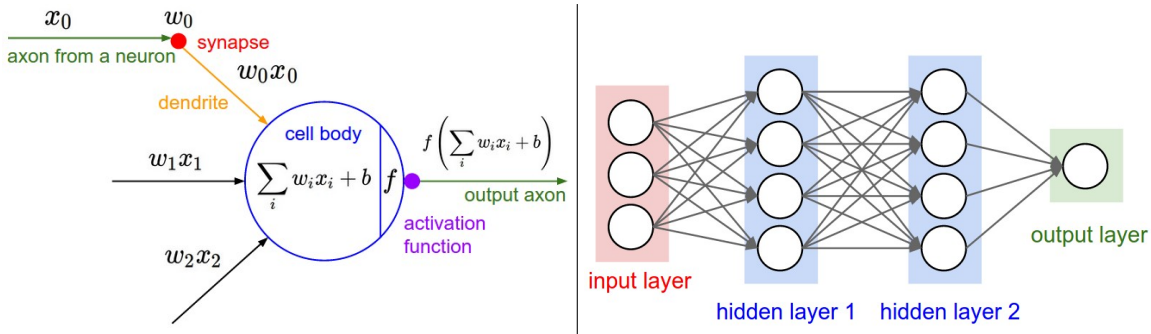
Such benefits do come with some drawbacks. Embedded devices are a more resource constrained environment when compared to personal computers or servers. Because of limited processing power, it is not feasible to train ML models directly on microcontrollers. Also it is not feasible to do online learning 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 [7]. To make it fit on a microcontroller and still have space for our application, it would have to be simplified.

The usual workflow, while developing machine learning models for microcontrollers, is to train a model on training data on a computer. When we are satisfied with the accuracy of the model we quantize it and convert it into a format understandable to our microcontroller. This is further described in section 1.3.1.

1.2 Neural networks

Although the first models of neural networks (NN) were presented in 1943 (by McCulloch and Pitts) [1] 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 [1].

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: [3]

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**. In Figure 1.2b we can see a neural network with an input layer with three inputs, two hidden layers with four neurons each and an output layer with just one neuron. If all inputs of neurons in one layer

are connected to all outputs from the previous layer, we say that a layer is **fully connected** or **dense**, Figure 1.2b is an example of one. NNs with many hidden layers fall into the category of deep neural networks (DNN).

1.2.1 Activation functions

Activation functions introduce non-linearity to a chain of otherwise linear transformations, which enables ANNs to approximate any continuous function [1]. There are many different kinds of activation functions as seen on Figure 1.3, such as sigmoid function and rectified linear activation function (ReLu). A 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 [3]. It takes a real number and squeezes it into a range between 0 and 1. It was later shown that training NNs with sigmoid activation function often hinders the training process as saturated outputs cut off parts of networks, thus preventing the training algorithm from reaching all neurons and correctly configuring the weights [3]. It has since fallen out of practice and is nowadays replaced by ReLu or some other activation function.

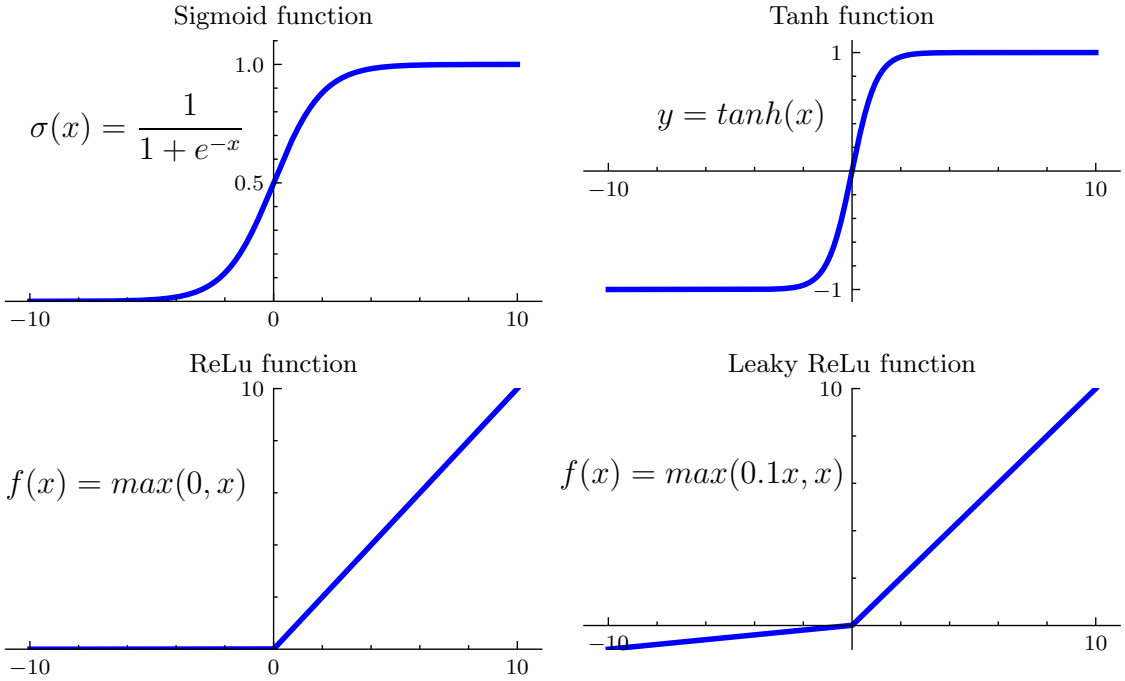


Figure 1.3: Different activation functions and their equations.

Another commonly used activation function is a softmax function (seen in 1.1, which is takes a vector as an input, computes an exponential of every element inside it and divides that with the sum of exponents of all elements [1] The end results is that softmax function transforms vector of values into a vector of probabilities. Softmax is usually used as an activation in a last layer of a classifier network.

$$\sigma(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}} \quad \text{for } i = 1, \cdots, K \text{ and } y = (y_1, \cdots, y_k) \in \mathbb{R}^K \tag{1.1}$$

Where:

y - Input vector

K - Number of elements in input vector

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

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, a neural network is shown a small batch of training data. Each instance is feed into NN and the final output label is calculated. This is known as **forward pass**, which is the same as making predictions, except that intermediate results from each neuron in every layer are stored. Calculated output is compared to an expected one using a **loss** (also known as **cost**) function. The loss function returns a single value, which tells us how badly is our NN performing, the higher it is, the 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 the 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 the backpropagation algorithm does this in steps. After computing the loss function algorithm analytically calculates how much each output connection contributed to

the loss function (essentially local gradient) with the help of previously stored intermediate values. This step is recursively done for each layer until the first input layer is reached. At that moment algorithm knows in which direction should each weight and bias change so that value of the loss function lowers. A procedure known as a **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 a loss function. Gradient Descent is not only used when training neural networks but also when training other ML algorithms.

