Tampere University

Leevi Oranen

UTILIZING DEEP LEARNING ON EMBEDDED DEVICES

Master of Science Thesis
Faculty of Medicine and Health Technology
Examiners: Antti Vehkaoja
David Hästbacka
August 2021

ABSTRACT

Leevi Oranen: Utilizing deep learning on embedded devices
Master of Science Thesis
Tampere University
Biomedical Engineering, MSc
August 2021

The aim of this thesis was to review the tools needed for the development of deep learning application on an embedded system and how this can be done in practice. The most important and used frameworks (TensorFlow, Keras, PyTorch, Caffe and MATLAB) for building and training machine learning models were reviewed. Another aim was to study and compare the development experience and performance of four different single-board computers.

The machine learning tools studied were comparable in model training performance. Each tool has its strengths and weaknesses. Keras is easy to use and beginner friendly, but the customizability is limited. PyTorch, on the other hand, is very customizable, but requires more understanding about machine learning. TensorFlow works well with TensorFlow Lite, enabling model optimization on mobile and embedded devices.

When choosing a tool, the compatibility of devices plays a very important role. If the same program needs to be rewritten separately for each device, this will be very expensive for the company. This compatibility can be improved by favoring devices that support the most common standards such as Khronos Group standards.

The tested devices were Google Coral, NVIDIA Jetson Nano, NXP S32V234 and Raspberry Pi 4. The test application used in this thesis was emotional detection which consists of two parts. First, a face had to be found in the image, after which the face was cropped and fed into the emotional detection model. The the devices were compared with three parameters: the time taken to detect the face, the time taken to identify the emotion, and the number of processed frames per second.

The test consisted of two variables: input source and face detection algorithm. The used input sources were live stream and the pre-recorded video. The face detection algorithm was performed both with the “Haar cascade object detection” (HCOD) algorithm and in the most optimal way for each device. For example with Google Coral this means that the optimized face detection model from Google Coral’s website was used. The test results with optimal face detection were not easily comparable because the implementation changed so much.

As a result, the NXP S32V234 had the best performance. However, the programming development on that device was challenging, which meant that not all tests could even be performed. The optimal test performance with a camera stream varied between 25,4 FPS with S32V234 to 10,9 FPS with Jetson Nano. Google Coral, NVIDIA Jetson Nano and Raspberry Pi 4 were more or less in the same category in the test with HCOD face detection.

As these tests show, embedded devices have become powerful enough to perform heavy deep learning calculus. This opens up new opportunities for many research areas to make human lives healthier, happier and safer.

Keywords: Deep Learning, Machine Learning, embedded device, Linux, GPU, CPU, TPU

TIIVISTELMÄ

Leevi Oranen: Syväoppimisen hyödyntäminen sulautetuissa järjestelmissä
Diplomityö
Tampereen yliopisto
Biolääketieteen tekniikka, DI
Elokuu 2021

Tämän työn tavoitteena oli esitellä tarvittavia työkaluja syväoppimista hyödyntävän ohjelman kehittämiseen sulaututetussa järjestelmässä ja testata joitakin käytännössä. Työssä käytiin läpi tärkeimmät ja käyteyimmät työkalut (TensorFlow, Keras, PyTorch, Caffe ja MATLAB) koneoppimis mallin rakentamiseen ja kouluttamiseen. Toinen tavoite oli tutkia ja vertailla neljän eri yhden piirilevyn tietokoneen kehityskokemusta sekä suorituskykyä.

Tutkitut koneoppimistyökalut ovat suorituskyvyltään samankaltaisia neuroverkkojen koulutuksessa. Jokaisella työkalulla on omat vahvuutensa ja heikkoutensa. Keras on helppokäyttöinen ja aloittelija ystävällinen, mutta muokattavuus on rajallista. PyTorch puolestaan on hyvin muokattava, mutta vaatii enemmän ymmärrystä koneoppimiesta. TensorFlow toimii hyvin TensorFlow Liten kanssa, mikä mahdollistaa mallien optimoinnin mobiililaitteissa ja sulaututetuissa järjestelmissä.

Työkalujen valinnassa yhteensopivuus laitteiden on erittäin tärkeässä roolissa. Jos sama ohjelma pitää kirjoittaa erikseen jokaiselle laitteelle tämä tulee yritykselle todella kalliiksi. Tätä yhteensopivuutta voidaan parantaa suosimalla laitteita, jotka tukevat yleisimpiä standardeja kuten Khronos Groupin satndardeja.

Testatut laitteet olivat Google Coral, NVIDIA Jetson Nano, NXP S32V234 ja Raspberry Pi 4. Testiohjelmana käytettiin tunnetilan estimointia, joka koostui kahdesta osasta. Ensimmäiseksi kuvasta piti löytää kasvot, jonka jälkeen kasvot rajattiin ja syötettiin tunnetilan estimointi mallille. Laitteita verrattiin kolmella parametrilla: ajalla, joka kului kasvojen tunnistamiseen, ajalla, joka kului tunnetilan tunnistamiseen sekä prosessoitujen kuvien määrällä sekunnissa.

Testissä oli kaks muuttujaa kuvan lähde sekä kasvojentunnistusalgoritmi. Lähteenä käytettiin sekä suoraa kuva että ennalta tallennettua videota. Kasvojentunnistusalgoritmi tehtiin sekä “Haar cascade object detection” (HCOD) algoritmilla, että jokaiselle laitteelle optimaalisimmalla tavalla. Esimerkiksi Google Coralin tapauksessa kasvojen tunnistuksessa käytettiin Coralin nettisivuilta haettua optimoitua mallia. Tästä jälkimmäinen ei ole helposti vertailtavissa, sillä toteutukset erosivat niin paljon toisistaan.

Tulokseksi saatiin, että NXP S32V234 oli suorituskyyltään paras. Kuitenkin ohjelmointikehitys kyseisellä laitteella oli haastavaa, minkä vuoksi kaikkia testejä ei voitu edes suorittaa. Optimaalisen testin tulos suoralla kuvalla Coralille oli 18,9 FPS, Jetson Nanolle 10,9 FPS ja S32V234 25,4 FPS. Google Coral, NVIDIA Jetson Nano ja Raspberry Pi 4 olivat kutakuinkin samassa luokassa HCOD testissä.

Kuten nämä testit osoittavat, sulautetuista järjestelmistä on tullut riittävän tehokkaita, jotta niitä voidaan käyttää raskaaseen syväoppimiseen. Tämä avaa monille tieteenaloille uusia mahdollisuuksia tehdä ihmisten elämästä terveempää, onnellisempaa ja turvallisempaa.

