TinyFace: Extreme Edge Face Detection on Embedded Devices

Teodor Fratiloiu

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

Department of Electrical and Computer Engineering
Technical University of Munich

Supervised by	Alexander Hoffman M.E., M.Sc.
	Chair of Real-Time Computer Systems
Submitted	September 23, 2020

With my signature below, I declare that this thesis is my own work, based on my study, and that I have acknowledged and documented all material and sources used in my research and during the writing of this work.

September 23, 2020	
Date	Signature

Abstract

Machine learning powered facial recognition is a capable and versatile tool, orchestrating cutting-edge applications in many areas of our daily lives. The backbone of many of these methods are deep, convolutional neural networks, which enable us to develop increasingly complex systems for face detection, recognition and processing. Deep CNN's traditionally require copious amounts of computational power and storage, yet the world of embedded devices often abounds in just the opposite, having many performance and storage constraints instead. We are henceforth required to find innovative solutions and workarounds for these limitations, while keeping quality and speed compromises at a minimum, if we want to run these advanced models on small, low-powered embedded systems. This work presents a machine-learning based application of human face detection, built with a TensorFlow-Lite backend and achieving a model size in the low hundreds of KB, while leveraging optimization methods such as quantization and pruning of weights, on a fully convolutional neural network model architecture. We further try to automate the workflow of building, training and optimizing new, deep, convolutional neural network models for embedded machine learning as much as possible, and, in the end, strive to present a working ansatz for solving the challenge of having an integrated, end-to-end ML-powered face detection product on tiny, micro-controller and edge devices.

Contents

Chapter 1

Introduction

In December 2019, Pete Warden and Daniel Situnayake of Google's AI research group published a book titled *"TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers"*. In this work, they take on the challenge of reducing the complexity and, very importantly, size, of TensorFlow Lite model architectures (originally designed for use on smartphones) to such an extent that they are realistically deployable to more constrained embedded devices and systems. The book defines the main working principles of running machine learning on embedded devices. [30] The term *TinyFace*, which was coined for the title of this thesis and, more broadly for the research project this thesis is a part of, is, therefore, a word-play on the title of the *TinyML* book, which was instrumental in the development of this thesis, and we want to acknowledge this from the very start.

Machine Learning (ML), a subclass of Artificial Intelligence (AI), is simply an *umbrella term* for many different systematized techniques one uses to teach a computer or make it learn by itself how to perform traditionally *human-only* activities. These could be tasks such as making *predictions* or *classifications* and, more generally, understanding a larger context, with the intent of earning insight and creating something new from a given input. In fewer words, Machine Learning means a computer trains itself to draw conclusions from existing data, and generate new data. That is to be done *without* deterministic algorithms, in the traditional computer programming sense - by this, we mean that *we do not program the AI ourselves, but that it trains itself* and that the results will vary, within a certain margin, for different runs, even with the same input. [9] But let us not understand from this that there won't be any algorithms at all; just the opposite. As a matter of fact, there will be, almost without exception, numerical programming, just not necessarily that much logical code (expressions, conditionals, loops, and such). As practical examples, let us take smartphone fingerprint sensors, or face detection sensor-arrays, which will often leverage ML pattern recognition to detect whether it truly is the actual owner of the phone who is trying to gain access, and decide whether to let him or her in. Or social media apps, which use AI detectors, for adding face filters to selfies, E-mail providers who use AI classifiers as spam filters, banks which often have regression-based AI protection against credit card fraud, or high-frequency trading firms, which often also rely on ML technology, such as random forests, for split-second decision making when buying, and selling, securities, or generative adversarial networks, which can generate new data, which, at least superficially, appears realistic. [10] A more detailed overview will be offered in Chapter 2.

The more specific sub-category of machine learning methods this thesis focuses on is deep, convolutional neural networks. These are incredibly complex systems processing vast mathematical structures, which traditionally require such high computing power to be useful for anything, that they could only really be deployed outside of scientific applications and laboratories, and into the real, consumer-world products in just the past two decades. With increasing hardware per-watt performance and dramatic price drops in manufacturing costs, machine learning has started moving *down from the cloud* and into end-user devices. First, that meant having ML on smaller servers in a back room, then on desk computers and laptops, and, in more recent years, on smartphones. The next step is bringing these techniques to ever smaller, embedded devices, such as micro-controller boards, which are the next great frontier in this field.

The motivation for bringing ML technology to tiny computers can be quickly formulated as feeding the ever-increasing need of the world for small, connected, electronic devices, to be deployed everywhere and be available around-the-clock, while taking on more and more tasks, fulfilling them reliably, accurately and repeatedly, and, on top of that, is also being expected to drop in price and grow in performance and speed. TinyFace, therefore, attempts to offer a contribution in the new field of small-scale face detection, namely detecting, with a camera, if a human being presented herself in front of the device. This could, for instance, be useful in a security application, where a smaller, embedded, camera device would detect whether a person has appeared before it, than wake up a larger computer to perform full facial recognition, in a similar fashion to how wake words work for digital assistants.

A typical machine learning development flow consists, most of the time, of compiling adequate datasets in terms of representation, size, and quality, building model architectures for the used convolutional neural networks and specifically defining their layer architecture in terms of size, type, and number of layers, then leveraging the dataset for training the neural network model. *Training* is the process of feeding data in a controlled and feedback-bound manner into a mathematical model with the scope of *teaching* it to perform a task that would require human intuition. After having trained a model to an acceptable level of performance, we can move on to using it in real-world applications. This step is called *inference*. ML on embedded systems is expected to only do inference on the MCU, with neural-network training and optimization being done off-device. Machine learning on embedded devices and specifically face-detection will be addressed in Chapters 3 and 4, respectively.

The technologies used for TinyFace are frameworks such as TensorFlow (Lite), which offer us the necessary tools to construct and train neural networks, as well as processes to make them fit on our embedded hardware, such as bit-level quantization, model compression and weight-pruning techniques. Furthermore, a data-collection and dataset-compilation workflow, training and evaluation routine, as well as a practical application, are presented. This will be the subject of Chapters 5 and 6.

Without any further ado, let us proceed to the next chapter, where several Machine Learning techniques, as well as some field-specific terminology, will be discussed, and where we will present a theoretical background for the subsequent chapters.

Chapter 2

Prerequisites

To accomplish our goal of detecting whether a face is present in one given image, we need to first explain the methods that will be used. Let us, therefore, look into Machine Learning (*ML*) from a theoretical perspective. We will address three main points in this chapter: the problems ML tries to solve, what a Convolutional Neural Network (*CNN*) consists of, and, lastly, how such a theoretical contraption is relevant for our application - we will discuss, amongst others, training, parameters, hyperparameters, and optimization. A thorough, textbook-like, explanation is not intended. Instead, the focus will be on offering a brief overview of the key concepts.

2.1 Machine Learning

As Tom Mitchell put it in his 1997 book, *Machine Learning is the study of computer algorithms that improve automatically through experience.* [21] Let us expand on this definition: we want to have an algorithm, that *learns* from *data* and translates that experience into a mathematical model, which is used to make predictions or decisions, without the system designer having explicitly programmed a solution to the problem beforehand. [15]

Let us first define some key terms:

- **Dataset**: this refers to a collection of data points, either labeled or not, belonging to different classes and categories of objects, which our ML model is to learn about. A sample is offered in *Figure 2.1*;
- **Training**: also known as *fitting*, this is the iterative process of *learning* from the given data and the different classes of training will be discussed below. After a network has finished training on data, its state becomes permanent - it becomes what is known as a *frozen* tensor;

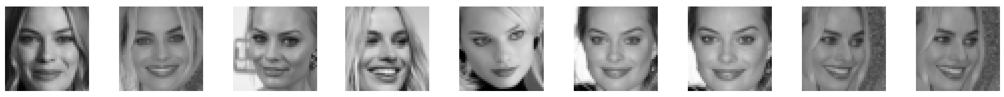


Figure 2.1: Sample images from a training dataset, showing Margot Robbie. The cropping around the face is very tight in this example. The amount of padding can and often does influence model performance.

- **Inference**: this is the part that comes *after* training, and refers to using the ML model on actual data, to draw conclusions and accomplish something in the real world. For instance, using a face detector to look for faces in an image is inference - an image is given as input, mathematical operations from the oriented graph of the neural network are applied to that image tensor, and an output is eventually computed from mathematically *combining* these two objects.

2.1.1 Classes of Learning and ML Algorithms

There are many different classes of Machine Learning. They can generally be broken up by type of training, or by desired outcome. Let us look at ML from the first perspective. There are, in essence, three types of learning: supervised, unsupervised, and reinforcement learning [16]. *Supervised learning* means that we train the network in a controlled, procedural manner, usually offering data and *feedback* at each training step. This translates to using labeled datasets - each data point has something similar to a *name tag*. Thus, the neural network learns where that data point belongs. *Unsupervised learning* refers to the algorithm analyzing data on an *as is* basis, observing features and drawing its own conclusions from it, without direct feedback from the trainer - this category of systems is following patterns and trends in the data, instead of relying on a third party trainer, of which there is none, in contrast to supervised learning methods. This usually implies that the computer will try to learn a complete statistical overview of the data to understand it. The main difference between the two can be formulated as follows: supervised learning means that we study different instances of *x*, each with a known *y*, and learn to predict unknown *y's* for new *x's*; unsupervised learning means we observe some interesting features and probabilistic properties of *x* and try to determine, as Goodfellow et al have it, its probability distribution *p(x)*. The final class is *Reinforcement Learning*, which, as the name suggests, keeps on learning in a sustained manner. This is a direct result of the fact that, in comparison to supervised and unsupervised learning - where training is completely separated from inference - in reinforcement learning, we don't have such a strict limit between the two. Instead, the algorithm keeps on learning while being used, and from new data. This also leads into the next big difference with reinforcement learning, namely that it is primarily used for dynamic systems. This can be useful, for instance, in places where there is a *feedback loop*, such as in control theory-related problems and automation tasks. We will not go into more detail on reinforcement learning techniques.

Let us now analyze ML algorithms from the second perspective - the desired outcome. Here, we can have procedures to make predictions (regression), to detect and analyze patterns, or divide the data according to commonalities (cluster analysis), to classify one given input into one of several classes (classification), and lastly, to derive and *distill* features from data thorough feature reduction. This thesis will focus on classification algorithms only. This class of ML can generally leverage all three kinds of learning - supervised (for instance, *Naive Bayes*), unsupervised (for instance, *Support Vector Machines - SVM* or *Random Forests*, which both also use supervised learning) and reinforcement (generally, only *Neural Networks* can use all 3 types of learning) [9]. An overview, with more examples, can be found in *Table 2.1*.