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

1.2.3 Convolutional neural networks

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

In Figure 1.4 we can see an example of CNN which takes an image of a car as an input and outputs probability results in five different classes.

Specific to CNNs are two different types of layers, **convolutional** layers and **pooling** layers. Each convolutional layer detects some sort of shapes, 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 the spatial dimension, thus decreasing the number of parameters and operations needed in CNN. After a few alternating pairs of convolutional and pooling layers, the output of the last pooling

[1]Scientists Weisel and Hubel showed that different cells in the primary visual cortex of a cat responded differently to different visual stimuli [3]. Some were activated when shown a horizontal line in a specific location, some were activated by vertical lines. More complex cells responded to boxes, circles and so on. CNNs also detect simpler shapes first and use them to detect more complex ones later.

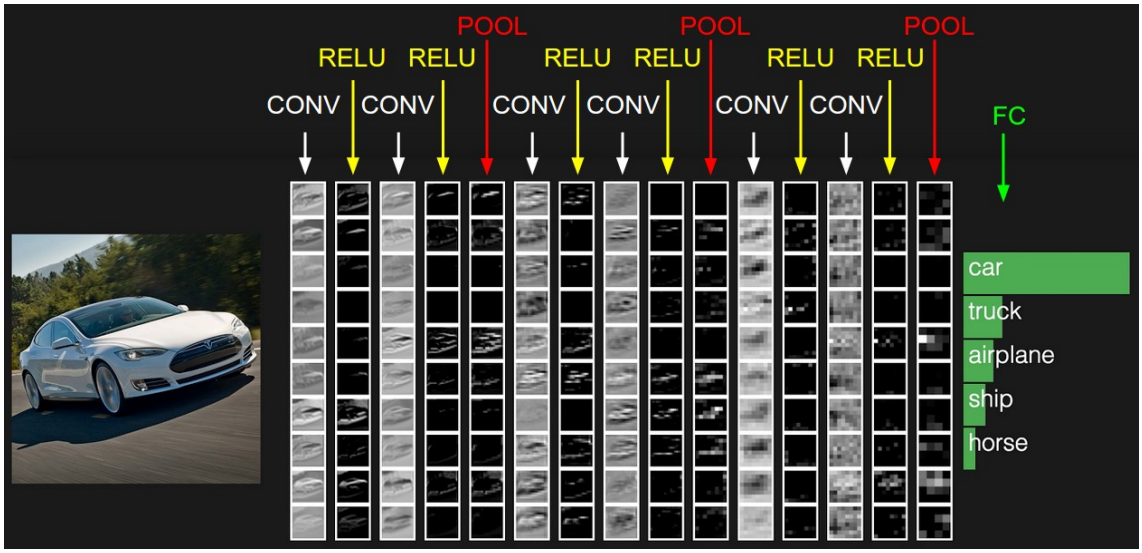


Figure 1.4: CNNs usually consist of alternating convolutional layers and pooling layers. Last polling layer is flattened out and feed into fully connected NN. Image source: [3]

layer is flattened out into one dimensional vector and feed into fully connected NN which produces probability results in given classes.

It makes sense to explain how convolutional and pooling layers work in greater detail as this will be important later when we will be designing our CNN models in section 2.5.

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 these matrices we will refer to them as volumes.

Convolution layers perform dot products between input volume and several **filters** or **kernels** to produce output volume. In these layers, filters are configured through the training phase. We can see a concrete example in Figure 1.5. 2D filter with size 2 x 2 covers a part of the input volume, over which element-wise multiplication is computed, elements are summed and the result is written into the first element of output volume.

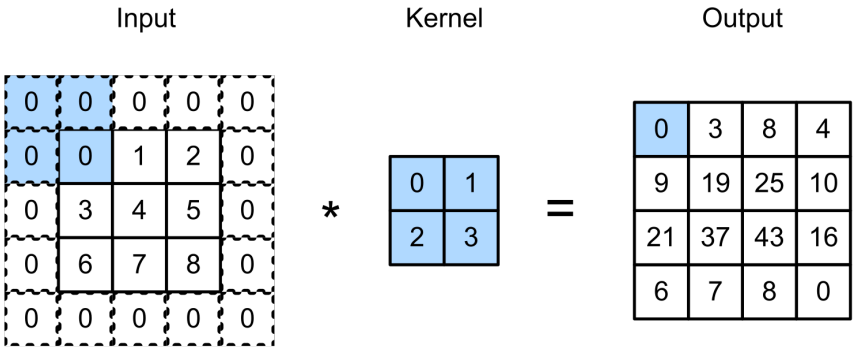


Figure 1.5: Dot product operation between filter and zero-padded input matrix. Image sources: [8]

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

The size of output volume depends on several factors as seen in 1.2.

$$V_o = (V_i - F + 2P)/S + 1 \tag{1.2}$$

Where:

V_i - Input volume size, only width or height

V_o - Output volume size, only width or height

F - Filter or receptive field size

P - Amount of zero padding used on the border

S - Stride length

If we examine the example in Figure 1.5 we can see that input with a size 3 x 3, stride 1, padding 1 and filter with a size 2 x 2 produces output with size a 4 x 4.

The depth of output volume is equal to the number of filters used in the convolutional layer as seen in Figure 1.6, it is a norm that a single convolutional layer uses a large number of filters to produce a deep output volume [3]. It is also common to set padding, stride and filter size so that the width and height of input volume are preserved. This prevents the information at the edges to be lost too quickly [3].