Avainsanat: syäoppiminen, koneoppiminen, sulautettu järjestelmä, Linux, GPU, CPU, TPU

CONTENTS

1 Introduction 1
2 Deep learning on embedded devices 3
2.1 Hardware used for deep learning 7
2.1.1 Cloud computing vs End-device computing 8
2.1.2 Hardware acceleration 9
2.2 Deep learning in automotive industry 10
2.2.1 Driver monitoring system 10
2.2.2 Self-driving cars 11
2.3 Deep learning in Medical industry 12
3 Deep learning software development on embedded device 14
3.1 The Deep Learning development workflow 14
3.2 Software stack for DL on embedded device 17
3.2.1 Deep Learning training frameworks 18
3.2.2 File exchange 25
3.2.3 Deploying neural network models on embedded device 27
3.2.4 Processing unit API 28
3.3 Deep Learning Framework Compatibility 31
4 Hardware test materials and methods 33
4.1 Test methods 33
4.2 Google Coral 35
4.3 NVIDIA Jetson Nano 36
4.4 NXP S32V234 38
4.5 Raspberry Pi 4 39
5 Hardware test results 41
6 Discussion and conclusions 45
References 47

LIST OF FIGURES

1.1 All the tested boards are shown. The first row: Google Coral Dev board, Raspberry Pi 4, Second row: NVIDIA Jetson Nano, NXP S32V234 2

2.1 Relationship between Artificial intelligence, Machine learning, Neural Network and Deep Learning 3
2.2 The basic difference between training and inference with deep learning model is that in training labeled data is fed to DL model and the model tries to minimize the prediction error. In inference trained model is used to classify unknown data. [6] 4
2.3 One Neural network node 5
2.4 Popular activation functions [10] 5
2.5 VGG convolutional neural network [12] 6
2.6 Finding face features in the 2012 ImageNet challenge [13] 6
2.7 Illustration of cloud, edge and device computing [14] 7
2.8 Visualized difference between car automation levels [33] 11

3.1 Illustration of tools used in ML and DL development and computing. 17
3.2 Summary of the most used DL frameworks with GPU support [48] 19
3.3 ONNX enable conversions between frameworks and model optimization for various hardware [69]. 26
3.4 Neural Network Exchange Format supports the most popular training frameworks and is a intermediate step to different inference engines [73]. 27
3.5 Overview of processing unit APIs. Abstraction level increases towards the top. Modified from [79] 29
3.6 The main differences between OpenCV and OpenVX. Modified from [79]. . 30

4.1 Haar Cascade features 34
4.2 Every pixel in the integral image is sum of pixels in its left and above. . . . 34
4.3 Google Coral Dev Board (left) and USB accelerator (right) [93] 36
4.4 NVIDIA Jetson Nano Developer Kit [97] 37
4.5 NXP's S32V234 Vision and Sensor Fusion Evaluation Board [100] 38
4.6 Raspberry Pi 4 with Camera Module V2 40

LIST OF TABLES

2.1 Comparison of end-device and cloud computing. Modified from [15] 8

3.1 Summary of the most used DL frameworks 25

4.1 Summary of the SBCs used in the performance test 40

5.1 HCOD test with camera stream: the values are averages of one frame processing. 41
5.2 HCOD test with standard video: the values are averages for one frame processing 41
5.3 Optimal test with camera stream: the values are averages for one frame processing. 42
5.4 Optimal test with standard video: the values are averages for one frame processing 42

LIST OF SYMBOLS AND ABBREVIATIONS

ACC	Adaptive Cruise Control
AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DL	Deep Learning
DMS	Driver Monitoring System
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HCOD	Haar Cascade object detector
HW	Hardware
IDE	Integrated Development Environment
MCU	Microcontroller Unit
ML	Machine Learning
NHTSA	National Highway Traffic Safety Administration
NN	Neural Network
NNEF	Neural Network Exchange Format
ONNX	Open Neural Network Exchange
SAE	Society of Automotive Engineers
SBC	Single-board Computer
SW	Software
TF	TensorFlow
TPU	Tensor Processing Unit

1 INTRODUCTION

The automotive industry is going to experience great changes over the next few years. Global Market Insights' prognosticators predict that the markets for artificial intelligence in automotive industry will increase from today's (2020) $1 billion to $12 billion by 2026 [1]. This would mean a 200 % growth in just six years. In addition to automotive industry, also the medical industry also offers great possibilities for embedded deep learning applications from epileptic seizure detection to malaria detection in rural areas of Africa [2, 3].

One of the greatest challenges for this prediction is that AI and deep learning (DL) models tend to be computationally heavy. There are cloud servers that could be used to overcome the calculation part but cloud computing will create other issues such as latency, reliability and cost. For this reason it is important to find solutions where safety critical systems are not dependent only on cloud computing.

The recent rapid development of Single-board Computers (SBC), such as Raspberry Pi, has enabled the execution of computations on powerful, low-energy consumption Internet of Things devices. Simple tasks such as face detection and recognition and action recognition are nowadays easy to implement and they could make people's lives easier and potentially save hundreds of thousands of lives by alarming sleeping or distracted driver or by bringing diagnostic tools to people that are not able to get to the hospital because of long distances or other reasons.

The aim of this thesis is to present hardware and software tools intended for embedded deep learning inference. In this work only a limited amount of these tools are assessed.

A concrete objective is to compare how do the SBCs compare in DL image recognition performance and in developer experience. Figure 1.1 presents the tested SBCs which are Google Coral Dev board, Raspberry Pi 4, NVIDIA Jetson Nano and NXP S32V234. All of them are commercially available.