Task \ Class	Supervised	(Partially) Unsupervised	Reinforcement
Regression	Linear&Logistic	Conditional GAN	
Cluster Analysis		K-Means&Anomaly Detect.	
Classification	Naive Bayes	SVM/SVC [4]	Neural Networks*
Feature Reduction		PCA	

Table 2.1: Classes of Machine Learning and their related tasks. Neural Networks can use more than one learning technique during their training, or just any single one of the 3. [33] Conditional GAN's can be used to generate new data, such as new images, whereas linear and logistic regression are older, less sophisticated techniques, used to approximate functions. [19] Naive Bayes is a name for problem-solving techniques based on statistical analysis. SVM's can work as both supervised and unsupervised learners - in which case the method will be called *support vector clustering - SVC*. This table only has the intention to offer a very brief overview.

.

However, for our application, let us focus on supervised learning with Neural Networks.

2.2 Deep, Convolutional Neural Networks

Neural Networks are mathematical objects modelled after *brains* (human or animal) [9]. The main focus is that they use *connections* to make sense of data. However, let us not expect that this anatomical analogy will go any further, since we do not intend to discuss neurology or the cognition mechanisms of living creatures. The term *deep* is an indication of the fact that the *insides* of these mathematical representations are *hidden* from view, whereas the term *convolutional* is a reference to the fact that the type of networks we will discuss below employ at least one *convolutional layer*. A visual representation is given in *Figure* **??**.

2.2.1 Key Terminology

Therefore, let us continue by introducing some key terminology related to CNN's:

- **Node:** also neuron or cell, this is one unit of the neural network. It is the intersection of *connection lines* inside the network, and is often a place where mathematical operations will be applied to the data.
- **Layer:** this is often described in literature as a level of the network [9]. It can be either *visible* - input and output layer - or *hidden* - located inside the network, in between the two.
- **Input Layers:** this refers to the data being fed into the Neural Network, such as an image; it will usually have been processed into a mathematical object. For instance, inputs are often multi-dimensional matrices, or *tensors*.

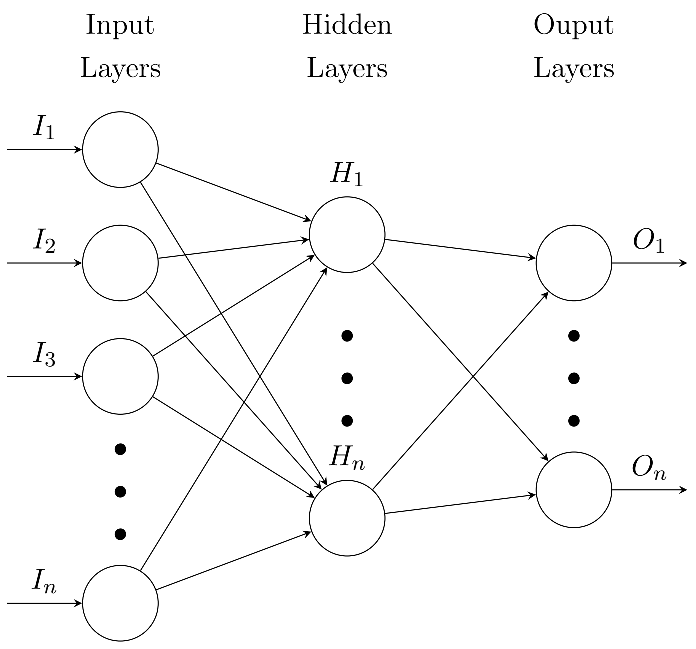


Figure 2.2: A Deep Neural Network. Source: [43]

- **Output Layers:** this represents the result that the neural network produces; for example, a multi-class classifier will deliver different percentages for each class it has to classify an object into, with each percentage representing its rate of confidence that the given input object belongs to that specific class.
- **Convolution:** in our case, this does not refer to the mathematical operation from Fourier Analysis. Instead, convolution refers to *dragging* a filter layer across a matrix or tensor to obtain a different representation of that structure - more formally, applying a linear transformation to a tensor, which results in a different representation, that can be more useful to a computer. By *different* representations, we mean new tensors, such as *feature maps* or *oriented gradient representations*, which help the computer more aptly visualize the trends and patterns in the data, which it can subsequently learn from.
- **Kernel:** this is the *unit* of convolution; it is the *window* that we drag over the input data to derive the convoluted representation out of it.
- **Pooling:** by this, we mean the process of *summarizing* or *compressing* several points into a single, new value. For instance, *max-pooling* takes the maximum value from a matrix. *Average-pooling* takes the mean value instead.
- **Weights and Biases:** these are the numerical values that the neural network learns during training, to represent an output *f(x, w)* as a linear combination of the given *n-dimensional* input x and the *n-dimensional* weight vector w, plus the bias:

$$f(x, w) = x_1 * w_1 + ... + x_n * w_n + b \tag{2.1}$$

- **Activation function:** Generally, this will be a Rectified Linear Unit (ReLU) for convolutional neural networks; it is used right before the output layer to eliminate

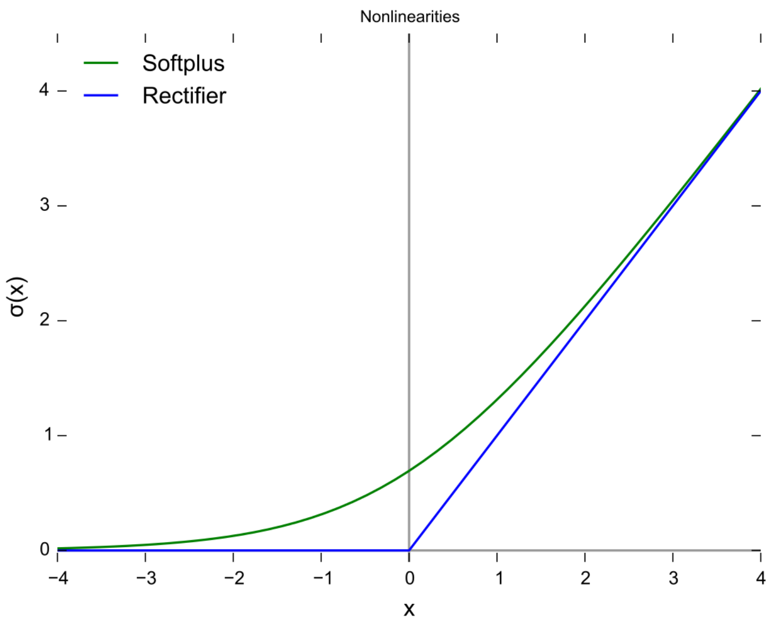


Figure 2.3: A ReLU (blue) and a Softplus (green) activation functions. Source: [2]

negatives and deliver a result inside the correct interval - above zero. Another very popular kind of activation function is Softmax. Examples of both are presented in *Figure 2.3*. [8]

- **Loss:** this refers to the *penalization* given to the NN for delivering results that differ from the expected output. For instance, widely used loss-layers include *sigmoid cross-entropy loss* or *softmax layers*.

2.2.2 On the Efficiency of the Convolution Operation

Given the fact that our model is to run on small, embedded devices, we should not forget that convolution is, at least in its general formulation, a highly inefficient operation, from a computational resources point of view - according to Goodfellow et al, it stands on the scale of $O(n^2)$. Modern applications often also use parallelization to save hardware power. [9]. We should also note, again, the difference between inference and training: in our case, we can only afford inefficient allocation of resources during training, but, for inference, this has to be tightly controlled, since inference must run on MCU's. We will address this problem again in the following chapters. An example for one widely used procedure is *General Matrix to Matrix Multiplication - GEMM*, or *im2col*. This operation implements convolution as linear matrix multiplication but arranges data differently from the standard element-wise order used when calculating per definition, to benefit from the memory access policies of computers. For instance, im2col takes advantage of data locality in RAM and, as such, increases performance, even if this comes with a memory capacity trade-off, given the relatively high amount of created redundant data. [41] A simple numerical example is given in *Figure 2.4*.

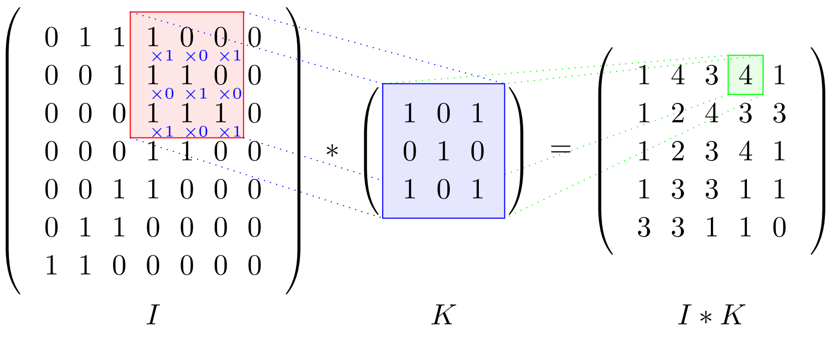


Figure 2.4: An example of the Convolution operation. Source: [1]

2.3 Training, Hyperparameters, and Optimization

Let us look into the training workflow of Convolutional Neural Networks. We will discuss four *parameters* - also known as *hyperparameters* that should be tweaked during training to optimize performance and increase accuracy, or, formulated in a more compact, mathematical way, to minimize loss.

- The number of training *epochs* (repetitions) that we train the network for;
- The size (in number of samples) of the *data batches* that we feed the neural network for every individual step of the training;
- The *divide* between training and testing data; for instance, many applications choose to reserve around 30% of the data for testing, while using the rest for training. It is also possible to reserve a small portion of the samples for what is known as validation - these samples will only be used for inference at the very end to more objectively evaluate the performance of the fully trained and *frozen* neural network;
- The *dropout rate*, which is the number of weights we simply delete out of the network after each training epoch to prevent overfitting.

Let us now investigate 3 other important metrics, which have a high impact on training performance.

- *Generalization error*: this measures the amount of *guessing* a trained model would do; in other words, the more *blank spots* there are in the training data, the more we can expect our model to have an erratic behavior when presented with inputs it had never seen before, during inference.
- *Bias*: this defines the tendency of the trained model to over-perform (in terms of e.g. classification or regression accuracy) for certain categories of inputs, while delivering sub-par performance or accuracy for others.
- *Variance*: this is the statistical *diversity* of data; in other words, this will mathematically represent the scale of *squared difference* we can allow our training samples to divert from their common mean. Higher variance generally means a more versatile and robust system, that can deliver better results on more varied inference inputs.

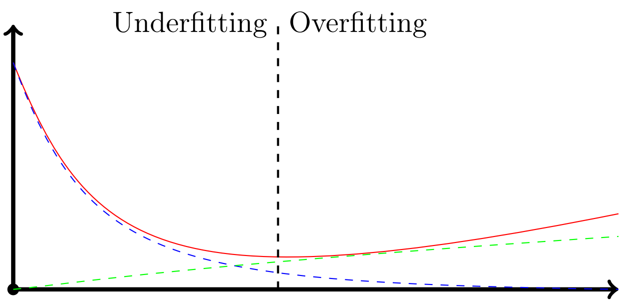


Figure 2.5: Training Neural Networks can result in **underfitting** and **overfitting**; the dashed, vertical line represents the optimal training goal and divides the figure into areas of underfitting and overfitting, respectively. The Red curve is the *generalization error* - which grows with the network's capacity, the blue one represents bias - which decreases with the amount of training, given enough statistical independence in the data - and the green one is the variance achieved.