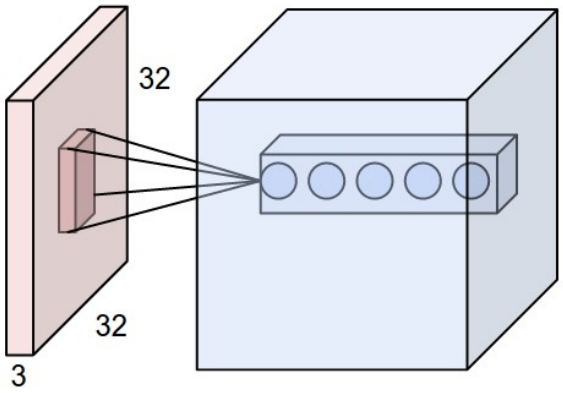


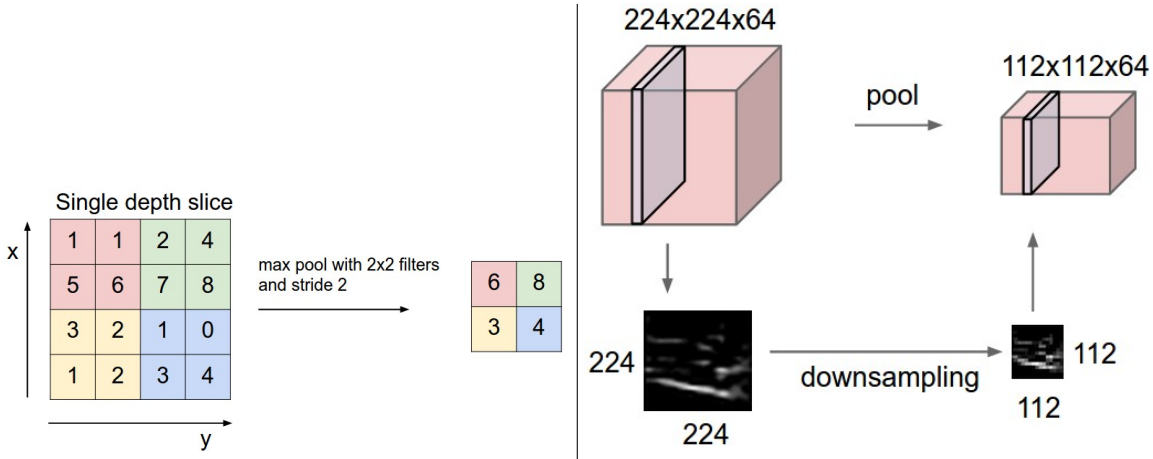
Figure 1.6: Convolutional layer with five different filters. Image sources: [8] [3]

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

1.2.3.2 Pooling layers

Polling layers perform the 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 1.7). Pooling is done on each depth slice separately of other slices, so depth size is preserved through the layer.

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



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

Figure 1.7: Polling layer. Image source: [3]

1.3 TensorFlow

TensorFlow is a free and open-source framework for numerical computation. It is particularly suited for large-scale machine learning applications [1]. It started as a proprietary project developed by a Google Brain team at Google in 2011 and became open-source in late 2015. It is used in many of Google's products such as Gmail, Google Cloud Speech and Google Search.

TensorFlow gives programmers tools for creating and training ML models, without needlessly diving into 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 an ML project is usually data preparation. After that is done, the creation of an ML model, its training and evaluation can be done in a few lines of Python code.

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

operations as well.

And finally, TensorFlow's trained output model is portable [1]. Models can be trained in one environment and executed in another. This means that we can train our model by writing Python code on a Linux machine and execute it with Java on a 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 [9]. It provides support for Android and iOS devices, and embedded Linux. TensorFlow Lite for Microcontrollers (TFLite Micro) is a recent port of TFLite (as of mid 2019), dedicated to running ML models on microcontrollers. TFLite itself provides API in different languages, such as Java, Swift, Python and C++. TFLite Micro uses C++ API, specifically C++11, which reuses a large part of the general TensorFlow codebase.

TFLite Micro library does not require any specific hardware peripherals, which means that the same C++ code can be compiled to run on a microcontroller or a personal computer with minimal changes. Users are only expected to implement their version of `printf()` function. As microcontroller binaries are usually quite big, flashing firmware to a microcontroller is a time consuming procedure. It makes sense to first test and debug the program that includes only ML inference specific code on a personal computer, before moving on to a microcontroller 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 [9]. To use the library for embedded development the whole project has to be cloned from the GitHub. The TensorFlow team provides users with several example projects that have been ported to several different platforms such as Mbed, Arduino, OpenMV and ESP32. Example projects show how to use TFLite API while showcasing different ML applications: motion detection, wake word detection and person detection.

Is important to know that TFLite is just an extension of the existing TensorFlow project. General steps for creating a trained ML model are still the same as seen in Figure 1.1, although we have to be aware of some details. Figure 1.8 shows all steps that are needed to prepare an ML model for running on a microcontroller.

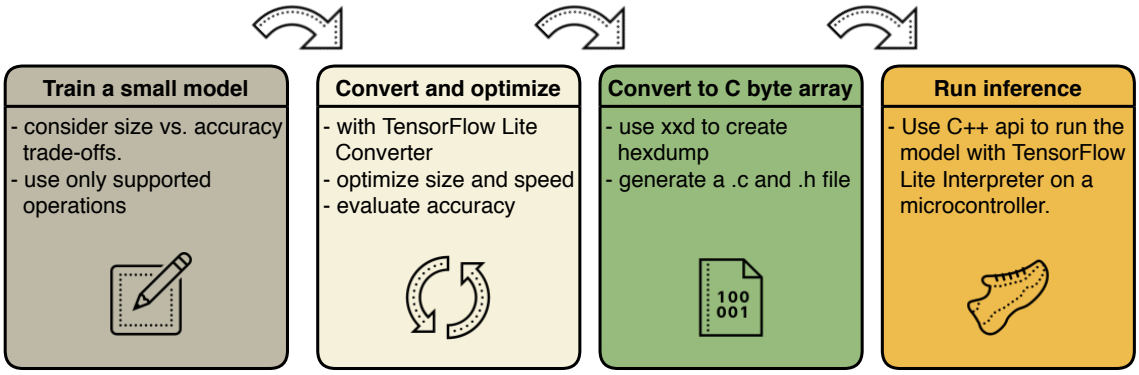


Figure 1.8: Workflow of preparing a ML model for an inference on a microcontroller. Icons source: www.icons8.com

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

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

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

1.3.1.1 Post-training quantization

By using quantization optimization we approximate floating-point numbers in a different format, usually with 8-bit integers. When computing neural networks we can quantize weights, biases and intermediate values output by separate neurons. Quantization has a dramatic effect on the size of the model and its execution speed. By changing 32-bit floating-point numbers with 8-bit integers size decreases by a factor of 4. Floating-point math is by nature slow to compute, many microcontrollers do not even have a floating-point unit. In comparison integer math is faster to compute, therefore quantized models are executed faster. Model accuracy decreases after using quantization, but usually 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 [10]. A device or a thing is a battery powered embedded system such as smart watch, heart monitor, or animal tracker which would transmit collected sensor data to an IoT gateway, which would relay the data over to the cloud. 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 the energy consumption of machines in factories, monitoring conditions of crops in agriculture, or monitoring locations of endangered species in African conservation parks.

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

Because our early detection system demands a decent battery life of several months and needs to send a small amount of data over one or two 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 [11]. NB-IoT was made specifically with embedded devices in mind, it has a range of to 15 km and it has deep indoor penetration [11]. Compared to Sigfox and LoRa it has better latency and higher data rate, but also higher power consumption [12]. However it is unsuitable for our use case as it operates on the network provided by the cellular base towers, which is inconvenient as the mobile connection in Assam, India can be inconsistent [13].