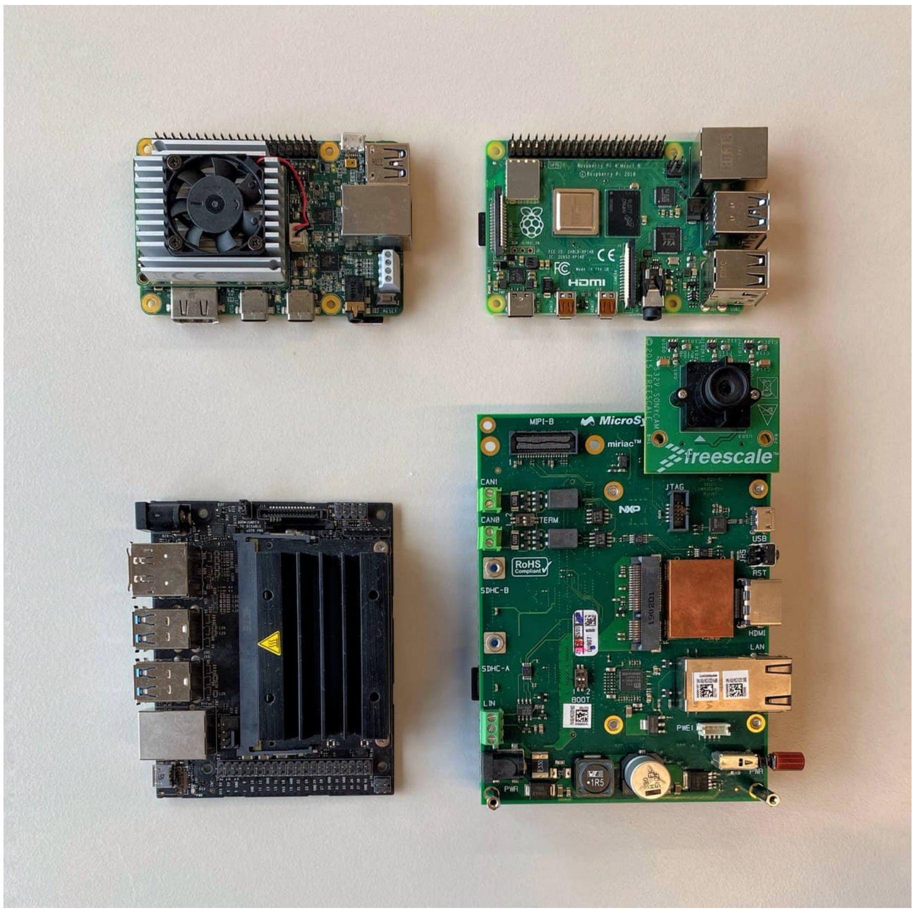


Figure 1.1. *All the tested boards are shown. The first row: Google Coral Dev board, Raspberry Pi 4, Second row: NVIDIA Jetson Nano, NXP S32V234*

This work presents the current state of this subject in 2020-2021. The field of computer science, especially the AI field, is developing rapidly and some of the tools and statements may not be valid anymore in the near future.

The thesis is organized as follows: Chapter 2 introduces deep learning in general, gives an overview of computational requirements for embedded devices and discusses the most used solutions for it. In addition, it presents real-world applications as well as present and future opportunities in the automotive and medical industries. Chapter 3 introduces the software tools which are relevant in this kind of software development. The tools are presented in the same sequence as they are utilized in the development process. It also reviews some close to hardware Application Programming Interfaces (API) even if these APIs are commonly handled by higher-level tools. Next, Chapter 4 explains the used test algorithms and discusses the development tools and experience of the tested SBCs. In addition, the test results are presented there. Finally, Chapter 6 concludes the work and presents the main findings of this study.

2 DEEP LEARNING ON EMBEDDED DEVICES

Our modern world creates a significant amount of data all the time. This data can be used to train deep learning models and help humans' everyday life. But what is deep learning? How can it be utilized it and implemented on embedded devicea? These questions will be answered in this chapter.

At first we need to define what deep learning means. Figure 2.1 shows how Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) are related to each other. AI is often defined as "the science of making computers do things that require intelligence when is done by humans" [4]. ML is a subset of AI. There are several different kinds of algorithms in ML such as support vector machines and artificial neural networks (NN). The name "Neural Network" comes from human biology because it resembles the human neuron system with connections between nodes. Deep Learning algorithms are just complex multi-layered neural networks. The word "deep" refers to the number of layers. In NN there are only a few layers and in DL there are several. [5]

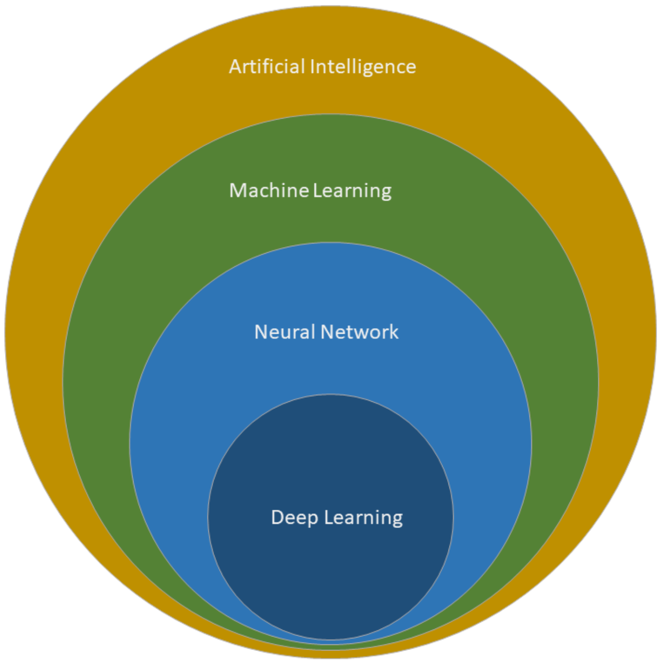


Figure 2.1. *Relationship between Artificial intelligence, Machine learning, Neural Network and Deep Learning*

Deep Learning, as well as in general ML, consists of two phases: training and inference. During the training phase labeled data is fed to the DL model. As shown in the top part of the Figure 2.2, the model tries to predict the labeled image and the error of the prediction is used to update the model. DL training requires massive amounts of computational power and data. Therefore, training is usually done on powerful machine or in the cloud. [6]

Inference is the process in which a trained DL model is used in an application that it was designed for [6]. In Figure 2.2, the picture of a bicycle is fed to the model which predicts what is shown in the picture. Inference requires much less computation power than training. However, it might be a challenge for small computers such as those used in embedded devices.