Let us now discuss *underfitting* and *overfitting*, which are two of the most common problems we must overcome during fitting.

- **Underfitting** is the problem that arises when the training data fails to represent the model in its full complexity. This means that there will be missing features or certain intervals of the *to-be-predicted function* are simply left out, and it is up to the algorithm to fill out these blank spots - often having less than ideal outcomes in terms of accuracy; the model will simply not know enough to make accurate calls. Underfitting can be caused either by simply having too little training data, in which case we need to increase the amount (and quality) of training data, or, in other cases, by an inappropriate choice of hyperparameters, such as too low of a number of training epochs, in which case the model does not have enough time to train.

- **Overfitting** is the exact opposite problem and it arises when the training fails to give the neural network a wide enough view of the problem for it to be able to generalize on new, real-life situations. This can also be seen as the model having focused so hard on the *minutia* of the training data, that it misses the *bigger picture*. Models trained in this manner will perform excellently on the training and testing data but will deliver poor results in the real-world. This is caused by either too little variance, or *diversity* in the training data - alas *no big picture*, or by a poor choice of hyperparameters such as batch sizes or training epochs - training for too long and with too little data in every step will likely result in overfitting.

The optimal point where we want our trained model to be is, therefore, right in the middle: neither too general nor spread out too thin, nor too specialized on the minutia of training data and *missing the bigger picture*. We want it to be just rightly balanced between these two; this is where neural networks tend to perform best. This is represented in *Figure 2.5*. What we obtain is a *U-shaped* curve for the generalization error of the trained network model, represented in *Figure 2.5*. We want our trained model to fall somewhere in the middle for optimality, as per *Goodfellow et al.* [9]

No.	**Name**	Description
1	2D Convolution	
2	Activation Layer	ReLU
3	2D MaxPooling	maximum of a region
4	2D Convolution	
5	Activation Layer	ReLU
6	2D MaxPooling	maximum of a region
7	Flattening Layer	
8	Dense Layer	fully-connected neurons
9	Activation Layer	ReLU
10	Dense Layer	fully-connected neurons
11	Activation Layer	Softmax

Table 2.2: A classifier Convolutional Neural Network architecture

2.4 Final Notes

Let us reiterate the main points that have been addressed in this chapter. We started out by discussing the problem Machine Learning tries to solve, and defined ML as a mechanism that tries to correctly *fit* or *estimate* an unknown function, and implicitly minimize the amount of *difference* there is between the *estimated function* and the real, *unknown function.* For our specific use case, we need a Convolutional Neural Network mechanism that can *classify* the given input into one out of several given categories; the model comprises of *nodes* and *layers*, representing *weights* and *biases*, that take an image as input and deliver percentages as outputs. After designing a model, we *train* and *test* it with data, evaluate its performance by certain *metrics*, then change and *drop* parameters accordingly, to improve its *accuracy*, and *minimize its loss.* A sample convolutional neural network model architecture can be seen in *Table 2.2.*

Having now achieved a certain level of understanding of the fundamentals of Machine Learning, and Convolutional Neural Networks, we can now confidently dive into the next chapter, where we will discuss the reasons why we want to bring ML down into the embedded world of tiny devices, as well as which technical problems we encounter along the way - while proposing solutions for most of them, too.

Chapter 3

TinyML: Machine Learning on Microcontrollers

Why might we ever consider bringing increasingly complex algorithms, with more storage and power overhead, to some of the smallest electronic devices, if their power is so low, and their storage capacity is already so limited, you might be rightfully wondering? Why not simply get more capable hardware and run everything remotely in a server room, where there are virtually no hardware constraints? There are several interesting reasons for approaching that process *differently.*

3.1 Motivation

Firstly, embedded devices and systems have a relatively low production cost and have achieved a high-penetration rate in many markets and industries [34]. They are already widely-deployed, used and well-known. With this in mind, it is only logical to try and bring new capabilities to existing hardware, and extend its range of applications even further. [20]

Secondly, we should look at the hardware of embedded devices: they are, in essence, microcontrollers (MCU's) with connected sensors, forming packages which are called, in Internet-of-Things (IoT) parlance, *edge devices* [42]. The implications can be significant: by bringing ML out of server rooms and closer to *metal* level - we can achieve a high level of data filtration and quality earlier on in the data flow of our systems, while getting more efficient data processing and, very importantly in this day and age, increasing data privacy, since the data is already getting processed (maybe also anonymized) directly at the sensor level, and parsed into a more meaningful format, instead of being dumped in close to raw format into a bigger, more vulnerable, network, to be first processed on an off-site, remote, system. [26]

These points should lay to rest most arguments against data processing right at the edge level of connected and IoT systems. Moreover, since data processing is, as we have seen in Chapter 2, a terrific application for Machine Learning, and considering the amount of interest in this topic at the present moment, we can safely conclude that tiny machine learning, *TinyML*, is worth pursuing. On this note, let us dive into the technical aspects.

3.2 Requirements of the Microcontroller, Edge Device World

The *TinyML* book defines its range of target devices as those being supplied with a power on the scale of *milliwatts* or less, and, very importantly, running 32-bit architectures, having some few hundreds of kB's, or up to a maximum of some megabytes, of RAM and a capacity of binary flash storage on a similar scale. [31]

Given the above definition, we can directly formulate two main problems that have to be addressed if we want to have ML on an MCU: on the one hand, how do we overcome the performance limitations, and on the other hand, how do we fit these models into memory.

3.2.1 What Happens in Memory

In this section, we will offer an overlook of the memory hierarchy of microcontrollers. With this, we hope that the role of the optimization steps undertaken later on will become more clear.

As we have seen in *Chapter 2*, it is very important to optimize our applications in terms of memory use, to be able to capitalize on the principle of locality (also known as *locality of reference*). [32] Processors tend to often access the same *cache lines* repeatedly, which in turn makes it a smart move to keep the data in memory, and not delete it immediately after its first use, since those bits of data might actually be needed again after a short while, and it would be inefficient to have to load them again from RAM.

A typical edge device will often have a hierarchical memory structure as most computer systems do. This memory will increase in terms of speed and decrease in terms of capacity, up from the high level of flash and RAM, into the CPU caches, and finally into the registers - which will be the fastest and tiniest, where the actual computations take place. An overview of this memory hierarchy is offered in *Table 3.1*.

There are, in essence, two types of memory locality: spatial and temporal. The first one refers to physically close or neighboring cache segments tending to be accessed more often than those standing further apart, while the second one refers to keeping used data where it was originally loaded or even loading it into still faster memory segments, since this might be called upon again. [7]

This is especially important for linear operations, such as the matrix multiplications we need for convolution, as our code will contain loops that reference arrays or tensors by indices (even if these indices might not always be increasing incrementally, as previously discussed). Let us take, for instance, matrices, the columns of which are represented in the cache as sequences of rows, and we need, for our convolution, to recall one previously called column. Spatial locality will, in this case, be of great advantage. Or let us consider the case of needing the result of a previous addition again, for computing an average pooling layer. In this case, it is a temporal locality that will help us.

3.2.2 Compressing Full-Size Models

In this section, we will address three specific post-training solutions to the above named size-related problems of memory and performance: *Quantization*, *Pruning* and *Compression*.

- **Quantization**: TensorFlow Lite offers 3 different types of post-training quantization: dynamic range, full integer, and float-16; we will primarily focus on full integer quantization, since it is most appropriate for Edge devices, while the other two are more targeted at smartphone applications, and therefore focus less on simple size

Memory Type	Average Capacity	Access Times
CPU Registers	8 - 256 registers	instant
L1 Cache	32 kB	very fast
L2 Cache	128 kB	fast
L3 Cache	256 kB	slower
RAM	512 kB	slower still
Flash	few MB	slowest

Table 3.1: Memory Hierarchy of an edge device. Source: [35]

reduction. Full integer quantization offers a reduction of up to 4 times in size, with, at the same time, up to 3 times faster inference. Both improvements stem primarily from the fact that the model weights are translated from 32-bit floating-point numbers down to 8-bit integers, which literally means that the model is using up around 4 times fewer bits in memory. [38] There will, obviously, be a certain degree of reduction in the dynamic range of the model, but this is a compromise we can account for and prepare for. There are also mixed quantization modes, such as 16-bit activations with 8-bit weights.

- **Pruning**: TensorFlow Lite also offers, in addition to quantization, the option to *remove* parameters from a model that only had a small impact on its output. This way, we can reduce the size even further. It should be noted that pruning must be done before quantization. Pruning essentially eliminates weights that are close to zero, which are especially useful for audio applications, such as speech recognition, where we only really need to focus on those areas of the sound spectrum where human voices are located in terms of frequency. Pruning or trimming insignificant weights from a given model results in a *sparse* model, which is both smaller in size and faster to run. [37]

- **Compression**: This refers to a new representation of the model in memory as a *flat buffer* instead of an oriented graph, as we would have represented it as a neural network. In other words, a TensorFlow Lite model architecture, which was originally a higher-dimensional graph with a complicated edge and node structure, will now be represented as a linear array, which an interpreter can recompose back into the original graph representation. [11]

We have now become familiar with the tools that we need for running *TinyML* and how they overcome the limitations of the embedded world. Therefore, in the next chapter, we will look into the problem of face detection and how it can be addressed with ML techniques. Then, in Chapter 5, we will dive into *TinyFace* and see how we can combine everything we have touched on so far.

Chapter 4

Machine Learning-Powered Face Detection

To understand how machine facial detection works, let us try to reflect on what humans *actually* do to identify faces: their visual sensors - the eyes - capture an image and send it to the brain. The brain, in turn, translates that image into a *thought*, an abstract representation, in which it searches for the existence of features, such as noses, eyes, mouths, hairlines, all being, essentially, high contrast points on a generally oval-shaped background (the face), which itself stands in contrast to a differently-colored background. In similar fashion, neural-network powered face detection takes a digital image, translates it into a representation that is understandable to computers via convolution, then looks for features and identifies them inside that new representation. The more features are present, the higher the confidence our ML-system can express towards the existence of a face in that given input.

If we were to sum up Machine Learning facial detection in one sentence, it would sound something similar to this: a computer translates a picture into a mathematical model, wherein it searches for patterns, which in turn confirm or infirm the existence of a face in a particular region of that specific given picture.

4.1 Classical Approaches for Face Detection

The methods presented below do not use neural networks for achieving their goal. They are based on methods from some years ago, which belong to what could be labeled as *classical* Machine Learning instead. [39]

4.1.1 Viola-Jones

With the advent of point-and-shoot digital cameras in the early 2000s, the manufacturers needed a simple, low-overhead, real-time procedure of identifying faces in an image, that could run on the constrained hardware of a digital camera. Thus, the Viola-Jones procedure (named after its authors) was created. The initial image classifier consists of a 38-layer cascaded network architecture. It was based, in essence, on a set of classifiers, which, would have offered poor accuracy as individual units, but, combined, offered greatly improved detection.