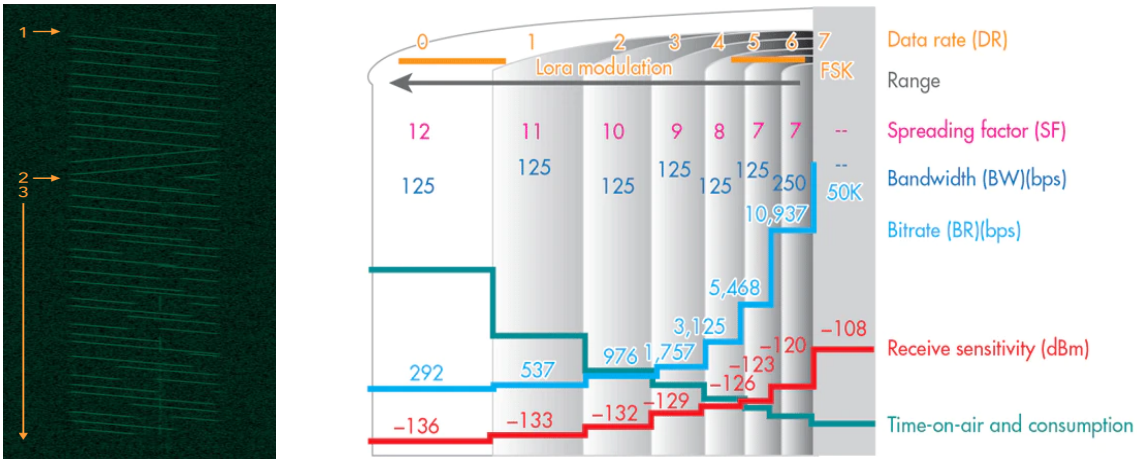
Sigfox is a radio technology developed by the company of the same name that operates on an unlicensed industrial, scientific and medical (ISM) radio band. In many views it is similar to LoRa, as it has the comparative range and power consumption [12]. However, there are a few important differences. Although Sigfox modules are a bit cheaper when compared to Lora modules, each message is paid, devices are limited to 12 bytes per uplink, 140 uplinks per day and only 4 downlinks are available per day. Sigfox devices can also only communicate with base stations, installed by the Sigfox company [12]. 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 set up 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 [14] [11]. The protocol is proprietary and owned by a semiconductor company Semtech, who is the sole designer and manufacturer of Lora radio chips in the world. LoRa protocol uses a modulation similar to chirp spread spectrum modulation [14]. As the protocol is proprietary exact details of it are not known, however it was reverse engineered by radio frequency specialist [15]. An example of a LoRa signal that was captured with a software defined radio can be seen in Figure 1.9a. Each symbol is modulated into a radio signal whose frequency is either increasing or decreasing with a constant rate inside of a specified bandwidth.

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



(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: [15] [16]

This kind of modulation gives LoRa extreme resiliency against the interference of other radio frequency signals that might be using the same frequency band [14] [16]. For example, on a lower part of Figure 1.9a we can see a signal with constant frequency transmitting inside the bandwidth window that the LoRa signal is using. This kind of interference is 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 the range, power consumption and bit rate of a LoRa signal. For example, as seen in Figure 1.9b, by increasing the 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 [14]. Its specification is open, so anyone can implement it. LoRaWAN takes care of communication between

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

LoRaWAN has a star of stars topology [14]. Devices deployed in the field transmit messages on frequency bands that differ from region to region. Messages are received by gateways which relay them to the network server. The network server displays relayed messages, decodes them and sends them to various applications. If the same message is heard by several gateways, the server drops all duplicates. The server also decides which gateway will send a downlink message to a specific device.

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

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 1.10. 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) [17]. We can broadly classify IR cameras into two categories: photon detectors or thermal detectors [17]. Photon detectors convert absorbed EM radiation directly into electric signals by the change of concentration of free charge carriers [17]. Thermal detectors covert absorbed EM radiation into thermal energy, raising the detector temperature [17]. The change of the detector's temperature is then converted into an electrical signal. Since photon detectors are expensive, large and therefore unsuitable for our use case, we will not describe them in greater detail.

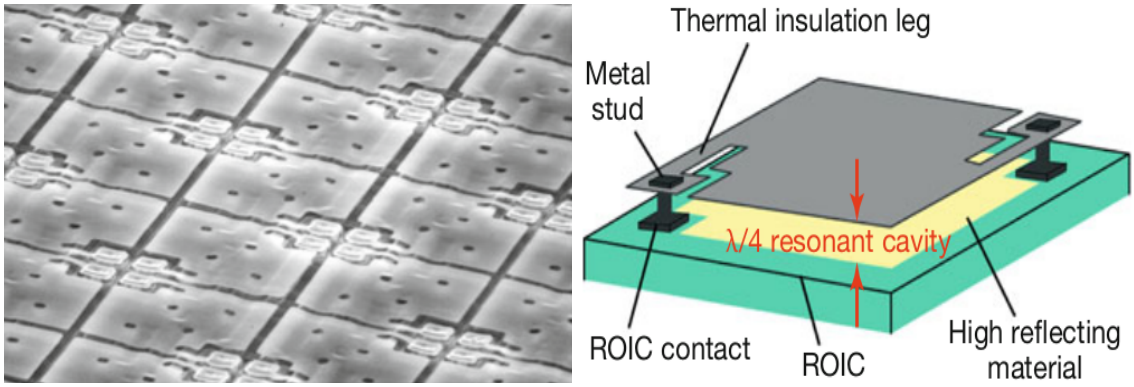
Common examples of thermal detectors are thermopiles and microbolometers. Thermopiles are composed of several thermocouples. Thermocouples consists of two dif-

Figure 1.10: Comparison between an image taken with a normal camera (left) and with a thermal camera FLIR Lepton 2.5 (right). Image source: Arribada Initiative [18]

ferent metals joined at one end, which is known as the hot junction. The other two ends of the metals are known as cold junctions. When there is a temperature difference between the hot and cold junctions, a voltage proportional to that difference is generated on the open ends of the metals. To increase voltage responsivity, several thermocouples are connected in series to form a thermopile [17]. Thermopiles have lower responsivity when compared to microbolometers, but they do not require temperature stabilization [17].

Microbolometers can be found in most IR cameras today [17]. They are sensitive to IR wavelengths of 8 to 14 µm, which is a part of the longwave infrared region (LWIR) [17]. Measuring part of a microbolometer is known as focal point array (FPA) (Figure 1.11a). FPA consists of IR thermal detectors, bolometers (Figure 1.11b), that convert IR radiation into an electric signal. Each bolometer consists of an absorber material connected to a readout integrated circuit (ROIC) over thermally insulated, but electrically conductive legs [19].