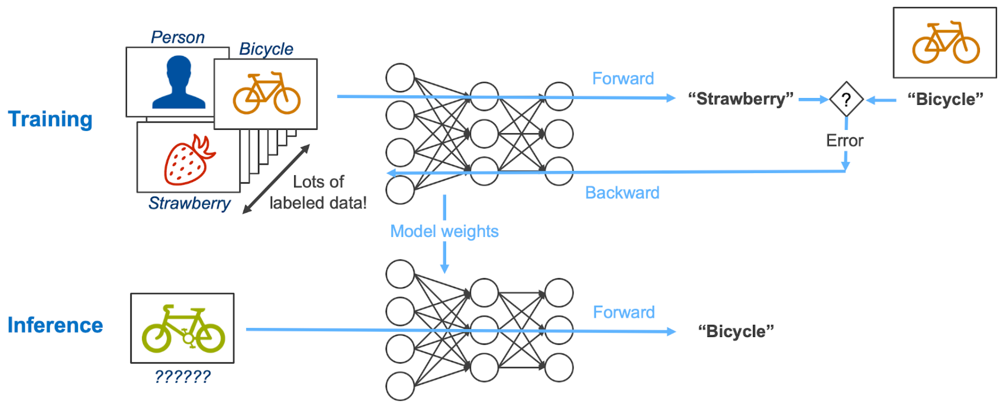


Figure 2.2. *The basic difference between training and inference with deep learning model is that in training labeled data is fed to DL model and the model tries to minimize the prediction error. In inference trained model is used to classify unknown data. [6]*

The NN principle is described in Figure 2.3. As an input, each node takes a weighted sum of output values of previous nodes plus bias. The sum goes through a non-linear activation function which creates the output of the node. While training the neural network learning parameters are weights and bias. Bias is simply a value that is added to the sum. It can improve the performance in some cases. The activation function must have a non-linear operation. Without a non-linear component, the neural network could be presented with just one layer which would not work in complex applications.

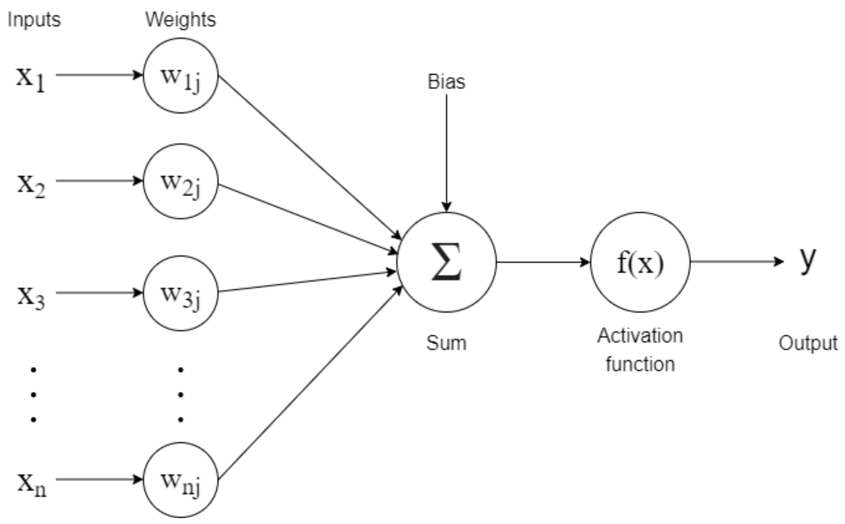


Figure 2.3. *One Neural network node*

There are several types of activation functions. They are typically simple as presented in Figure 2.4. Both Sigmoid and Tanh functions were popular in the past because of monotonicity, continuity and bounded properties. But as the computational power of the computers increased and neural networks became deeper, Sigmoid and Tanh had a problem called "vanishing gradient" while backpropagating the gradient. This was due to saturation as there could not be much deviation with very high values. The ReLu function solves this problem by having a bigger gradient than Sigmoid or Tanh. However, its disadvantage is that because all negative values become zero. It can lead to a dead neurons problem. This means that if the input value is less than zero, it cannot be updated ever again. This can be overcome by choosing smaller learning rates so that there would not be a big gradient and thus big negative weight in the ReLu node. Another option is to use ReLu variants such as leaky ReLu. [7, 8, 9]

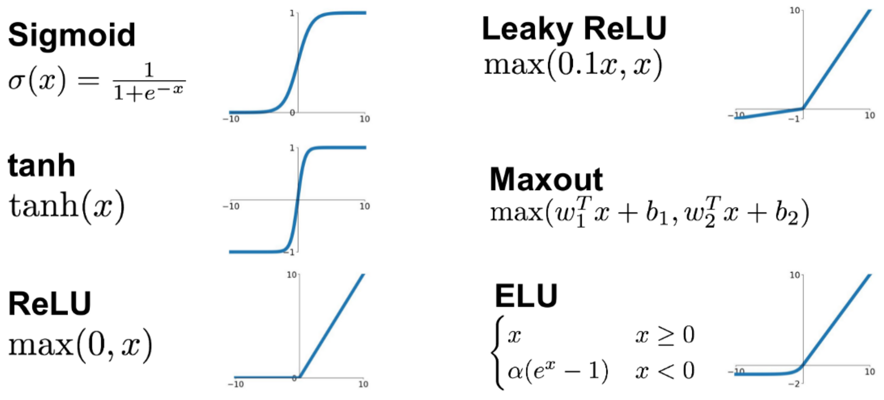


Figure 2.4. *Popular activation functions [10]*

Convolutional Neural Networks (CNN) are a subclass of Artificial Neural Networks. Figure 2.5 presents one kind of CNN called VGG (Visual Geometry Group) neural network by Simonyan and Zisserman from Visual Geometry Group of the University of Oxford [11].

In CNNs the main component is convolutional layers in which there are different kinds of filters also known as kernels. For example the first filter can be a 3x3x3 filter that convolves over the RGB image. The next layer input is a set of channels made by filters of the previous node.

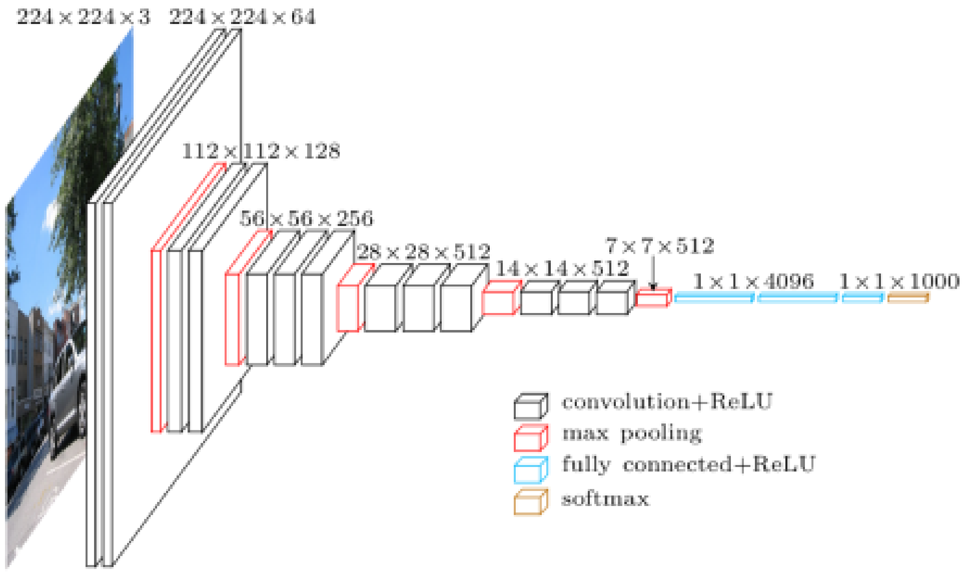


Figure 2.5. *VGG convolutional neural network [12]*

In VGG the input image is a 224x224x3 RGB image. The first hidden layer is a set of 3x3 filters. More precisely, there are 64 filters which each produces one channel for the next hidden layer. Another important component of VGG, as well as all CNN, is max-pooling. A max-pooling layer simply drops some of the data. This prevents overfitting and reduces the number of learning parameters. Overfitting is a phenomenon where machine learning algorithm learns the training data too well and it doesn't generalize to test data.