The Viola-Jones classifier was trained on around 4900 face images and 9600 non-face, negatives. It achieved an accuracy of around 95% and can process images at up to 15 frames per second. [25] The face detector was then developed and expanded into a general object classifier by the authors. Although it uses a similar model architecture, the object classifier does not take direct images as inputs anymore, but feature maps instead. [24]

Feature maps are, esentially, some measurable form of a property of, in our case, an image, or at least of some part of one image. [5] In this case, it is edges and contrast differences that let us identify the face as an object inside an image.

4.1.2 Histogram of Oriented Gradients

Another widely-used procedure for classical image classification is the Histogram of Oriented Gradients (HOG). This procedure essentially divides an image up into separate regions based on contrast. Inside these regions, we calculate the gradient - rate of change - in order to determine patterns. The concatenation of these features is saved as a new representation, which indicates the contents as belonging to a certain class, which can be identifiable. The main advantage of HOG is that its accuracy only depends on geometric shapes, but not on chromatic or contrast differences, which would influence the performance of older methods. Images used for HOG are normalized and their dynamic range reduced to further improve accuracy. [22] Mathematically, a HOG detector identifies *oriented Haar wavelets* [13], which are brief oscillations, starting slightly above zero (as absolute values) and soon dropping back to very close to zero again. Picking up on these *signals* is the main mathematical principle HOG depends on. HOG is often used for general shape detectors, where we don't want to use neural networks.

4.1.3 Support Vector Machines

Finally, let us look into SVM's, which are a more sophisticated category of Machine Learning technique that can be used for facial detection. Support vector machines are supervised learning systems, which can separate samples into different planes, and therefore determine which class different objects belong to. A visual representation can be seen in *Figure 4.1.* [6]

4.2 Face Detection with Convolutional Neural Network Classifiers

To understand how face detection with convolutional neural networks works, we need to firstly define *binary classifiers.* Binary classification is the process of assigning one of several different classes to an object, according to a set of rules. In order to find these rules, a neural network needs *labeled* training data, from which it can extract features, and then *learn* which family of objects those attributes belong to. As such, we can formulate the problem of facial detection with convolutional neural networks as a 4-step procedure.

1. Compiling training data
2. Designing a model architecture
3. Training a classifier, model size compression and optimization
4. Performing inference

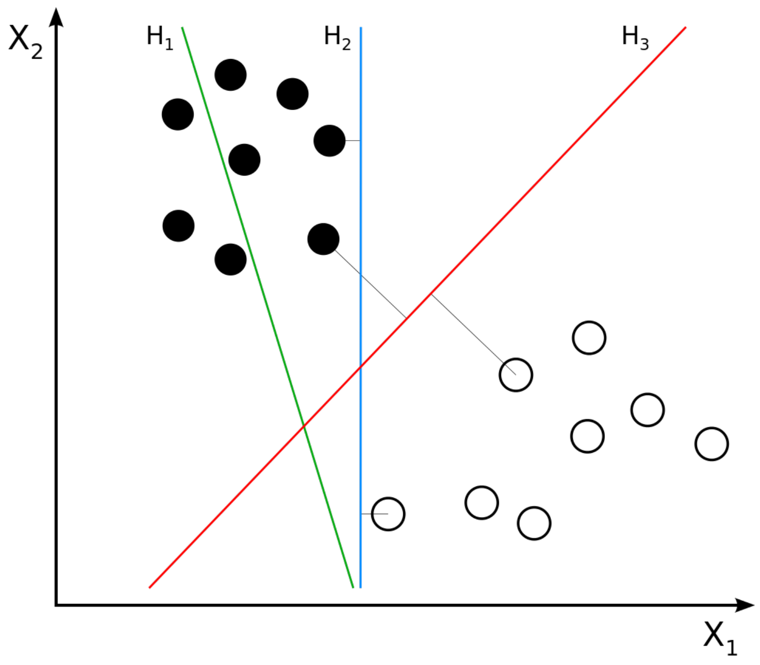


Figure 4.1: Visual representation of a Support Vector Machine. Source: [36]

The first 3 steps will be executed *a priori*, on a separate, more powerful computer. The final step can be performed in an interpreter, or, if required, the model could also be deployed directly to an embedded device.

4.3 Gathering Data for Training Neural Networks

As we have repeatedly seen, gathering relevant and properly selected data for training and validation, which presents high *statistical variance*, is of paramount importance to achieving high accuracy for our final model, when performing inference on real-world data. In the following sub-sections, we will discuss an end-to-end approach for compiling datasets of images of actors' faces, filtering the data, pre-processing the images to the desired dimension and color channels, and finally exporting them into a format that is appropriate for model training.

4.3.1 Compiling a Dataset

Dataset generation is a task that is often best accomplished with automation. Having resolved to also generate our own data for training, a Python scripted tool for web-scraping and image processing was built.

In essence, the tool can search Google and IMDb by actor names, compile a list of links of images of the actors by using Selenium [28] and BeautifulSoup, then download and save the images from those links. The tool then automatically detects the faces in those images with OpenCV, crops them out with a *randomized* padding border to increase the variance of the background, then saves those faces to the desired output shape given by the user. It can save the images as either 8-bit RGB or grayscale.

Dimension	n : 1	x : 2	y : 3	c : 4
Content / Dataset	Number of Samples	X Dimension	Y Dimension	Color Channels
LFW [17]	13,633	250	250	3
Own Dataset 1 (GS)	200,000	32	32	1

Table 4.1: Shape of training data arrays for different datasets as represented in memory.

Dimension	n : 1	x : 2	y : 3	c : 4
Content / Dataset	Number of Samples	X Dimension	Y Dimension	Color Channels
Aligned Images [3]	621,126	340	340	3
YT Faces [44]	5,462,875 videos	1920	1080	3

Table 4.2: Other datasets, which the author has identified and experimented with, but not used for training the final model, due to, amongst others, complexity and hardware limitations.

Using this tool, several training datasets were compiled, and the above-named models were trained using this training data.

The finally compiled dataset consists of two categories of labeled pictures: firstly, images of faces, then, secondly, images of random backgrounds extracted from pictures, which are used as so-called *negatives*. The reason we need this second class of pictures is that a binary classification tool needs, as its name suggests, exactly two classes to fit objects into.

4.3.2 Image data processing

To process the downloaded pictures into a format suitable for face detection, the above presented tool has the ability to, firstly, filter out the faces from the larger downloaded images, than to save those faces as individual pictures, with a border padding that is randomly defined - to help with versatility - then save them, either as color or grayscale pictures and to a desired dimension. Furthermore, we can define custom padding borders for images, including a randomizer that ensures we always set the face slightly off-center.

4.4 Designing a Convolutional Neural Network Model Architecture

For the building of a neural network model architecture, several formerly designed models, from literature, were analyzed to determine their fitness for TinyFace. An overview of some of the most prominent examples the author has experimented with, are listed below. Their respective architectures can be found in *Tables 4.4* to *4.6*.

1. DarkNet [40]

2. LeNet [18]

3. VGG [29]

Model Architecture	TF-Lite model size	Number of Layers
SqueezeNet	4.8 MB	31
DarkNet	4 MB	26
LeNet	4 MB	11
Optimized SqueezeNet	**139 KB**	31

Table 4.3: Comparison of different model architectures in terms of size and number of layers.

4. SqueezeNet [14]

The author has built a Python-based framework, on which many different combinations of datasets and model architectures were trained and tested, with different hyperparameters. The results of this process will be discussed in Chapter 5.

A more advanced neural network model architecture, developed by Google and available open-source, has also been considered. It is called FaceNet, and it was primarily designed for facial recognition, since it can *very* accurately identify facial features of individuals and tell apart one person from another with over 90% accuracy after having trained on just 10 pictures of that person. FaceNet therefore is not only capable of face detection, but also advanced one-to-many facial recognition, and this puts it outside the scope of this project, or the performance of embedded hardware in general. [27]

As we can see in *Table 4.3*, out of the several investigated model architectures, the best TinyFace performance has been achieved by an optimized version of SqueezeNet, the size of which the author has reduced by using TF-Lite quantization and compression.

4.4.1 A Model Architecture That Stands Apart

The used neural network model is based on the *SqueezeNet* architecture. SqueezeNet - which is a classification neural network - manages to achieve very small model sizes, because it uses *concatenations* to merge layers. As such, the network is a sequence of convolutional and pooling layers, plus what its authors call *fire modules* or layers, which use the above named convolutions. An overview of one fire layer can be seen in *Table 4.5*, and the complete architecture of the neural network can be seen in *Table 4.6*.

Other models that the author has experimented with are VGG-3, LeNet and DarkNet, which were presented in the previous chapter. However, the size of these could not be reduced enough to make them fit on the embedded device.

For determining the training parameters, the author experimented by gradually increasing the dimensions and definition of training data until either the performance started decreasing or the computer ran out of memory. For instance, trying to train with anything more than 50,000 color images with sides larger than 100 pixels, could not fit inside the video memory of the used graphics card, and resulted in an error.

Layer
2D Convolution
2D MaxPooling
2D Convolution
2D MaxPooling
"Bottleneck Block"
2D MaxPooling
"Bottleneck Block"
2D MaxPooling
"Bottleneck Block"
2D MaxPooling
"Bottleneck Block"
2D Convolution

Layer
2D Convolution
2D Convolution
2D Convolution
ReLU
2D MaxPooling
2D Convolution
ReLU
2D MaxPooling
Dense Layer
ReLU
Dense Layer
Softmax

Layer
2D Convolution
2D Convolution
2D Max-Pooling
Dropout
2D Convolution
2D Convolution
2D Max-Pooling
Dropout
2D Convolution
2D Convolution
2D Max-Pooling
Dropout
Dense Layer
Dropout
Dense Layer

Table 4.4: DarkNet, LeNet and VGG model architectures

2D-Convolution - Window Size: 1 * 1			
Activation - ReLU			
2D-Convolution	Window Size: 1 * 1	2D-Convolution	Window Size: 3 * 3
Activation	ReLU	Activation	ReLU
Concatenation			

Table 4.5: A Fire Layer [14]

Layer
2D-Convolution
Activation
2D Max-Pooling
Fire Layer
Fire Layer
Fire Layer
Fire Layer
Dropout
2D-Convolution - Window Size: 1 * 1
Activation
2D Average-Pooling
Activation

Table 4.6: SqueezeNet model architecture [14]

