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 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 [2]. 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 from an accelerometer data. To train its learning algorithm, also known as a **model**, we need to expose it to 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 frequency or 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 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** [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 previous two and reinforcement learning, where model acts inside 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 field of deep learning, which is a sub-field of machine learning.

Training of deep learning algorithms 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 returned. 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 have to occur in order to get from an idea to a working ML based system as seen on Figure 1.1.

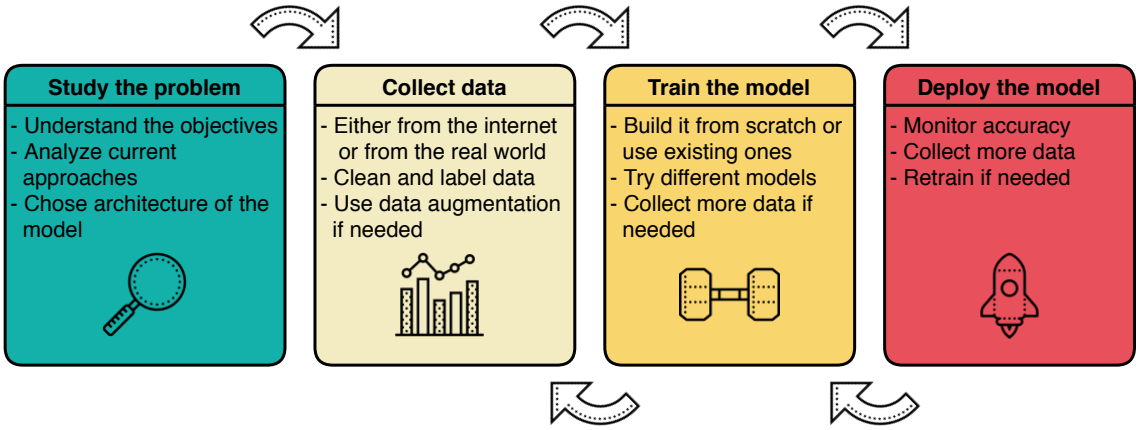


Figure 1.1: Workflow of solving a generic machine learning problem. Icons source: www.icons8.com

First problem has to be studied, it has to be understood what are objectives and decided which approach will be used. Here we decide on rough architecture of the ML model that we will use. 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. 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 can also use it to collect more data and retrain the model.

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 smart devices comes with many benefits. One of them is reduced power consumption. 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 time between event and action. 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, we would have to greatly simplify it.

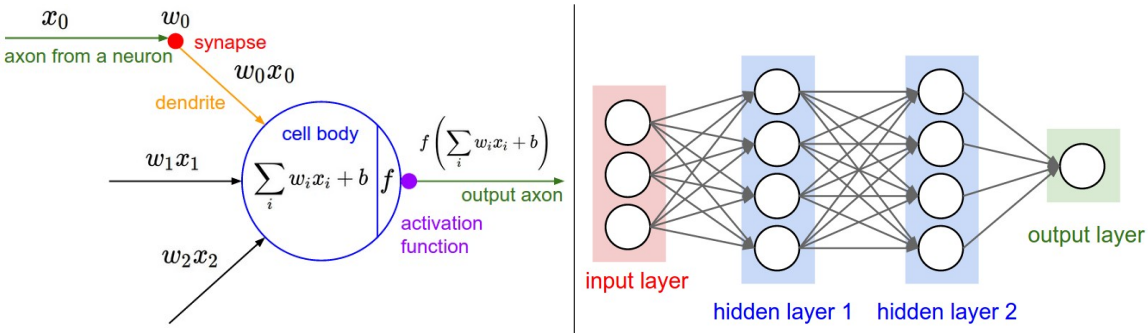
Usual workflow, while developing machine learning models for microcontrollers, is to train a model on training data on some powerful computer or server. When we are satisfied with the accuracy of the model we then quantize it (more about quantization in chapter TODO: ADD CHAPTER NUMBER, SHOULD THIS BE

A FOOTNOTE?) and convert it into a format understandable to our microcontroller.

1.2 Neural networks

Although 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**. 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 [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). 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 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 [3]. It has since fallen out of practice and is nowadays replaced by ReLu or some other activation function.