Absorber material is made either out of metals such as gold, platinum, titanium, or more commonly out of semiconductors such as vanadium-oxide (VOx) [19]. The important property of absorber materials is that electrical resistance changes proportionally with material's temperature [17]. When IR radiation hits absorber material, it is converted into thermal energy, which raises the absorber's temperature, thus

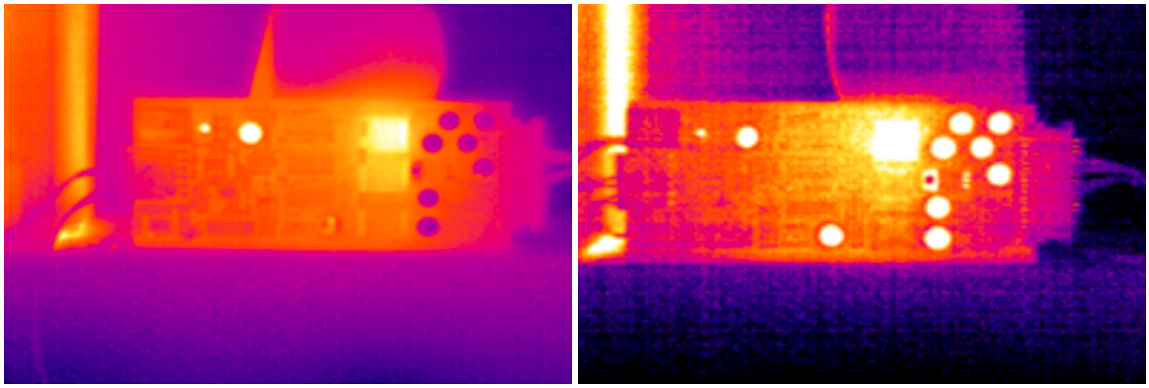


(a) Focal point array (b) Bolometer

Figure 1.11: (a) Focal point array under electronic microscope. (b) Bolometer with $\lambda/4$ resonant cavity. Image source: Vollmer, Möllmann [17]

changing its resistance. To detect the change in resistance, ROIC applies steady-state bias current to absorber material, while measuring voltage over conductive legs [17].

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 [17]. The thermal conductance of bolometers is minimized by enclosing FPA into the vacuum chamber, thus excluding thermal convection and conduction due to surrounding gasses. The only means of heat transfer that remain are radiant heat exchange (highly reflective material below the 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 the thermal image. Today's thermopiles can achieve NETD of 100 mK, microbolometers 45 mK, while photon detectors can have NETD of 10 mK. Although tens of mK does not seem a lot, we can see in Figure 1.12 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 [20]

1.5.1 Choosing the thermal camera

The choice of thermal camera was made by Arribada Initiative [18]. They tested several different thermopiles and microbolometers while searching for desired properties. The camera had to be relatively inexpensive and small enough so that it could be integrated into a relatively small housing. The main property that they searched for was that elephants could be easily recognized from thermal images. That meant that the camera needed to have decent resolution and low NETD. Cameras were tested in Whipsnade Zoo and the Yorkshire Wildlife Park where images of elephants and polar bears could be made.

They tested two thermopile cameras (Heimann 80x64, MELEXIS MLX90640) and two microbolometer cameras (ULIS Micro80 Gen2, FLIR Lepton 2.5). Although thermopile cameras were cheaper than microbolometer cameras, the quality of images they produced was inferior, as can be seen in Figure 1.13.

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 [18].

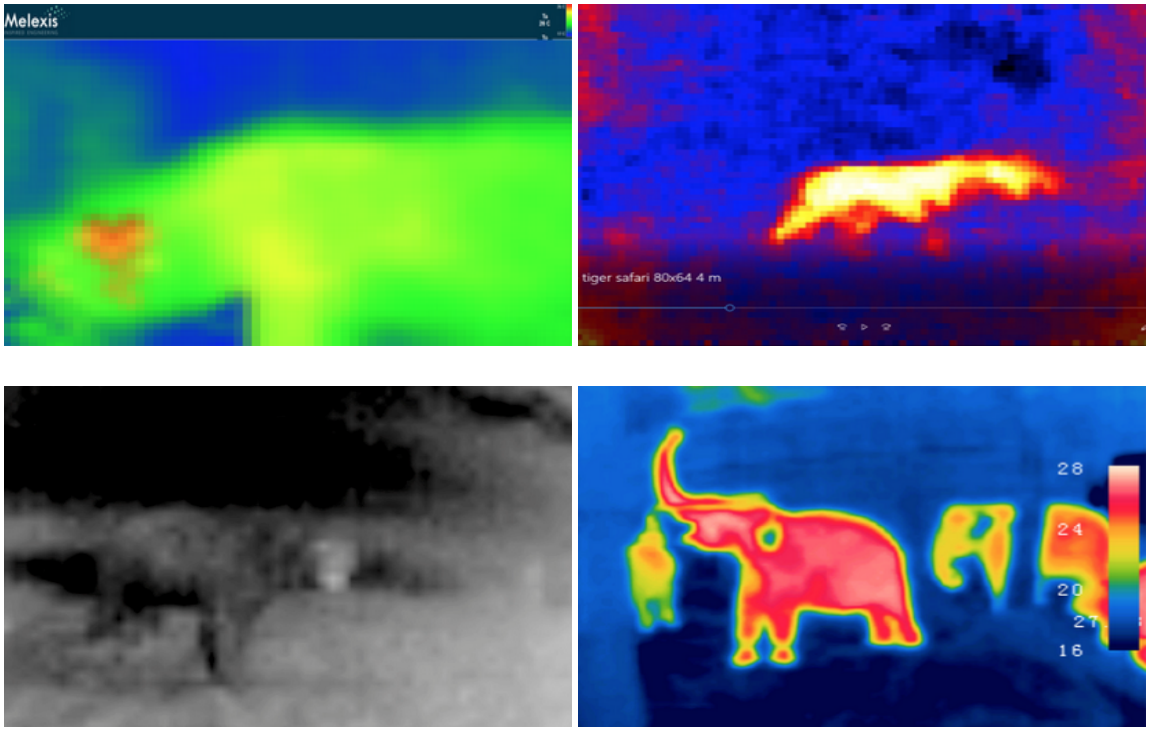


Figure 1.13: , MELEXIS MLX90640 (top left), Heimann 80x64 (top right), ULIS Micro80 Gen2 (bottom left) and FLIR Lepton 2.5 (bottom right). Image source: Arribada Initiative [18]

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

It is important to note that FLIR Lepton, like all microbolometers, requires frequent calibration to function properly. In temperature non-stabilized cameras small temperature drifts can have a major impact on image quality [17]. Calibration is done either by internal algorithms of the camera or by exposing the camera to a 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 afterward 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

the same and have exact specifications, they only differ in resolution, 3.5 has a 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 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 1.1 and 1.8.

We first had to set concrete objectives, while keeping in consideration various constraints. We then explored the dataset provided by the Arribada Initiative, analyzed different class representations and decided, if the dataset is appropriate for accomplishing objectives that we set earlier. The tools and development environment that we used in the process are also described.

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.

2.1 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.

2.2 Exploring the dataset

As mentioned in section **??** thermal image dataset was provided by Arribada Initiative [21] [22]. 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 2.1.

Figure 2.1: Camera trap used in Assam, India. Image source: Arribada Initiative [22]

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. Datasets like this are used to train a much more robust model, which performs better on never before seen image data when deployed in real life.

This was accomplished in ZSL Whipsnade Zoo, where they could take many images of elephants in a variety of different conditions [23]. 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 2.2.