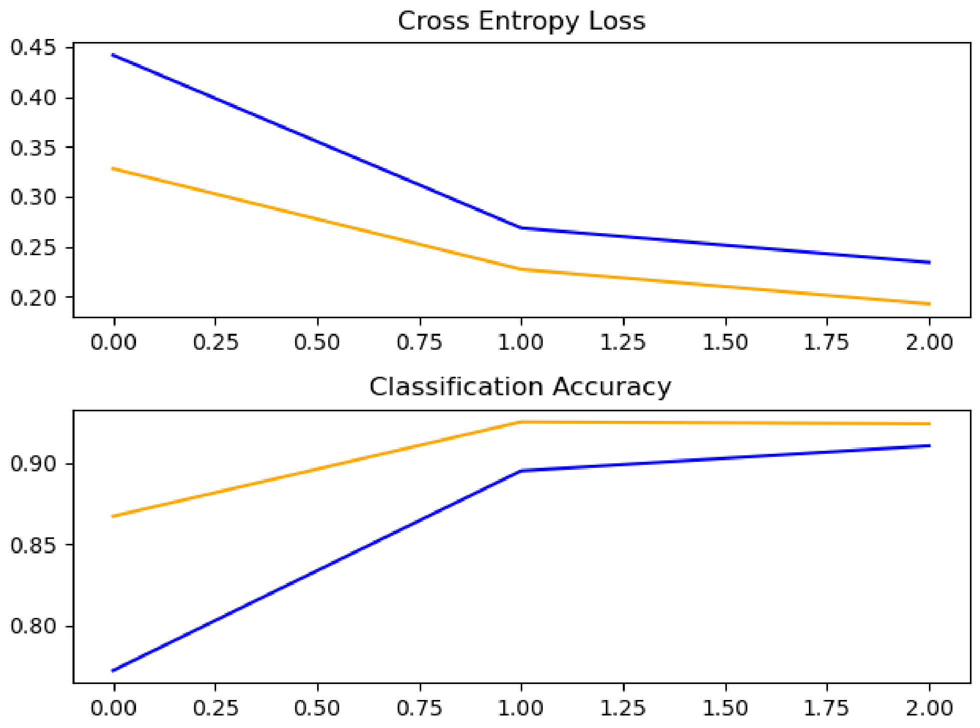


Figure 4.2: Training the final version of the model. A squeezenet-based model architecture, with 3 epochs of training, and a batch size of 32, plus a custom dataset of 16,000 samples.

4.5 Training the Model

Having now seen how a dataset can be compiled, let us look into what training could be like.

4.5.1 Discussing Datasets

We start off with *numpy* arrays of data. These are usually represented as having 4 dimensions in memory, and, therefore, a shape of *n:x:y:c*. A number 3 for Color Channels represents RGB images, a 1 represents grayscale images. For each color channel, we then have a *0-255* value, representing its respective color as an 8-bit number. We can calculate the total, uncompressed, size of the training data array by multiplying the numbers on every row.

Having these rough representations for dataset size, let us now look at how we can actually use this training data.

4.5.2 Tuning Hyperparameters

As we have seen in Chapter 2, hyperparameters such as training batch sizes, number of epochs, training data dimensions, randomness and variance or the amount of shuffling in the dataset, can have a relatively large impact on the final performance of the neural network, when it comes to inference.

We have empirically determined that, in order to avoid over- and underfitting, and achieve acceptable accuracy and performance during inference with actual faces, grabbed

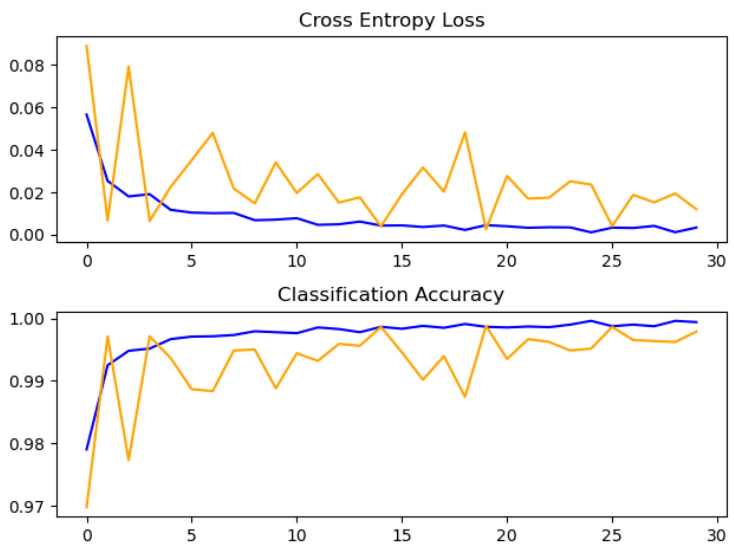


Figure 4.3: Summary of training a Squeezenet-type Neural Network with 30 epochs of training, a batch size of 20 samples and LFW as a training dataset. The yellow curves represent the actual values, the blue ones are median values. The plot represents a full-size Keras model and uses full *float-32* weights.

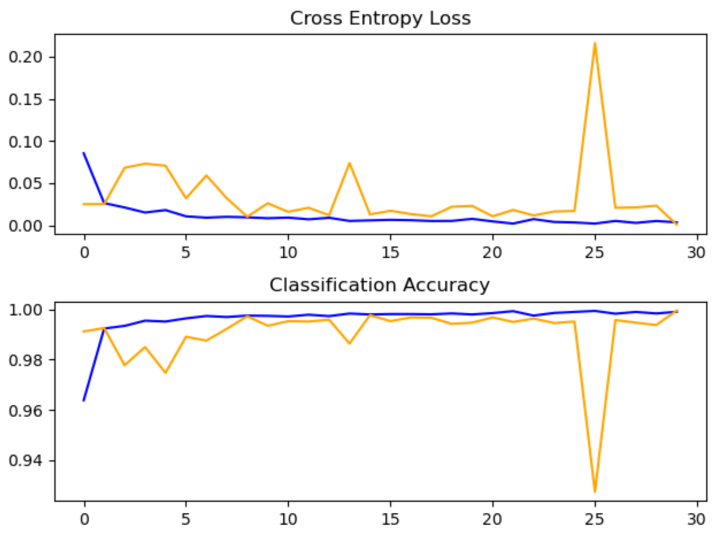


Figure 4.4: Summary of training a Squeezenet-type Neural Network with 30 epochs of training and a batch size of 20 samples. The yellow curves represent the actual values, the blue ones are medians. The plot represents a quantized model and uses uses *uint8* weights. We can see that the TF-Lite model has some pronounced spikes in both accuracy and loss around epoch 20, which might have been caused by the limited representation power of unsinged 8 bit integers.

real-time from a camera, we need around 5 epochs of training, with batch sizes ranging from 16 to 128 samples. Such a low number of epochs has been determined to be necessary to avoid overfitting. A total dataset size of 40,000 labeled samples - out of which 15,000 will be faces and 25,000 will be non-face negatives - with dimensions of *100*100*1*, grayscales therefore, shuffled, and with randomized borders around the faces, have been providing acceptable real-world performance.

Examples of the training process of a SqueezeNet-class model can be be seen in *Figure 4.3* - a full size model and *Figure 4.4*, in which our model is being quantized and compressed during training.

4.6 Inference and Performance Evaluation

For testing the performance of the compiled and trained models, we are using a Python-based TensorFlow Lite model interpreter, which we are feeding images as inputs, to perform inference on. It parses the image arrays through the weights tensor, and outputs confidence percentages as to whether the objects in the images belong to certain classes. The model interpreter can take input images from 2 sources: either from image files, or directly grab frames from the webcam. This application can run on any system which has Python installed and can very quickly help us validate the real-world performance of a model. It will be presented in more detail in Chapter 5.

Having looked into face detection, from both a historical, pre-Convolutional Neural Network, perspective, as well as at modern CNN-powered technologies, and having the knowledge to train, compile and evaluate Neural Network models, then to translate full-size tensors into compressed, embedded-sized, packages, we can now finally enter the next chapter: the implementation of *TinyFace*.

Chapter 5

Implementation

We have previously defined *TinyFace* as a face detector application, powered by neural networks, and running in a TensorFlow-Lite interpreter environment, where certain steps have been undertaken to reduce model size and complexity. This chapter will present a working application where this has been made possible.

As previously discussed, there are essentially 4 steps to building *TinyFace*: gathering data, experimenting with convolutional neural-network model architectures, training, and, finally, evaluating inference performance. With it being an iterative process, this means that after running through all 4 steps, we must go back to step 1 and try improving our model's performance even more.

5.1 Webscrapper

Webscrapper is a Selenium-based automatic web-scraping tool, which can search for, download, pre-process, crop and add custom or random side-paddings, sort and categorize, filter and save images where faces are present. We need Webscapper to build datasets for training, but we will not go into more detail about it. Some examples of custom-generated datasets that were used by the author during training, as well as the shape of their contents can be seen in *Table 5.1.* The best performance was achieved using *Own Dataset 3.*

Another aspect that has impacted the accuracy of the trained model was the amount of padding (background left intact around the face) and the place where it was applied (i.e. if the images were centered relative to the face or if they had been intentionally skewed

Dimension	n : 1	x : 2	y : 3	c : 4
Content / Dataset	Number of Samples	X Dimension	Y Dimension	Color Channels
Own Dataset 1 (GS)	200,000	32	32	1
Own Dataset 2 (GS)	50,000	64	64	1
Own Dataset 3 (GS)	30,000	100	100	1
Own Dataset 4 (Color)	15,000	48	48	3

Table 5.1: For each dataset, we have used almost all of the 4GB of VRAM the graphics card used for training had. The dataset size is proportional to the number of samples and the dimensions of each sample.

Dataset	Amount of Padding	Webcam Demo Accuracy
1	20 %	70-80 %
2	50 %	70-80 %
3	80 %	70-80 %

Table 5.2: Padding was applied as a percentage, divided randomly and unevenly between the left/right and up/down side pairs respectively. All datasets had 30,000 grayscale images with sides of 100*100. The amount of padding has not significantly influenced real-world performance on the webcam demo.

to one side). In *Table 5.2* we can see several different datasets of size 30,000 with side dimensions of 100*100, and the amount of (randomized - i.e. not equal on the two different sides, but randomly distributed and intentionally made uneven) padding aroud the image. This skewed padding gives us more realistic images.