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

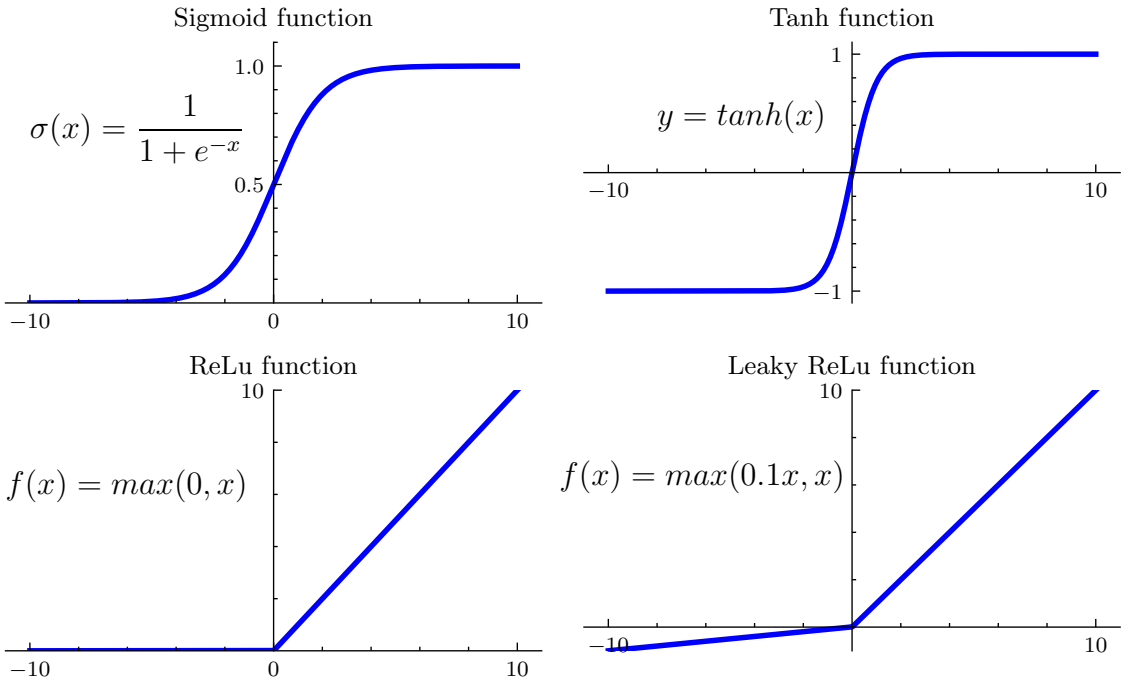


Figure 1.3: Different activation functions and their equations.

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]. It was shown that different cells in visual cortex responded differently to different visual stimuli [3]. 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.

On Figure 1.4 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.