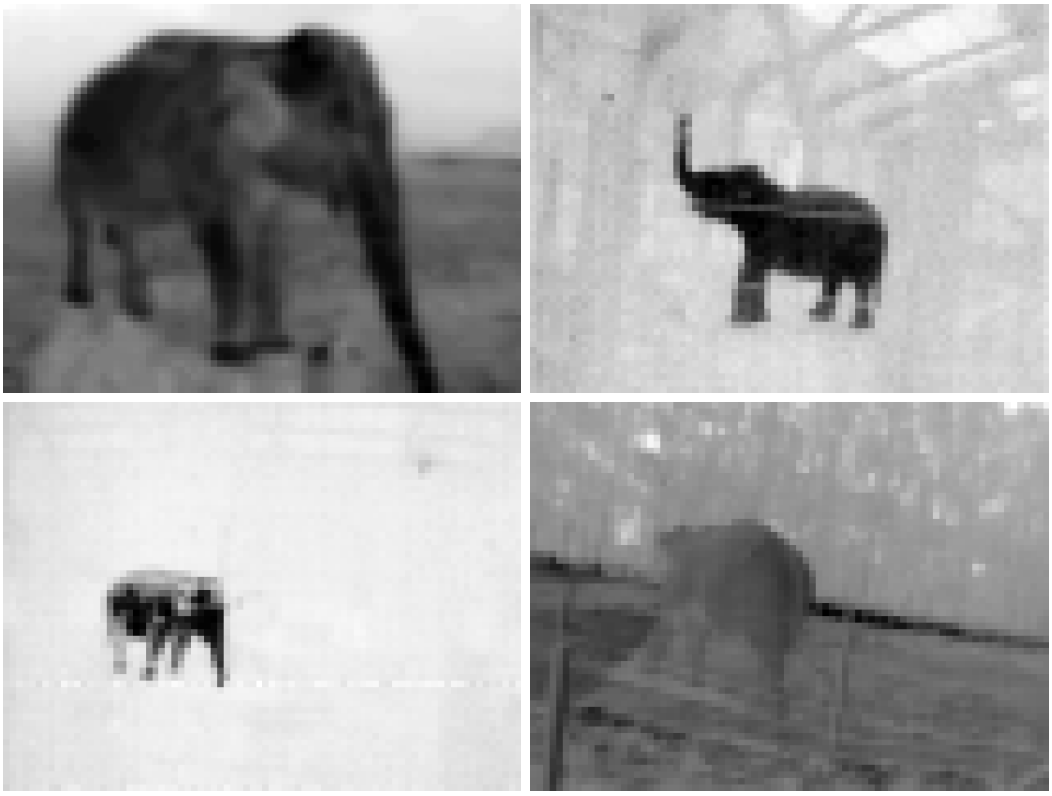


Figure 2.2: Thermal images of elephants from dataset.

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.

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 2.3.

   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

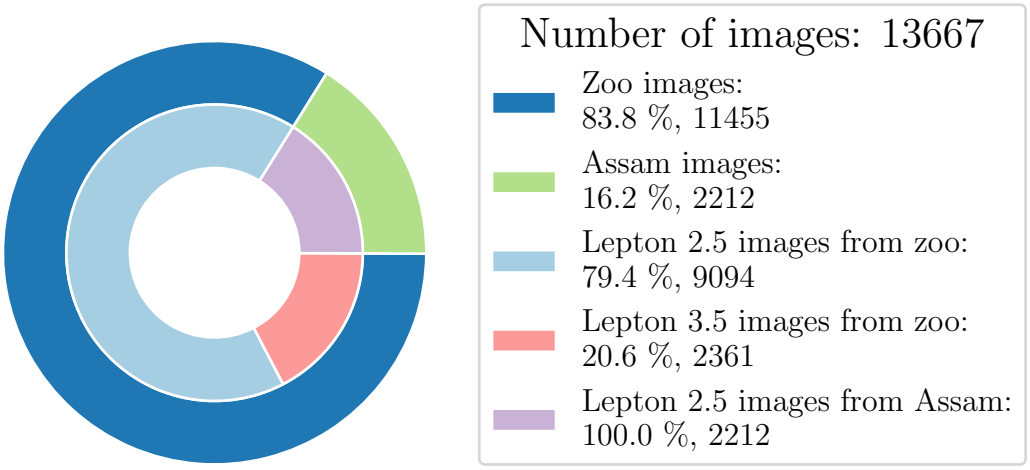


Figure 2.3: Distribution of used images from thermal dataset depending on image location and type of sensor.

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 2.4 in favor of elephant class.

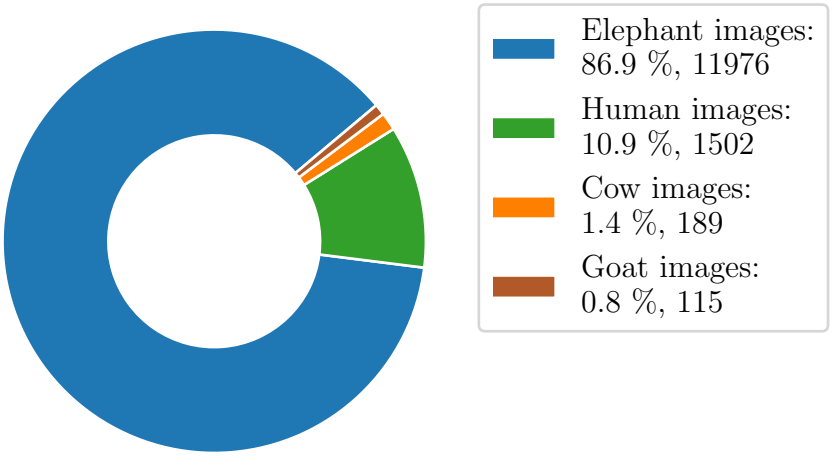


Figure 2.4: Class distribution of thermal images.

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.

2.2.1 Gathering thermal images

As 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 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 [24] 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 2.5.