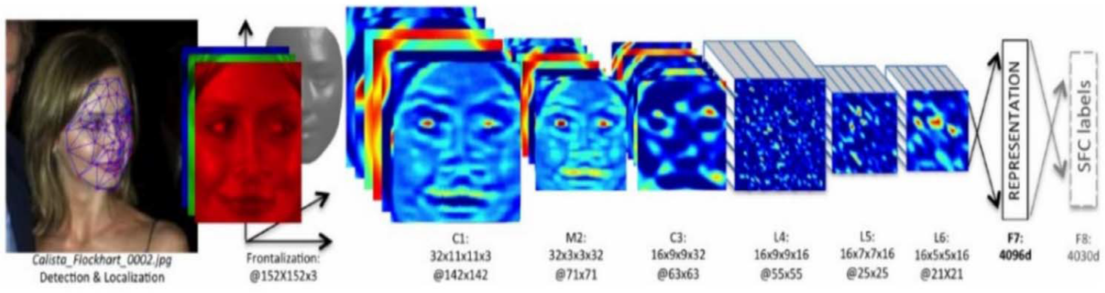


Figure 2.6. *Finding face features in the 2012 ImageNet challenge [13]*

Figure 2.6 illustrates how CNN finds important features for object detection. By progressing in layers, it increases the level of abstraction of the images. In the first few layers the face can be identified. However, in the further layers, it becomes too abstract for humans to recognize the content of the image.

2.1 Hardware used for deep learning

Deep Learning applications tend to be computationally expensive. For this reason, the hardware plays a crucial role in these kinds of tasks. In general, there are few ways to do embedded deep learning inference: cloud, edge and end-device (Figure 2.7).

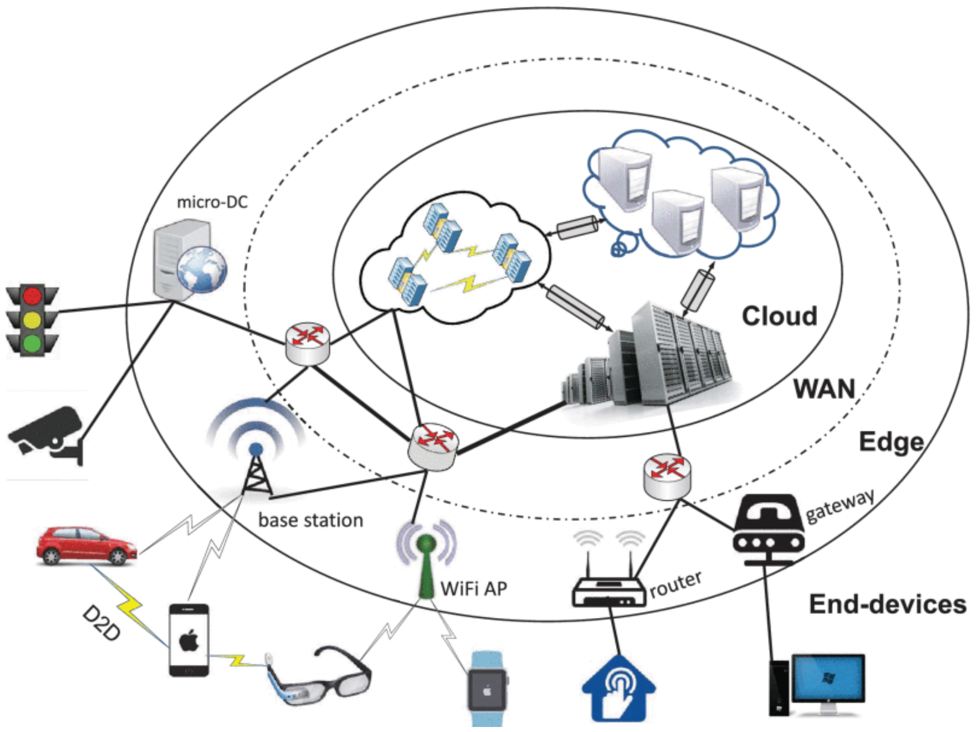


Figure 2.7. *Illustration of cloud, edge and device computing [14]*

Zhi Zhou et al. [14] used levels (0-6) to illustrate the usage of cloud, edge and device computing and their combinations in training and inference.

1. *Level 0: Cloud Intelligence* Both training and inference are done in the could
2. *Level 1: Cloud–Edge Coinference and Cloud Training* Training takes place in the cloud. Cloud–Edge coinference means that the data is partially offloaded to the cloud.
3. *Level 2: In-Edge Coinference and Cloud Training*
4. *Level 3: On-Device Inference and Cloud Training*
5. *Level 4: Cloud–Edge Cotraining and Inference*
6. *Level 5: All In-Edge*
7. *Level 6: All On-Device*

Next, the advantages and disadvantages of cloud and end-device computing are discussed. Then, the different hardware solutions and architectures are reviewed.

2.1.1 Cloud computing vs End-device computing

As described in Chapter 1, cloud computing is used to overcome the computational limitations of embedded devices. However, there are several factors to be considered so that the system can meet the requirements. Table 2.1 compares cloud computing to end-device (embedded device) computing.

Table 2.1. *Comparison of end-device and cloud computing. Modified from [15]*

	End-device Computing	Cloud Computing
Energy	Low	High
Latency	Minimum	Higher
Performance	Low	Much higher
Reliability	High	Lower
Privacy	High	Risk

These variables are not unambiguous and they depend on many things. First, **energy** consumption can be considered for the whole system (cloud and end-device) or just for the end-device. There must be some kind of end-device in both cases. A central challenge is the trade-off between the energy consumption of computation and transmission. The more energy is used for transmission, the less energy is required for computation and vice versa. Wireless communication modules tend to consume a lot of energy. Also, the signal strength and bandwidth of the network influence the transmission energy. [15] If the whole system is considered then only using the end-device will potentially save energy because energy is not used in the transmission and maintenance of the cloud.