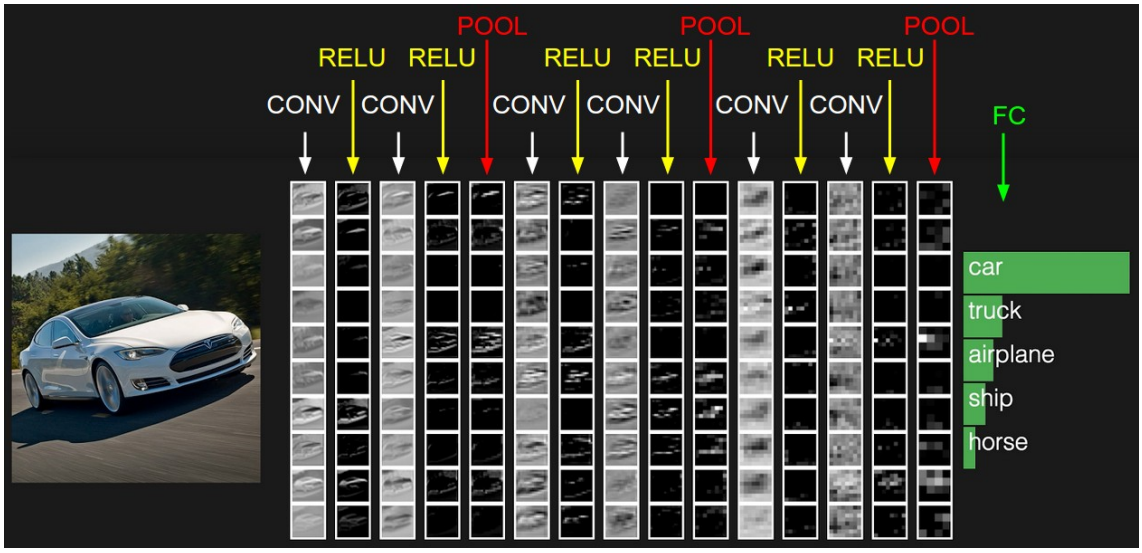


Figure 1.4: CNNs usually consist of alternating convolutional layers and pooling layers. Last polling layer is flattened out and feed into fully connected NN. Image source: [3]

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 probability results in given classes.

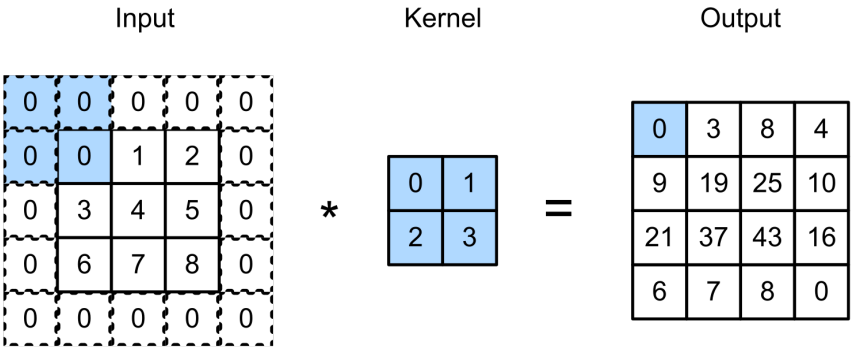
It makes sense to explain how convolutional and pooling layers work in greater detail as this will be important later when we will be designing our CNN models in chapter TODO.

1.2.3.1 Convolutional layers

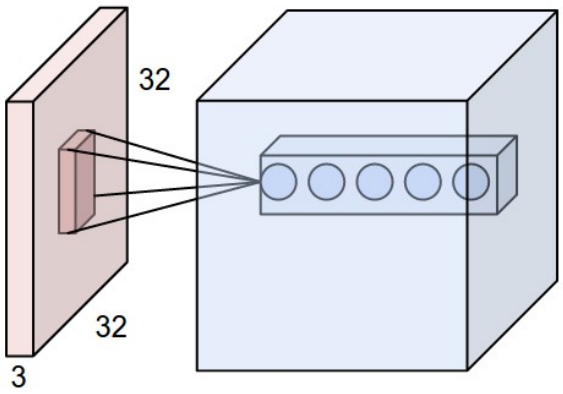
Data that CNNs operate on are 3 dimensional matrices, where width and height correspond to image resolution and depth corresponds to the number of color channels, 3 for colorful images (red, green, blue) and 1 for greyscale. When speaking about this data we will relate to them as volumes.

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

Filter then moves a fixed width 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 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.



(a) Example of dot product operation



(b) Convolutional layer

Figure 1.5: (a) Filter moves over zero-padded input matrix. (b) To produce output layer with depth 5, this layer would have 5 different filters. Image sources: [8] [3]

The size of output volume depends on several factors as seen in 1.1.

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

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 1.5a we can see that input with a size 3 x 3, stride