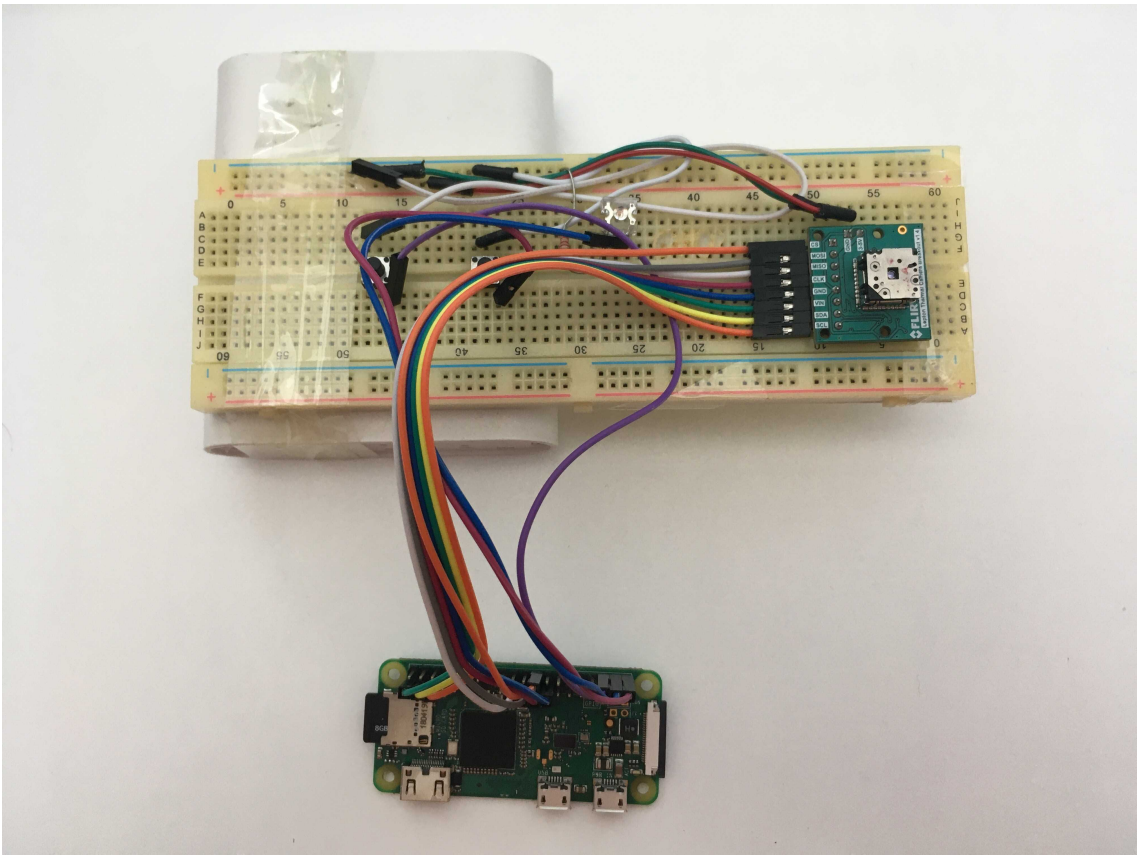


Figure 2.5: Camera setup used for taking thermal images with FLIR Lepton 2.5.

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

2.4 Image preprocessing

The image preprocessing phase is a pipeline process that differs from project to project. Our process can be seen on Figure 2.6.

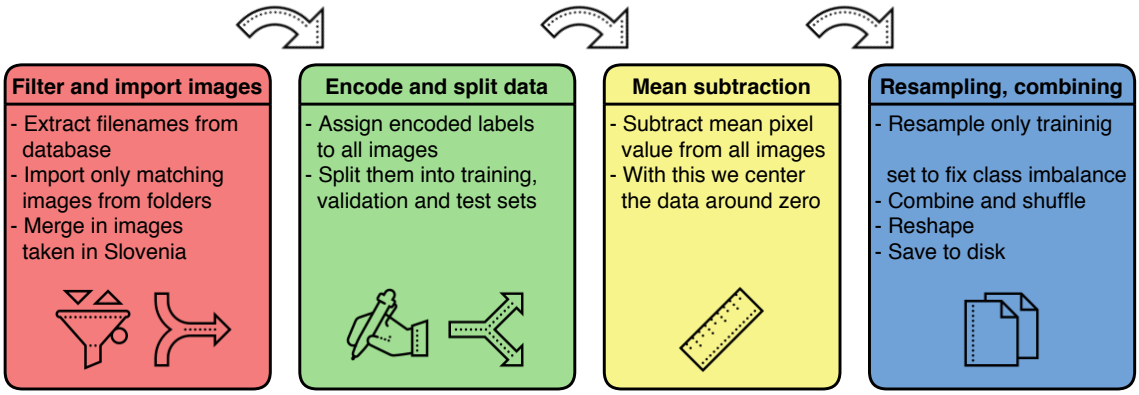


Figure 2.6: Image preprocessing pipeline. Icons source: [25]

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 2.7 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.

2.5 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

[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 [3]. Because images that were created with FLIR camera were all 8-bit encoded, therefore had the same range, this was not needed.

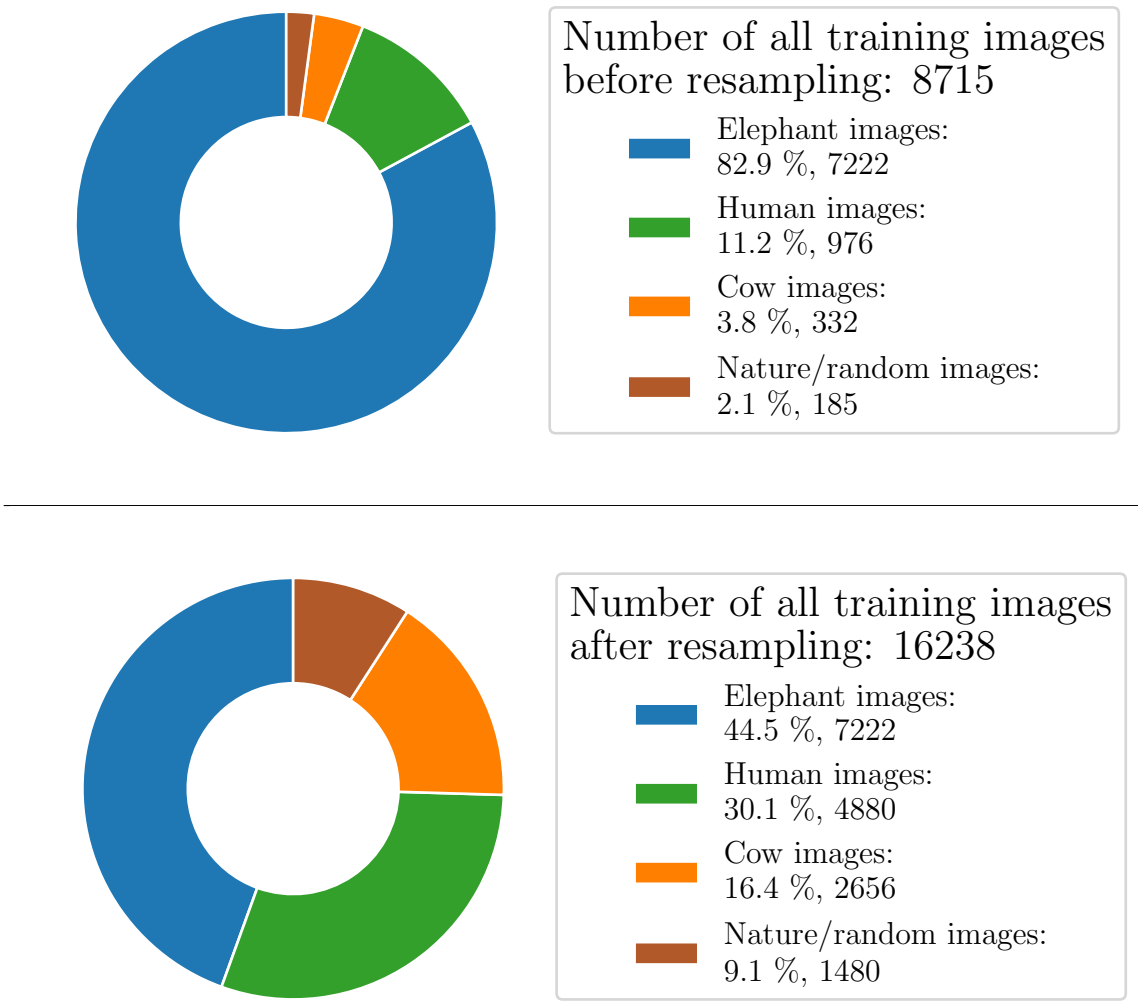


Figure 2.7: Distribution of training images before and after resampling.