Latency may be the most important variable in safety-critical applications such as health emergencies and autonomous driving. In these applications, even a slight delay may cause severe consequences. In other cases, it may at least weaken the user experience. Because only the end-device is used, it reduces the latency to the minimum because no time is wasted in data transmission. However, if the amount of transmitted data is small enough and the operations done are heavy then the overall time may be reduced by using cloud computing. The cloud computation **performance** is always higher because there is more flexibility in terms of e.g. space, energy consumption and cooling. [15] In the near future new 5G technology may help with latency because it can reduce theoretically the latency to only 1 ms [16] compared to 4G's 50 ms latency.

In addition to latency, network **reliability** is the most important advantage of end-device computing over cloud computing. If everything is done inside the end-device there is one less reliability component that needs to be aware. Also, if the system, such as a car, is moving, there can be low coverage areas where the continuous and fast internet connection cannot be provided.

Lastly, sending personal information to the cloud can pose a **privacy** risk. This is of course always an important concern but even more so with sensitive information such as personal health information. If the end-device is not connected to the internet, the risk of information leak is minimal. However, in some cases it would be convenient that the information could be shared with medical professionals. This is a common trade-off between convenience and privacy. [15]

2.1.2 Hardware acceleration

Embedded devices are combinations of software and hardware. These systems are usually designed for specific tasks such as controlling a robot vacuum cleaner or monitoring a driver's awareness. Typically, in embedded systems the software is running on a Central Processing Unit (CPU) which is a circuit that governs specific operations. CPUs are usually too inefficient for running DL models because they require parallel computing and CPUs are designed for serial processing [17]. This is why DL inference on embedded devices needs a hardware accelerator. There are a few different technical architectures for DL acceleration which are presented next. A common factor for all these architectures is that they are designed for parallel computing.

Graphics Processing Unit (GPU)s are commonly used for accelerating 3D graphics applications in computers. That was the only task for GPUs about two decades ago. At the beginning of the 21st century, computer scientists found out that GPUs could be used in other tasks as well. As GPUs became more programmable, they afforded more new possibilities than graphics rendering. [18] In DL, GPUs tend to have decent performance and great compatibility but consume more energy than FPGAs or ASICs [19].

Field Programmable Gate Array (FPGA) is a semiconductor integrated circuit that has programmable interconnects. The main difference between FPGA and **Application Specific Integrated Circuit (ASIC)** is that once manufactured ASICs cannot be reprogrammed [20]. Even though FPGAs and ASICs are energy-saving and require less computational resources, they are only popular in the research field because of the programming complexity and lower compatibility compared to GPUs. [19, 21]

Tensor Processing Unit (TPU) is a special type of ASIC by Google, which is specifically designed for deep learning tasks. The TPU combines a vector processing unit (VPU) with a dedicated matrix multiply unit (MXU). The MXU is responsible for matrix multiplications where as the VPU does all other tasks such as activation and softmax functions. This way, the TPU can be very powerful in large matrix multiplications tasks such as CNN or NN. [22]

Utilizing deep learning on embedded devices in different industries will be discussed in the next chapters. First, automotive and then medical industry applications are reviewed.

2.2 Deep learning in automotive industry

2.2.1 Driver monitoring system

The European Commission agreed in 2019 that every new car must be equipped with around 30 advanced safety features by 2022. This is a part of the European Union's (EU) long-term goal called, "Vision Zero". The commission states that these safety systems will prevent at least 140 000 serious injuries by 2038. [23] Some of those new safety features such as drowsiness and distraction monitoring, advanced emergency braking and lane keeping assistance require AI and most likely DL.

Drowsiness and distraction monitoring are key components of the Driver Monitoring System (DMS). In 2019, there were 3142 deaths caused by distracted driving and 697 deaths caused by drowsy driving in the USA [24]. Almost 4000 people lost their lives because the drivers did not pay 100 % attention to driving. Texting and driving is the most alarming distraction according to the US National Highway Traffic Safety Administration (NHTSA). Most of these deaths could have been prevented by installing a DMS.

Several methods for drowsiness and distraction monitoring have been proposed. Sang-Joong Jung et al. [25] used an electrocardiogram (ECG) sensor and analyzed the heart rate variability (HRV) to determine driver's fatigue and drowsiness status. Also, a driver input based system is proposed, where both steering wheel and throttle inputs are monitored [26]. However, the most studied and versatile methods are Computer Vision (CV) based systems [27, 28, 29]. With a CV-based system, drowsiness and distraction monitoring can be done simultaneously. Reddy Bhargava et al. [30] developed a DMS that featured eyes, mouth and face bounding boxes which were fed into drowsiness detection network. This was computationally too expensive, the model was compressed. In the end, the system achieved over 90 % accuracy and a 14,9 FPS performance with NVIDIA Jetson TK1.

Since DMS are safety-critical components relying on only one system can be risky. To increase accuracy and reliability, a combination of different systems may be the best way to design DMS. In their study, Boyraz et al. [26] combined driver input measurement with a CV system. Instead of using images for the decision-making, they measured five different features: eye closure, gaze vector (x, y), and head motion (x, y), which reduces the amount of data and thus computational load. They then combined the visually collected data with driver input data: steering wheel angle, vehicle speed, and force applied to the steering wheel. They reached 89 % accuracy for determining a drowsiness level from 1 to 5.

2.2.2 Self-driving cars

Self-driving cars are the future of mobility and are having a huge impact on the automotive industry. Autonomous vehicles utilize a wide range of technologies such as cameras, LIDAR and ultra sound. Information provided by these technologies allows the car to make safe decisions in traffic. [31] In this chapter different levels and requirements of self-driving cars are discussed.

On the one hand, self-driving can mean task assistance where the human delegates one specific action to a designated system such as cruise control. On the other hand, self-driving can mean a fully autonomous car that navigates the roads without human interference. The Society of Automotive Engineers (SAE) defines six levels of car autonomy from 0 to 5 as follows [32].