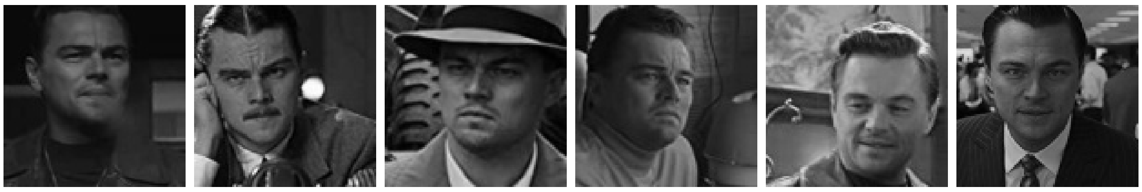


Figure 5.1: Training images with randomized side paddings

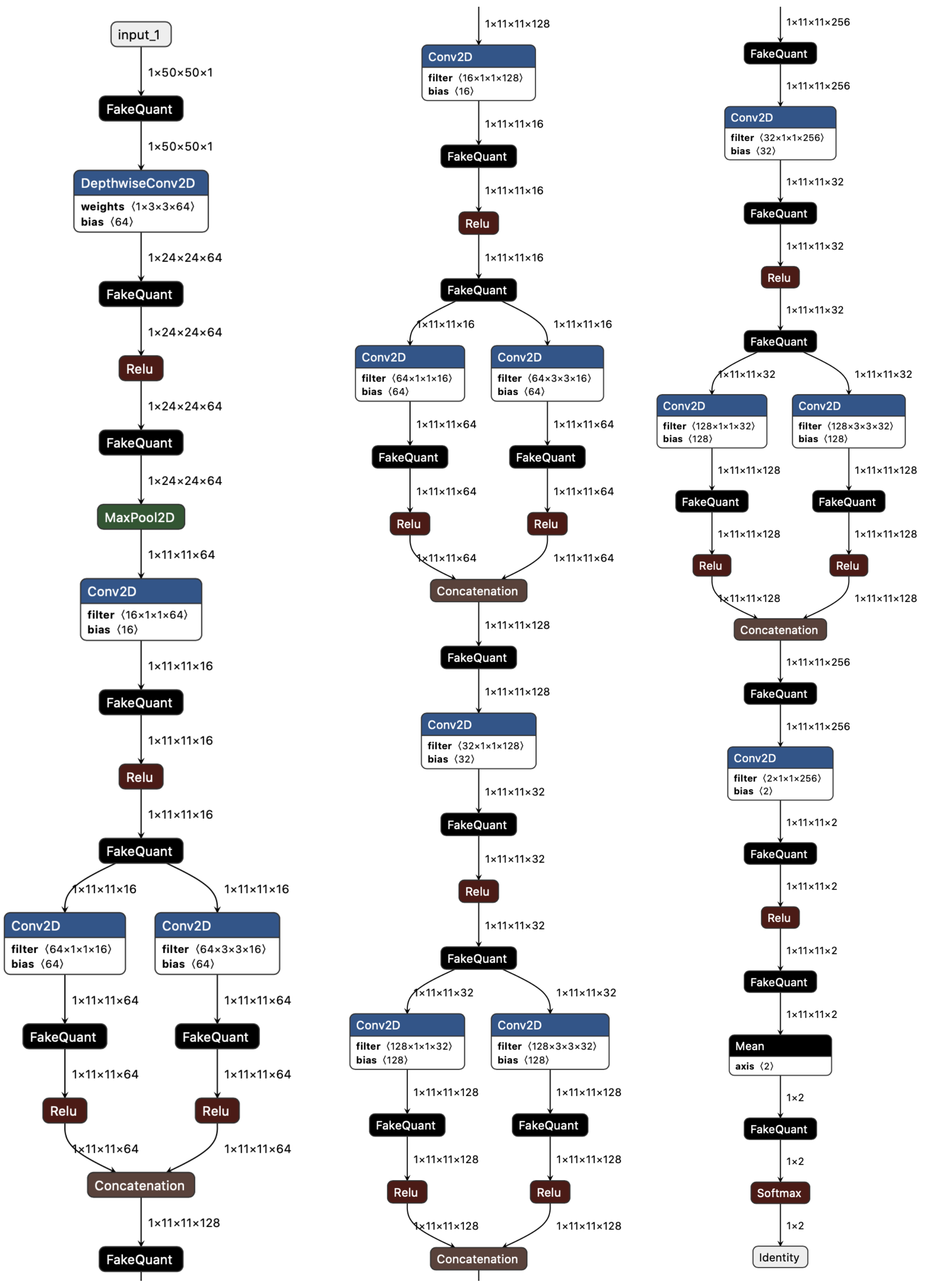
5.2 Compressed Squeezenet

5.2.1 Goals

Having resolved to formalize the training and evaluation of models, several Python scripts have been written to automate this task. Thus, we firstly needed to write up all the identified model candidates in TensorFlow, in order to be able to call them up for training. This was done for all 4 models from the previous chapter. After training and evaluating performance, the best accuracy and also the smallest size were achieved by Squeezenet.

5.2.2 Architecture

We have settled on a modified Squeezenet-type model architecture. In *Figure 5.2* we can find a fully expanded description of the trained and quantized model architecture, in graph form. In essence, we just need to remember from the previous chapter that the model comprises of several fire layers, which come one after the other in sequential order. We should note that the operation *FakeQuant* is still not entirely supported by the stable channel implementations of the TFLite interpreter.



(a) Top Third (b) Middle Third (c) Bottom Third

Figure 5.2: Expanded view of trained and quantized Squeezenet. Note that the image was divided so it would fit on one page. Generated from TF-Lite file by Netron. [23]

5.3 The Model Trainer

5.3.1 Purpose

This script handles loading training data into memory, configuring the models for training (e.g. parsing the correct image dimensions, by inspecting the loaded images) and, if needed, adapting the training data if the format is not completely correct for our purposes (e.g. color ranges vary - we often need to normalize images to a smaller dynamic range), calling the desired model, applying the needed quantization and compression steps, as well as delivering training metrics to help us evaluate accuracy.

5.3.2 Code Architecture

The script essentially calls 3 different methods to accomplish the above described task.

1. **load_dataset**: searches the path for each individual image file, loads it and its label, shuffles images and generated extra data by means of data augmentation - we have determined that this improves the final model accuracy.
2. **run_training**: compiles (*builds*) the model, applies quantization and compression and starts training. It then calls upon the *history* generator to give us an overview of the training process.
3. **summarize_diagnostics**: uses *pyplot* to represent the training analytics as plots.

We can also see the code structure in *Figure 5.3.*

5.4 The Webcam App and Inference

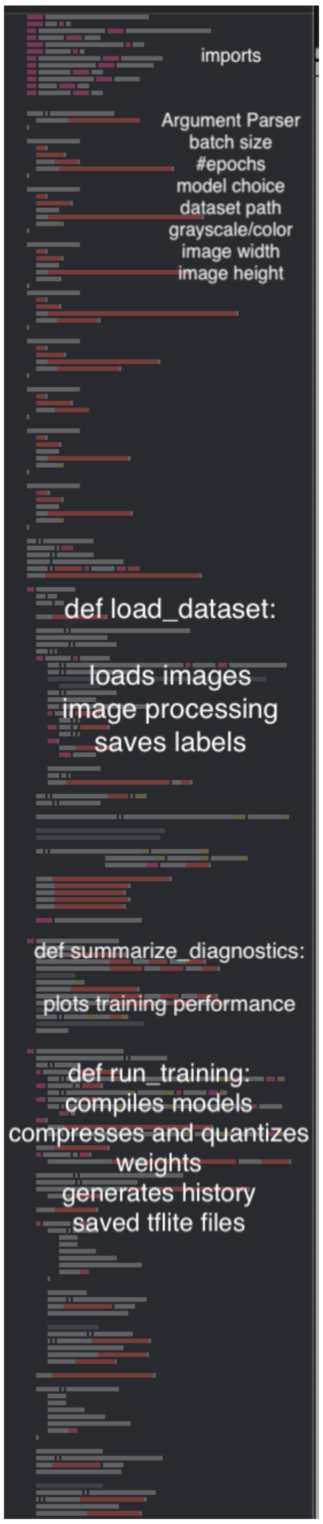
This is the live-demo type, model compiler-based, application where we can put the trained model to the test in real-life, by calling a model interpreter over an image from our webcam. We also have the option to load an image from a local file and run inference on it.

5.4.1 Code Structure

This application uses OpenCV to parse the webcam into image files in real-time, which can then be given to the model interpreter as numpy arrays to perform inference on. There are essentially 2 parts to this program: a method called *face_ detector*, which includes the TensorFlow code for instantiating the model and running inference and a *while True* loop which polls the webcam and passes the input to the above named method. The code structure is again presented in *Figure 5.3.* There is not much image processing involved, we just crop the image to the desired square shape, turn it grayscale (from color) and normalize the dynamic range to a range of 0-10.

5.4.2 Inference Performance

Calling the model statically over any image from the dataset will generally result in a perfect result - we can assume 99% accuracy. This is also the case for 85-90% of other



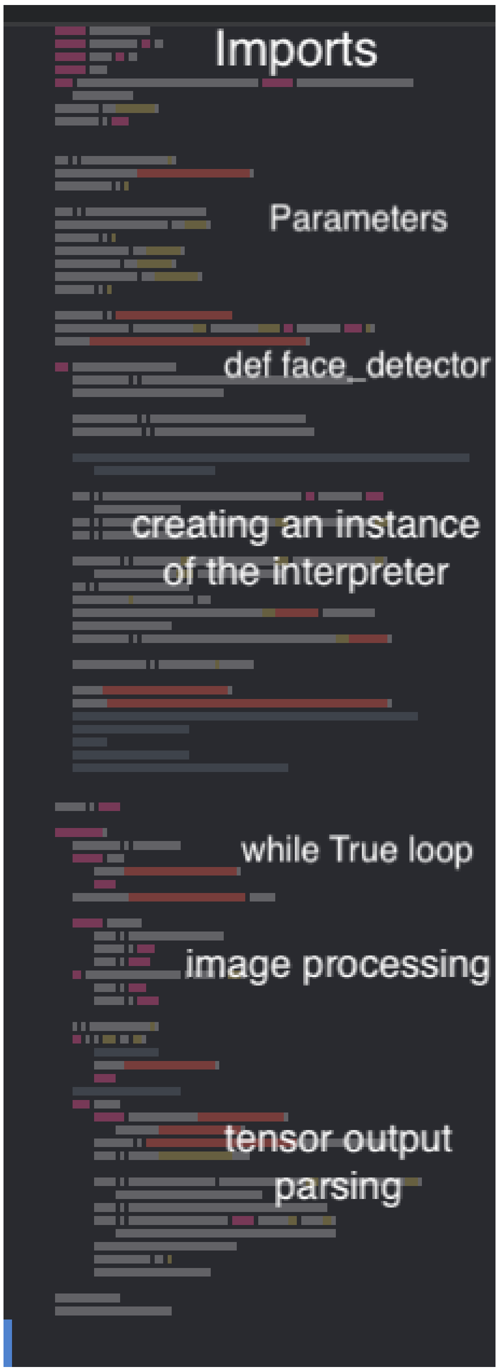


Figure 5.3: Code structure of the model trainer and the webcam app

well-lit and well-centered images, which had not been used for training, where the faces are clear and facing the camera. However, we must note that these are mostly images of actors either in movie scenes or at movie-related events. This means most images are not direct *mug-shots* of faces with uniform backgrounds, as an image from the webcam would be. This translates to the model behaving slightly differently with the webcam app. We have found that it *actually helps* the model in terms of accurately identifying our face if we tilt our head slightly to the side or find a non-uniform background to set ourselves in front of. Smiles tend to sometimes help, too - most training images feature smiling faces, after all, and removing glasses is also very helpful.