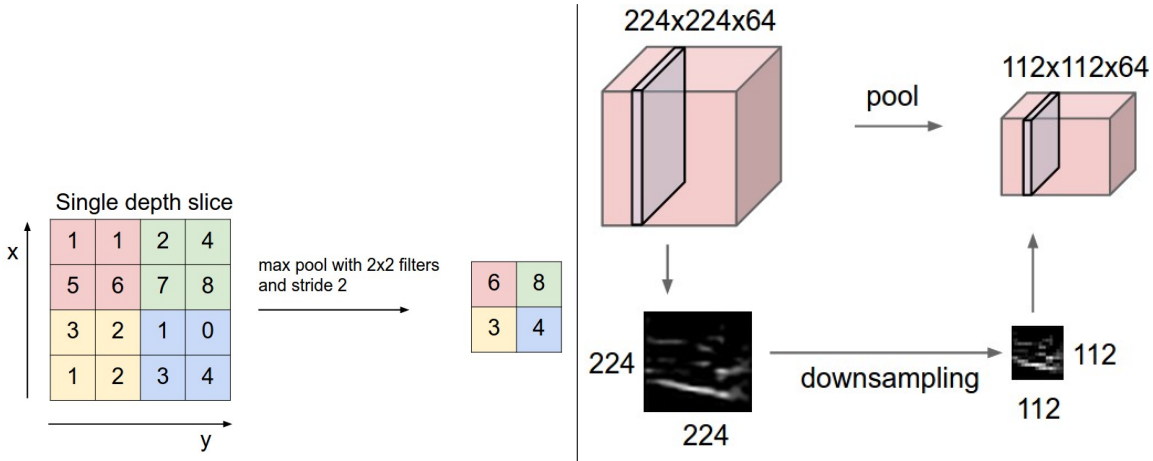
1, padding 1 and filter with a size 2 x 2 produces output with size a 4 x 4.

Depth of output volume is equal to the number of filters used in convolutional layer, it is a norm that a single convolutional layer uses large number of filters to produce a deep output volume [3]. 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 [3].

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 1.6). 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 1.6: Polling layer. Image source: [3]

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 [1]. TensorFlow 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 [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 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 [9]. It provides support for Android and iOS de-

vices, 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 [9]. 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.

Part of this thesis was concerned with porting TFLite Micro to **libopencm3**, our platform of choice. To compile source files and build binaries TFLite Micro uses build automation tool **GNU Make**. One large makefile that includes several platform specific makefiles dictates how firmware is built. By providing command line arguments users decide which example has to be compiled and for which platform. The build system makes some assumptions about locations of the platform specific files, which in case of example projects are scattered over GitHub repository. This and several other reasons make library hard to use when porting it to a new platform. In section TODO ADD REFERENCE we describe our build system that keeps library and application specific code separated in a clear and concise manner.

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 1.7 shows all steps that are needed to prepare a ML model for running on a microcontroller.

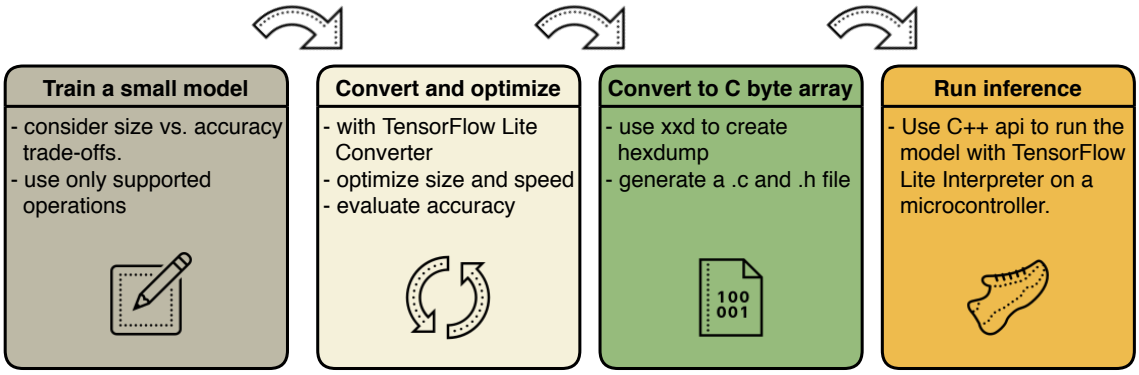


Figure 1.7: Workflow of preparing a ML model for an inference on a microcontroller. Icons source: www.icons8.com

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.

TensorFlow Lite Converter offers 3 different quantization methods: float16, dynamic range and full integer quantization [10]. Float16 quantization quantizes weights 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 range quantization stores weights 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 optimization also quantinizes math operations, thus even more increasing execution speed. This is an ideal choice for running models on microcontollers, however it should be noted that not all operations have full integer math implementations in TFLite Micro.