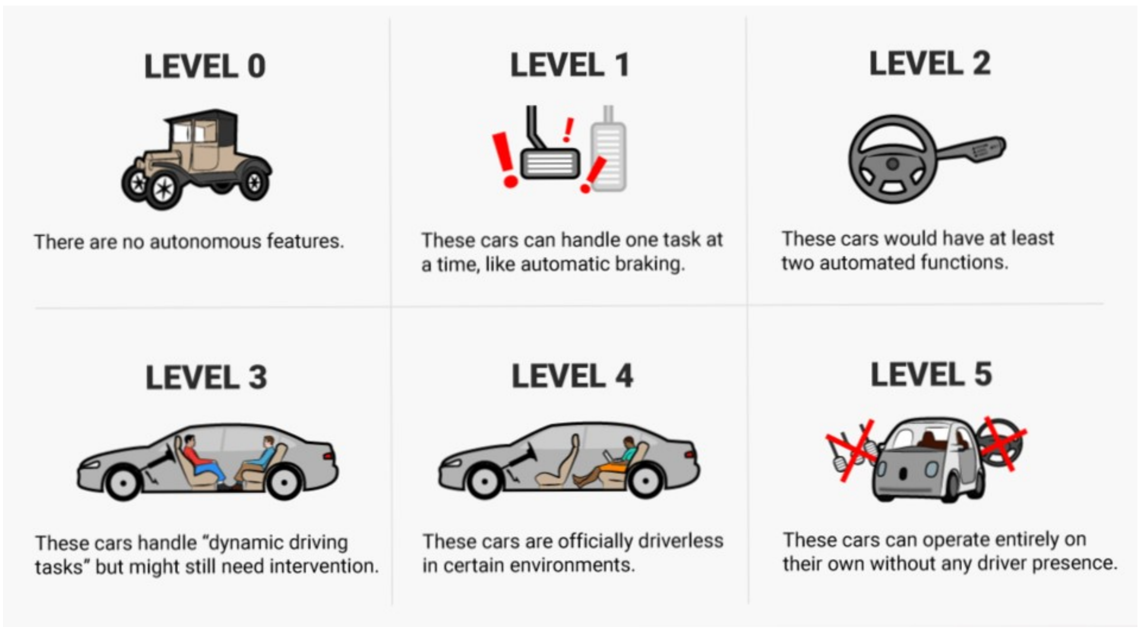

LEVEL 0
There are no autonomous features.
LEVEL 1
These cars can handle one task at a time, like automatic braking.
LEVEL 2
These cars would have at least two automated functions.
LEVEL 3
These cars handle "dynamic driving tasks" but might still need intervention.
LEVEL 4
These cars are officially driverless in certain environments.
LEVEL 5
These cars can operate entirely on their own without any driver presence.

Figure 2.8. *Visualized difference between car automation levels [33]*

Level 0: No Automation: A SAE Level 0 car has no automation. All the driving tasks are done by the driver at all times. The majority of used cars on the market are classified as SAE Level 0 [34].

Level 1: Driver Assistance: A SAE Level 1 car is controlled by the driver but some driving assistant features such as Adaptive Cruise Control (ACC) or lane centering may be included. Level 1 cars can have only one of these kinds of driving assistant features. ACC adjusts the vehicle speed to the car driving in front of it. The distance to the car in front can be measured by radar, laser or camera. While radar or laser systems do not require AI camera systems may require it.

Level 2: Partial Automation: If both lane centering and ACC are combined, the vehicle is classified as a SAE Level 2. Tesla's Model S is an example of this. [34] A Level 2 car requires the driver's full attention and the hands have to be kept on the wheel at all times.

Level 3: Conditional Automation: At SAE Level 3, the driver's full attention is no longer required. Nevertheless, the driver has to be ready to take over the driving task at a moment's notice. There are no Level 3 cars yet on the markets. Audi planned on releasing its new A8 with Traffic Jam Pilot feature. This feature could have been applied with speeds under 60 km/h. Due to legal concerns, Audi decided to postpone the roll-out of this feature [35]. BMW and Honda are also very close to releasing their new SAE Level 3 automation cars [36, 37].

From this level onward cars need to have very powerful computers to be able to run complex DL models. These models identify objects and create a 3D model of the car's surroundings. [31]

Level 4: High Automation: A SAE Level 4 car is fully automated under some circumstances. At this level the driver won't be required to take over at a moment's notice but must be ready to take over control if the car's sensors cannot provide reliable information. [31] For example raining and snowing is a challenge for LIDAR systems.

Level 5: Full Automation: At SAE Level 5, a car is fully automated. It doesn't even need a steering wheel because it won't require any human interactions at any point. Tesla's co-founder and CEO Elon Musk claimed in World Artificial Intelligence Conference "I remain confident that [Tesla] will have the basic functionality for Level 5 autonomy complete this year [2020]." [38]. This ambitious statement has been questioned for instance by Professor Melanie Mitchell who doubts that it is not possible to train DL algorithms to handle all possible situations [38].

Getting a self-driving car flawless is an arduous challenge, even impossible. When we discuss about autonomous vehicles, we need to keep in mind that people also make mistakes. Therefore, we need to determine an acceptable level of confidence for these cars. It may not be acceptable for a car to be only as good as a human, but whether it must be double or a hundred times better than a human, that is the big question.

2.3 Deep learning in Medical industry

For Deep Learning, there are huge opportunities in the medical industry. Image processing is one of the greatest areas where DL can help a specialist with simple tasks. This way, the expensive specialist can concentrate on actual patient care. For example, DL has given very promising results in fracture detection [39]. A study conducted by Accenture [40] estimated that AI applications could potentially generate $150 billion in annual savings for the U.S. health care system by 2026. Teikari et al. [41] reviewed embedded deep learning in ophthalmology (eye related medicine). They stated that embedded DL can be used for automated image acquisition. This would result in a high-quality image or recording without a specialized operator. There could be several delivery options for this. Firstly, patients could use shared devices in local supermarkets. Secondly, patients could be imaged before an ophthalmologist appointment in a hospital waiting room. Thirdly, patients who live in remote areas could be imaged with mobile imaging equipment. And lastly patients could do home monitoring for disease progression. [41]

Malaria is a tropical disease that occurs mostly in Africa but also in Asia and South America. Especially in Africa, health care is not as advanced as in the Western world which restricts diagnostics and access to treatment. At least the diagnostics part could be improved with edge AI. Feng Yang et al. researched [3] to research how malaria could be detected with a smartphone-based system. The Android phone was attached to the eyepiece of the microscope and the taken pictures were analyzed to find and count the number of parasites. This was implemented using the OpenCV4Android SDK library. In the end they reached 93,46 % accuracy. Malaria screening is only one example of this enormous potential of edge AI in rural areas.