Chapter 6

Results

In the previous chapters, we have looked in broader terms at the problems we are faced with when trying to make *TinyFace* a reality. In this chapter, we will look into the combination of decisions and parameter choices that have enabled us to accomplish our goal - the end to end development of a dataset generator, model training framework and *TF Lite* interpreter app.

6.1 The Final Dataset

The model has then been trained on a custom generated dataset, containing data that has been scraped from the web, on a total of 30,000 images - 14,000 faces and 16,000 non-face objects - as such we have a binary classifier. Each image is *100 * 100* pixels large and grayscale. Moreover, each image has been contrast-normalized to a range of 0 to 10, to minimize contrast variance, which would have otherwise been seriously impeding the model's performance during real-world use if left as it had been by default in the images (0-255). A side effect of this, it should be noted, is that the resulting training images look *pitch black* to the naked eye, and we can barely make out any details if at all since there is so little detail left for the human eye to pick up on. However, it is - in this case - advantageous to have so little dynamic range, since the neural network can use its resources to learn the actual geometric shapes and patterns of the faces, instead of wasting valuable memory on learning less-relevant contrast differences (for instance, a soft shadow to the side of one's nose) or, in other words, learning *shades of gray* unnecessarily.

6.2 Optimized SqueezeNet

When choosing which model architecture to use, color images we initially used for experiementation. However, these only really made the model size increase so drastically that the finished files could only barely fit inside the memory of the embedded device. As such, the decision was made to switch to exclusively gray-scale images. This also meant the model complexity was reduced since the convolutional layers only had to be applied to one dimension, instead of three.

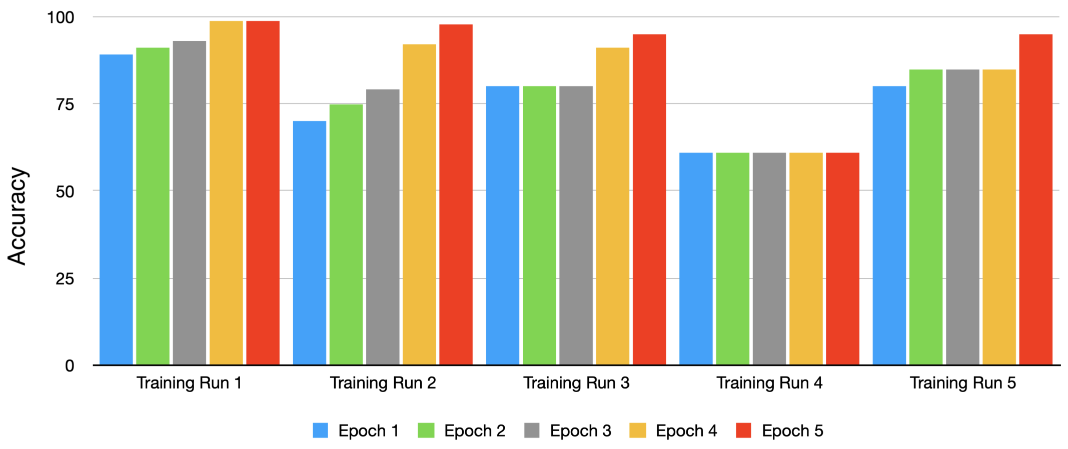
The final model was, as previously mentioned, based on SqueezeNet. The final trained model file is *139 KB* large, after having been compressed and quantified to 8-bit weights.

6.3 The Golden Training Formula

After repeatedly experimenting with different numbers and combinations for each of the hyperparameters, we have settled on a set of values that delivered the highest accuracy. The hyperparameters for the most successful training sessions have been set as follows:

- 5 training epochs
- a batch size of 32 samples
- a 75%-25% divide between training and test data
- a dropout rate of 10%

The choice of hyperparameters and the evaluation of training performance is a lengthy process, and a relatively big time drain, in general, plus highly repetitive. That makes it a prime candidate for automation. A solution for this will be proposed in the next chapter. In *Figure 6.1*, we can see the training accuracy of the model, as reported by TensorFlow. We have achieved 99% accuracy several times on the training data, which means that our model really is a good fit for learning images.



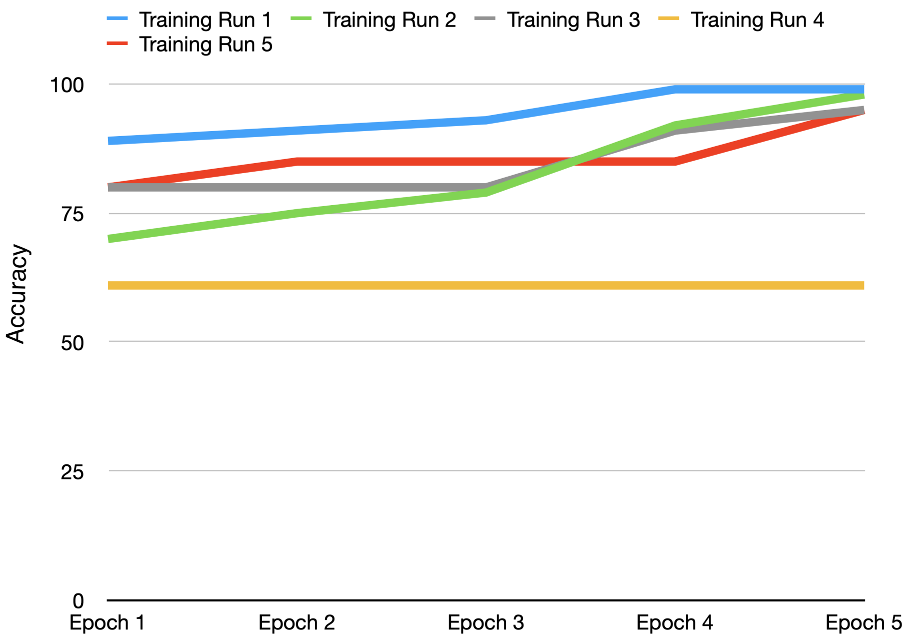


Figure 6.1: Training with 5 epochs, for five different runs. During run number 4, the model failed to learn the data, and averaged out at a reported 61%.

Image	% confidence Face	% confidence Non-Face	Result
1	61 %	38 %	Face
2	48 %	51 %	No Face (it's a tie, almost)
3	63 %	35 %	Face
4	41 %	58 %	No Face
5	37 %	62 %	No Face

Table 6.1: Results of inference when the interpreter was called on the example images.

6.4 Example Inference

Let us look at five representative images of the author, that the interpreter has been called upon. We have 3 figures. In *Figure 6.2*, we can see the original images, as they appear to us in the *live-view* of the webcam. In *Figure 6.3*, we can see what the converted images look like *in the eyes of the interpreter*. In *Table 6.1*, we can see what the interpreter has determined for each image.

For the first image, we have a clear answer: a face. For the second image, probably because of the tilted head, we have a not-so-clear 48-51 divide between face and non-face. The third image, where even glasses are present, is a clear *face*. The fourth image, where the webcam was partially covered, is a non-face object. The same is true for the last image, where the webcam has, once again, been intentionally covered.

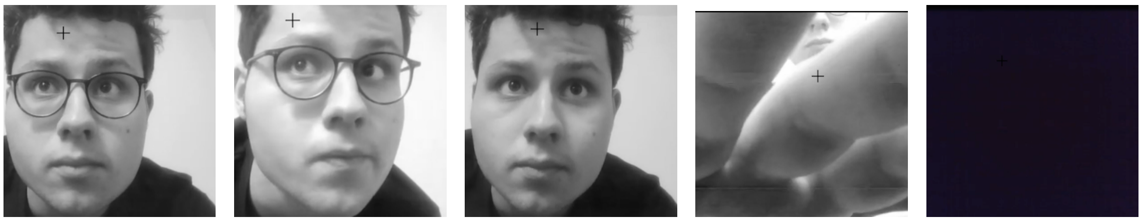


Figure 6.2: Original images, as seen on webcam - cropped by hand to similar sizes for the sake of comparison with the images in the following figure and turned to black and white from color for consistency.

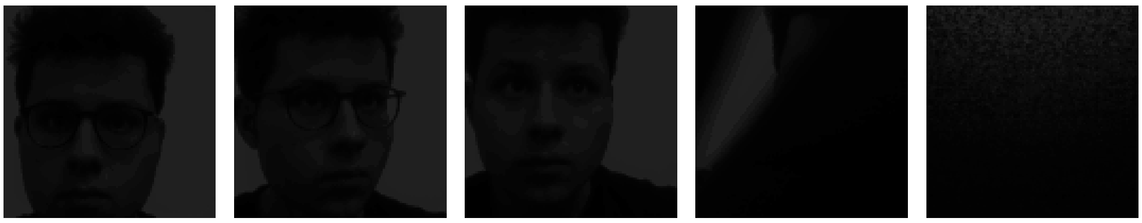


Figure 6.3: What the interpreter actually *"sees"*. The images have sides of 100*100 pixels, their dynamic range has been downsampled to a value between 0 and 9 grayscale.

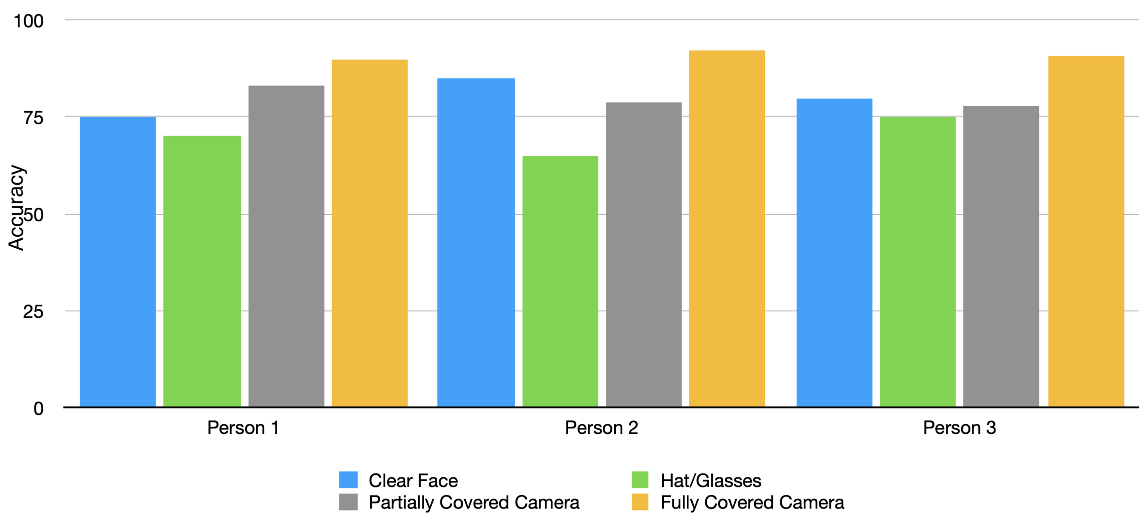


Figure 6.4: Results of another accuracy test for inference: how many of 20 images were correctly identified.