Model accuracy decreases after using any of quantization methods decribed above, but it is usually only a percent or two.

1.4 Thermal cameras

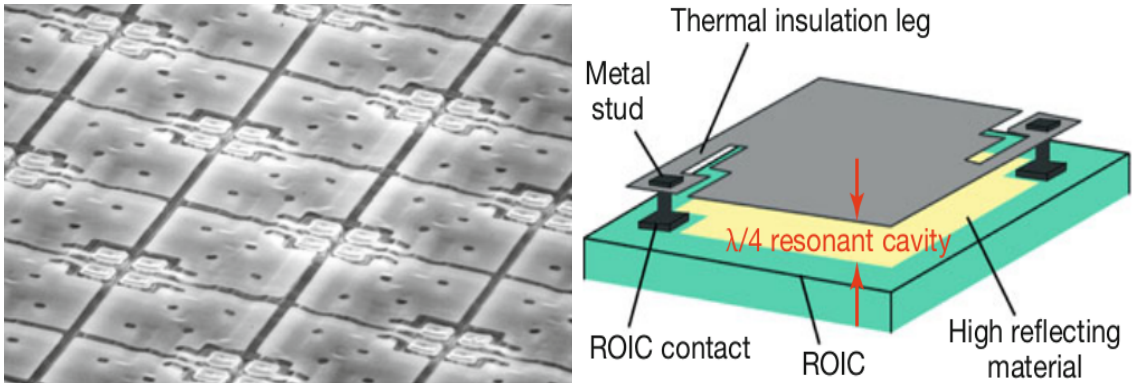
Thermal cameras are transducers that convert infrared (IR) radiation into electrical signals, which can be used to form a thermal image. A comparison between a normal and a thermal image can be seen on figure 1.8. IR is an electromagnetic (EM) radiation and covers part of EM spectrum that is invisible 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) [11]. We can broadly classify IR cameras into two categories: photon detectors or thermal detectors [11]. Photon detectors convert absorbed EM radiation directly into electric signals by the change of concentration of free charge carriers [11]. Thermal detectors covert absorbed EM radiation into thermal energy, raising the detector temperature [11]. 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 1.8: 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 [12]

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 [11]. Thermopiles have lower responsivity when compared to microbolometers, but they do not require temperature stabilization [11].

Microbolometers can be found in most IR cameras today [11]. They are sensitive to IR wavelengths of 8 to 14 µm, which is a part of longwave infrared region (LWIR) [11]. Measuring part of an microbolometer is known as focal point array (FPA) (Figure 1.9a). FPA consists of IR thermal detectors, bolometers (Figure 1.9b), 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 [13].



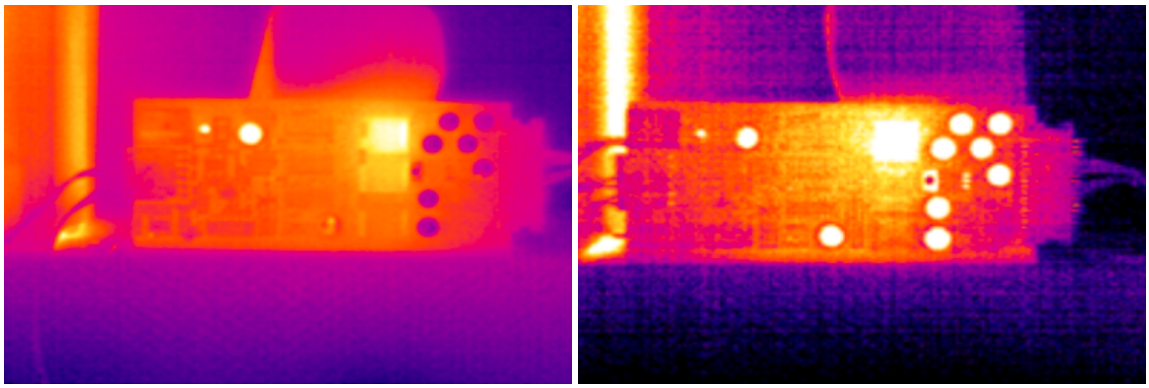
(a) Focal point array (b) Bolometer

Figure 1.9: (a) Focal point array under electronic microscope. (b) Bolometer with $\lambda/4$ resonant cavity. Image source: Vollmer, Möllmann [11]

Absorber material is made either out of metals such as gold, platinum, titanium or more commonly out of semiconductors such as vanadium-oxide (VOx) [13]. Important property of absorber materials is that electrical resistance changes proportionally with material's temperature [11]. 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 [11].