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 1.4. The best way to present the model is by inspecting the Sequential API code that creates it, code is shown in Figure 2.8.

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 [3] [1]. 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 [1]. 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 [3]. 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 parameters 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

[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 [1].

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 2.8: CNN architecture written in Python using Keras Sequential API.

2.6 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.

2.7 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 2.9 we can see the so-called Impulse Design tab, where we design by selecting different blocks. With input block, we tell what kind of data are we inputting, either image or time-series data, and with processing block we decide how are we going to extract features. With learning block, we can choose to use a neural network provided by Keras, an anomaly detection algorithm, or a pre-trained model for transfer learning.

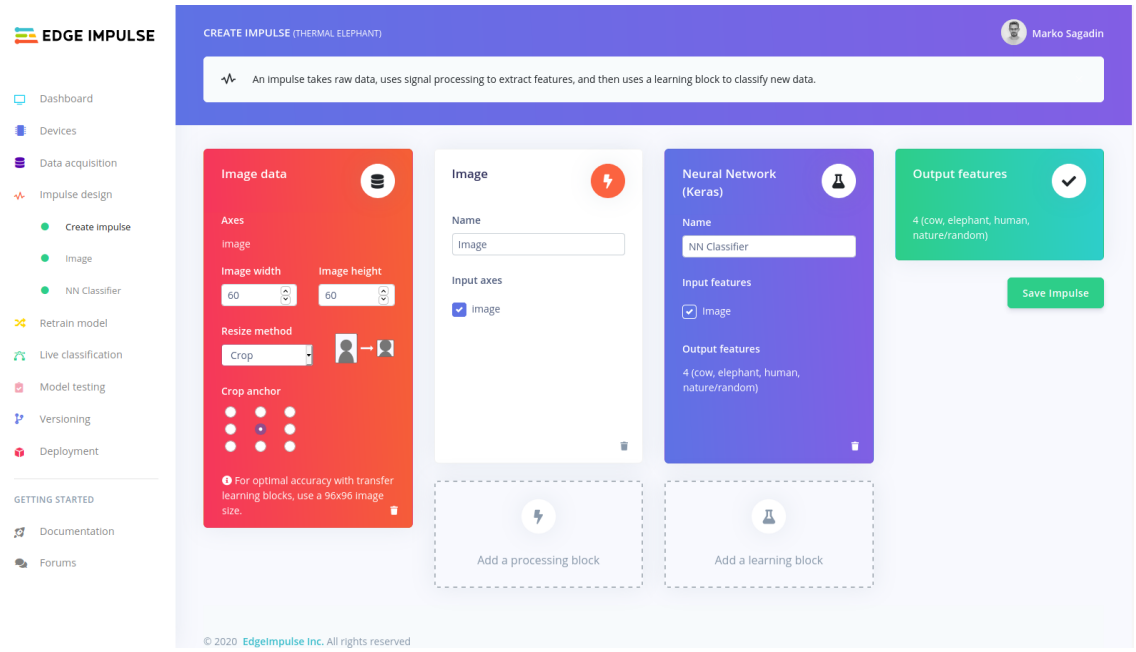


Figure 2.9: Creating a neural network in Edge Impulse Studio.

Since we were training with image data, we selected an image input block. As Edge Impulse did not support images of different image ratios at the time, we had to crop our images to 60 x 60 pixels. For processing block, we selected the image processing

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

block as this was the only possible choice and for learning block, we selected Keras's neural network block.

Neural network block is configurable, we could either define our network with different blocks representing layers or switch to the text editor with Keras Sequential API code, where we could do our adjustments. Settings such as learning rate, number of epochs, and confidence rating are also available regardless of the option we chose. 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 2.10, inferencing time is estimated for a Cortex-M4 microcontroller, running at 80 MHz. Edge Impulse also does conversion to an optimized full-integer model automatically.

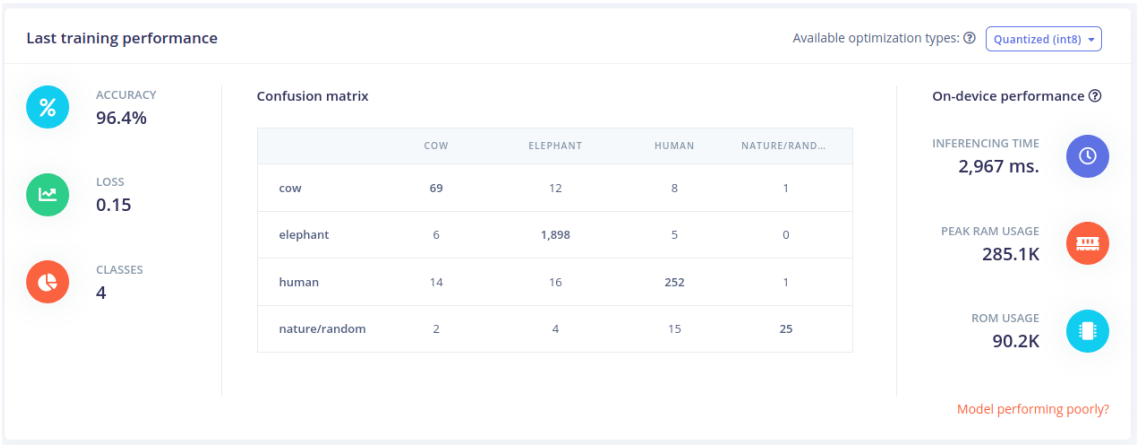


Figure 2.10: After training training performance report.

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

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

Bibliography

[1] Geron, A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems, 2nd edition.* O'Reilly Media, Sebastopol, CA, 2019.

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

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

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

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

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

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

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

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

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

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

[12] Polymorph, IoT connectivity comparison (GSM vs LoRa vs Sigfox vs NB-Iot). Available on: https://www.polymorph.co.za/iot-connectivity-comparison-gsm-vs-lora-vs-sigfox-vs-nb-iot/, [28.9.2020].

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

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

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

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

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

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

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

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

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

[22] 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].

[23] 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].

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

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