Person	Clear Face	Glasses/Hat	Partially Covered Camera	Fully Covered Camera
1	75 %	70 %	80%	90 %
2	85 %	65 %	80%	90 %
3	80 %	75 %	80%	90 %

Table 6.2: Results from above, transcribed.

6.5 A Longer Test

We have also run inference for longer periods, with the faces of several people, and achieved the following results, presented in *Figure 6.4 and Table 6.2.* These other faces will not be included, since they are not the author's own. We can see that the inference was *relatively* accurate in our empirical tests: often around 80%, and this in 4 different scenarios, with 3 different people.

6.6 Discussing the Results

Let us, therefore, close this chapter by looking back at what has been achieved. We have so far presented a complete, end-to-end, process for running face detection in a TensorFlow Lite interpreter. We started out with compiling our own dataset and developing our own model architecture, then optimized size and performance, deployed to the interpreter, and evaluated performance. As such, we have created a new application of *TinyML*, that we have called *TinyFace*. We have achieved an accuracy level of around 80 % or more in our empirical test, which is to be expected for models of such size on real-life benchmarks. Moreover, let us not forget that systems designers of such systems would often implement several extra safeguards at higher abstraction levels, higher up in the system, to compensate for the crude accuracy of our almost metal-level implementation of Machine Learning.

With this in mind, and having now accomplished what we set out to do, let us close this chapter and proceed to the final one, where we will present possible future research directions, and deliver our closing remarks.

Chapter 7

Conclusion and Future Work

In the past chapters, we have walked through the steps of developing a workflow for building face detection systems for MCU hardware.

This thesis started out with a formulation of the problem of running machine learning-class applications on very constrained edge-devices, specifically neural network type applications for face detection. The second chapter offered a quick introduction into machine learning, whereas the third chapter went into depth about the memory limitations of embedded devices. Chapter 4 helped the reader understand which steps could be undertaken to overcome the limitations of edge devices, as well as how the author decided which model architectures and training data to use for the presented practical project. The fifth and sixth chapters went into specific details about the practical project and presented a discussion of the achieved results and their implications.

This last chapter will be the place where we present possible future research directions for TinyFace, and draw some conclusions.

7.1 Deployment to Embedded Hardware

It is the final goal of the author to have his trained model run on embedded hardware eventually. This will be the goal of future research. So far, we have identified 2 ways to accomplish this. Firstly, we can run an *offline interpreter*, on-device, having the model saved as a flat buffer, instead of a graph, in memory. This is quite versatile and can run most models flawlessly, however, it comes with relatively large memory and performance overheads. As such, it is often too much for STM32 boards to handle. The second option is to use a direct model compiler, which can translate *.tflite* files into *C++* programs, that can compile directly. This option requires even less overhead, since we have no need to load the TensorFlow operations separately into memory. With both options, we estimate that we would use up a maximum of 900 KB of memory in total, with our own model, on top of the usual MCU code.

In addition to running the above-named task, we need some firmware on-device, to handle the hardware inputs (camera capture) and outputs (display and LED handling). To this end, we are planning to use STM's stock firmware, which includes a bare-bones real-time operating system, that still offers scheduling and synchronization mechanisms.

7.2 Automating Model Optimization and Training

There is, in essence, one topic that we think should be addressed here. That is the problem of automatic hyper-parameter optimization. *Grid Search* promises a solution to this issue.

As mentioned in the previous chapters, the choice of hyper-parameters is often approached as a trial-and-error process, where the programmer experiments with different values, incrementally increasing performance until a satisfactory level is achieved. However, given the large impact hyper-parameter choice can have on the final accuracy of inference and how long it can take to find the right combination of training parameters manually, it might actually be smarter to automate this task.

Grid search offers a solution to this problem. [12] As implemented in the *SciKit* library, grid search enables us to try out different hyperparameters with our model during training. We can provide the script with either a list of values we would want it to try out autonomously, or we can, more broadly, define a plane of values (combined set of ranges), where the system will look for optimal values. We also have to provide a performance metric to evaluate the performance and help determine the optimal combinations. We can even train more models in parallel to speed up the process. As such, this has the potential to completely change how we will be doing model training for TinyFace.

7.3 Final Remarks

This work has presented a possible pathway for developing a face detector, with one's own gathered data and any ML model one desires. The contribution to the field of *tiny* Machine Learning is the fact that convolutional neural networks have been added into the model architecture. Tiny Machine Learning is itself a very new field, and the literature on the topic is still being written. The tools themselves are being actively developed and substantial changes are made at leasy once every month to the open-source code and API's that had been used to develop this project. This goes on to show just how much we can add to this field.

This being said, the author hopes this work has at least provided a useful insight into what is possible on such tiny hardware, and what embedded machine learning has to offer.

Acknowledgements

When setting out to work on *TinyFace*, we were right the beginning of the COVID-19 pandemic, back when places were starting to close down, and everybody was ordered to stay home. As of September 2020, when these lines are written, not a whole lot more has become any clearer. Still, I consider it relevant to mention this situation we are in, as it has had its obvious effects on all of us, and this project would have likely looked different too, had it not been for these unusual and strange times.

On another note, I would like to close this work by expressing my gratitude to my supervisor, Alex, for his being always so incredibly positive and helpful in all situations, and for everything he has taught me. Moreover, I would like to thank the chair - RCS - and Dr. Daniel Müller-Gritschneder.

I would also like to thank Mom and Dad for their support. I do feel very privileged for having them.

List of Figures

List of Tables

Bibliography

[1] URL: https://tex.stackexchange.com/questions/437007/drawing-a-convolution-with-tikz.

[2] *Activation functions figure.* URL: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[3] *Aligned Faces Dataset.* 2020. URL: http://www.cslab.openu.ac.il/download/wolftau/aligned_images_DB.tar.gz.

[4] Asa Ben-Hur et al. "Support vector clustering". In: (2001).

[5] Christopher Bishop. *Pattern recognition and machine learning.* 2006.

[6] Corinna Cortes and Vladimir Vapnik. *Support-Vector Networks.* 1995. URL: http://image.diku.dk/imagecanon/material/cortes_vapnik95.pdf.

[7] John L Hennessy David A Patterson. *Computer Organization and Design: The Hardware/Software Interface.* 1993.

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: (2011).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016, pp. 96–161, 362, 440–460.

[10] Ian Goodfellow et al. "Generative Adversarial Networks". In: *Proceedings of the International Conference on Neural Information Processing Systems (NIPS 2014)* (2014), pp. 2672–2680.

[11] *Google Protocol Buffers.* 2020. URL: https://developers.google.com/protocol-buffers/docs/faq.

[12] *Grid Search.* 2020. URL: https://scikit-learn.org/stable/modules/grid_search.html.

[13] Alfréd Haar. *Zur Theorie der orthogonalen Funktionensysteme.* 1910.

[14] Forrest N. Iandola et al. *Squeezenet: Alexnet-level Accuracy with 50x Fewer Parameters and <0.5MB Model Size.* 2017. URL: arxiv.org/pdf/1602.07360.pdf.

[15] Koza J.R. et al. *Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming.* Springer, 1996.

[16] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. "Reinforcement Learning: A Survey". In: *CoRR* cs.AI/9605103 (1996). URL: https://arxiv.org/abs/cs/9605103.

[17] *Labelled Faces in the Wild Dataset.* URL: http://vis-www.cs.umass.edu/lfw/.

[18] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: (1989).

[19] Richard T. Marriott, Sami Romdhani, and Liming Chen. “Intra-class Variation Isolation in Conditional GANs”. In: (2018). URL: https://arxiv.org/pdf/1811.11296.pdf.

[20] Rick Merritt. *TinyML Sees Big Hopes for Small AI.* 2019. URL: www.eetimes.com/tinyml-sees-big-hopes-for-small-ai.

[21] Tom Mitchell. *Machine Learning.* McGraw Hill, 1997.

[22] Bill Triggs Navneet Dalal. “Histograms of Oriented Gradients for Human Detection”. In: (2005). URL: http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf.

[23] *Netron.* URL: https://github.com/lutzroeder/netron.

[24] Michael Jones Paul Viola. “Rapid object detection using a boosted cascade of simple features”. In: (2001). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.6807.

[25] Michael Jones Paul Viola. “Robust Real-time Object Detection”. In: (2001). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.4868.

[26] F. Rao and E. Bertino. “Privacy Techniques for Edge Computing Systems”. In: *Proceedings of the IEEE* 107.8 (2019), pp. 1632–1654.

[27] Florian Schroff, Dmitry Kalenichenk, and James Philbin. *FaceNet: A Unified Embedding for Face Recognition and Clustering.* 2015. URL: arxiv.org/pdf/1503.03832.pdf.

[28] *Selenium Official Website.* 2020. URL: www.selenium.dev.

[29] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: (2015). URL: https://arxiv.org/pdf/1409.1556.pdf.

[30] Daniel Situnayake and Pete Warden. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers.* O’Reilly, 2019, pp. 127–354.

[31] Daniel Situnayake and Pete Warden. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers.* O’Reilly, 2019, p. 3.

[32] William Stallings. *Computer organization and architecture : designing for performance.* Prentice Hall, NJ, 2010.

[33] Kenneth O. Stanley and Risto Miikkulainen. “Efficient Reinforcement Learning through Evolving Neural Network Topologies”. In: (2002). URL: http://www.cs.utexas.edu/~ai-lab/pubs/stanley.gecco02_1.pdf.

[34] statista.com. *Embedded computing market value worldwide 2018-2027.* 2018. URL: www.statista.com/statistics/1058799/worldwide-embedded-computing-market-revenue.

[35] *STM32 Specifications*. 2020. URL: https://www.st.com.

[36] *SVM image from Wikipedia*. 2020. URL: https://en.wikipedia.org/wiki/Support_vector_machine.

[37] *TensorFlow Lite - Trimming insignificant weights*. 2020. URL: https://www.tensorflow.org/model_optimization/guide/pruning.

[38] *TensorFlow Lite Quantization*. 2020. URL: https://www.tensorflow.org/lite/performance/post_training_quantization.

[39] *The Importance of classical Machine Learning Algorithms*. URL: https://www.quora.com/How-practically-relevant-are-classical-machine-learning-algorithms-like-SVMs-or-decision-trees-as-compared-to-deep-learning.

[40] *Tiny Darknet*. 2020. URL: https://pjreddie.com/darknet/tiny-darknet/.

[41] Pete Warden. *Why GEMM is at the heart of deep learning*. 2015. URL: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/.

[42] Andrew Wheeler. *How Engineers Are Using TinyML to Build Smarter Edge Devices*. 2020. URL: https://new.engineering.com/story/how-engineers-are-using-tinyml-to-build-smarter-edge-devices.

[43] Mark Wibrow. URL: https://tex.stackexchange.com/questions/153957/drawing-neural-network-with-tikz.

[44] *YouTube Faces dataset*. 2020. URL: https://www.cs.tau.ac.il/~wolf/ytfaces/.