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 [11]. 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 1.10 what a difference of 20 mK means for image resolution and noise.



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

Figure 1.10: 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 [14]

1.4.1 Choosing the thermal camera

Choice of thermal camera was made by Arribada Initiative [12]. 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 1.11.

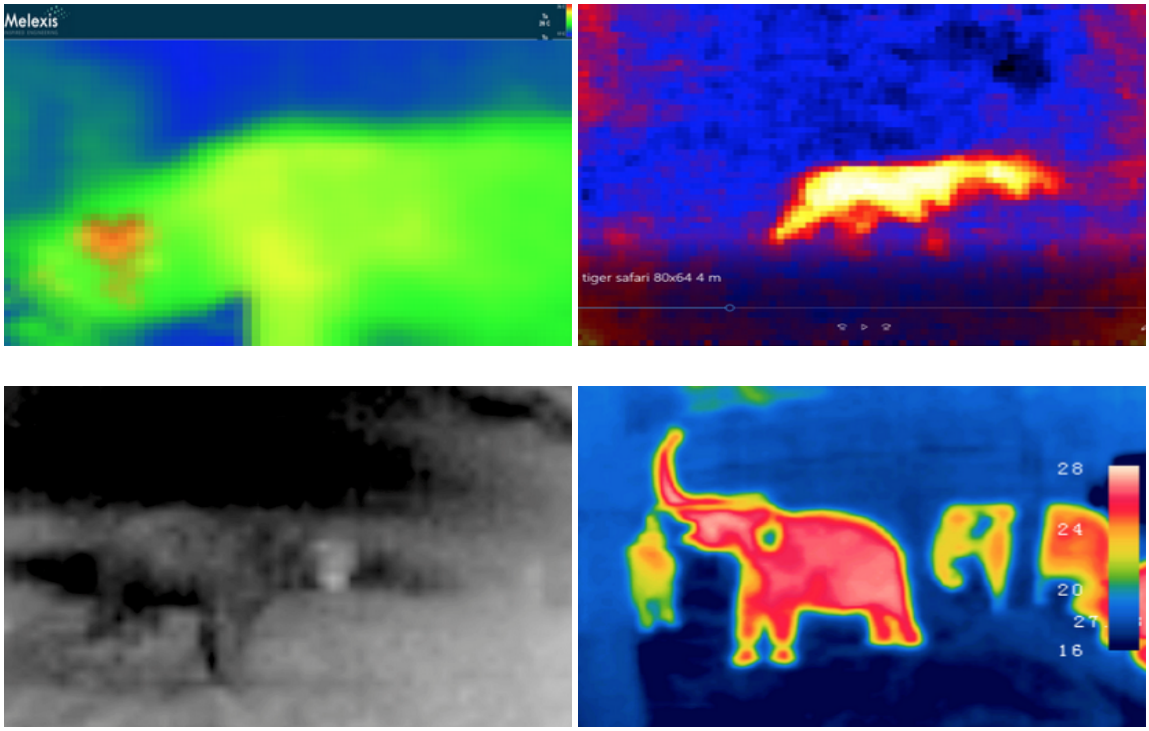


Figure 1.11: Comparison of image quality made by different thermal cameras, MELEXIS MLX90640 (top left), Heimann 80x64 (top right), ULIS Micro80 Gen2 (bottom left) and FLIR Lepton 2.5 (bottom right). Image source: Arribada Initiative [12]

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 [12].

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 [11]. 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.

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] TensorFlow, Post-training quantization. Available on: https://www.tensorflow.org/lite/performance/post_training_quantization, [21.9.2020].

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

[12] 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].

[13] 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.

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