Even though DL/ML based systems have been proven to be better than humans in many cases there is a concern of using these systems in the medical industry due to its unpredictability. In the paper "What do we need to build explainable AI systems for the medical domain?" Andreas Holzinger et al. [42] discuss how and why AI systems are needed to be explainable. They pointed out that medical professionals have to understand why and how given decisions have been made. In their research paper "Deep neural networks are easily fooled", Nguyen et al. [43] found out that DNN models can classify completely unrecognizable images for humans as near-perfect examples of random class. A similar study [44] by Su et al. stated that even one pixel change can result in misclassification. In addition, the most accurate methods such as DL are the least transparent and the most transparent methods such as decision trees are usually less accurate [45]. The question is how device manufacturers can ensure patient safety of the patient in the medical domain. This problem is not only relevant to the medical industry but automotive and other industries as well.

The challenge for separates a medical image classification is that image annotation for model training tends to be even more expensive than in other industries. Annotations, such as detecting bone fracture, must be done by a medical specialist of a certain field which is more expensive than for example annotating traffic signs.

3 DEEP LEARNING SOFTWARE DEVELOPMENT ON EMBEDDED DEVICE

Deep learning models are not usually trained on embedded devices. Typically, the deep learning model's computationally expensive training is done on high-powered machines. In this process, machine learning frameworks are used. The next step is to optimize the trained model and deploy it to the embedded device. This optimization is done using frameworks such as TensorFlow Lite and TensorRT.

This chapter will cover the basic steps of DL software development on embedded devices and some useful frameworks for that purpose.

3.1 The Deep Learning development workflow

This is the workflow for DL model and algorithm development for embedded devices:

1. Decide on a goal
2. Collect a dataset
3. Design a model architecture
4. Train the model
5. Convert the model
6. Run inference
7. Evaluate and troubleshoot [46]

Decide on a goal

Before designing any algorithms, the goal has to be defined. This step is important for the developer to decide which data to collect and which model architecture to use. For example in driver monitoring the goal can be "detect if driver's eyes are closed". This is typical classification problem where the output is a probability of different classes. In this case "open or "closed". However, if the goal is to detect where the driver is looking, then the output is a vector that is illustrated by direction. [46]

Collect a dataset

In many cases, this part can be the most time-consuming. It is hard to estimate how much data is required to train an effective model. The complexity of the model and the relationship between variables, the number of parameters and the amount of noise affects to the data requirements. In DL applications the principle is: the more data, the better. [46]

However, what data to collect needs to be considered. It is essential to collect data from every possible scenario to increase robustness. For the example of driver monitoring, data should contain different lighting conditions, various skin colors and people of different ages. [46]

After collecting data, interesting data points need to be determined. Labeling data can take hundreds of hours and for example in medical applications it can be costly if the labelling needs to be done by specialized doctor. For driver monitoring purposes, labels such as "smoking" or "looking at the phone" can be assigned. [46]

Design a model architecture

There are infinite possibilities for model architectures and it requires experience and knowledge to judge which architecture works best for a given problem. Sometimes it can be more convenient to base the model on an existing architecture developed by researchers rather than starting from scratch. When designing model for embedded devices, constraints of the device need to be considered. Models that run on a desktop may not run on smaller devices due to memory and speed restrictions. Additionally, some embedded devices may have hardware acceleration which can speed up the execution of certain types of model architectures. To achieve best results, the model architecture should be designed for the target device. It needs to be kept in mind that designing a model architecture is an iterative process. Developers often go back and forth within the ML workflow to find a working model. It is almost impossible to know which model works and which does not without experimenting. Experience can be helpful. [46] There are several frameworks for creating model architectures of which the most popular are discussed in Chapter 3.2.1.

Train the model

Model training is a process where training data is fed to the model to adjust its learning parameters. At first, the model's weights and biases are set to random values. Weight adjusting is done by comparing the model's training data output with the desired output by using a backpropagation algorithm. This algorithm finds the global minimum by the method of steepest descent [46, 47]. After training the full dataset for one cycle, one epoch is done. Usually, these epochs are done several times until the model's performance stops improving. [46]

There are two common performance metrics to monitor model's training, accuracy and loss. Accuracy is the percentage of how often the model guesses the expected answer and loss means how far the model was from the expected answer. Accuracy should approach 100 % and loss 0,0 during training. In practice models are rarely perfect (accuracy 100 % and loss 0,0) and it depends on the application and how accurate it needs to be. It is not guaranteed that the model reaches acceptable accuracy with any number of epochs. Then the developer has to go back to designing the model architecture and try again. It is also possible that the problem was too challenging and any ML algorithm cannot solve it. [46]

Convert model

The trained model can be used on a desktop. However, if the model is intended to be used on an embedded device, it has to be converted to the format that the device's interpreter supports. For example, TensorFlow (TF) has made interpreters for mobile devices called TensorFlow lite. It also provides converters which makes it easy to convert TF models to tflite-format. Covering tools are discussed in Section 3.2.3.

Run inference

A trained and converted model is not enough in itself. It needs additional code to support data preprocessing and instructions on what to do for predicted results. It is typical for classifiers that the output gives a probability for all classes and these probabilities will sum to 1. The class with the highest probability is the prediction. In real-life cases, interference such as the sensor malfunction may cause false predictions. Thus, it may be useful to filter the predictions. [46] For the DMS example, it may be hard to judge when to alarm for drowsiness. If the alarm turns on as soon as the diver closes his eyes, it will be irritating quickly. However, if the delay is too long, an accident could already have happened by the time the alarm turns on. The same is the case when using the phone while driving: if the system alarms as soon as there is one phone prediction, real or false, it is not working correctly. There should be some averaging or another filtering instance.

Evaluate and troubleshoot

When the device is up and running it should be evaluated and tested if it works as it is supposed to work. Even though the model's accuracy was acceptable in the training phase, the real-world performance might be poor. Perhaps the training data did not represent real cases or it is not processed the same way. Another reason is that the model was overfitted. “Overfitting” means that during the training phase the model learned to predict training data too well. A model's accuracy for training data can be very high but its robustness for any other data can be poor. In practice, the model memorized the training data and developed shortcuts for it. For example, if a classifier should detect if the driver is sleeping or not and in training set all "sleepers" are bearded white men and "not sleepers" are long-haired black women, a model can find some false features for detecting drowsiness such as skin color. Another problem arose in spring 2020 when people started to wear face masks. A face covered with a mask could be a great challenge for models that were not trained for it. Overfitting is very common and it can be avoided e.g. by collecting more training data and ensuring that the data has enough variation. [46]

If the system performance is still not good enough after checking all hardware and other troubleshooting options, others should be considered such as designing the model architecture again. In total, well-performing systems will usually need multiple iterations to reach maximum performance.

3.2 Software stack for DL on embedded device

In this chapter, DL on embedded devices is presented as a stack of different SW tools. The underlying idea is that SW developers can pick and combine different frameworks for the DL pipeline. Figure 3.1 presents some of these components.

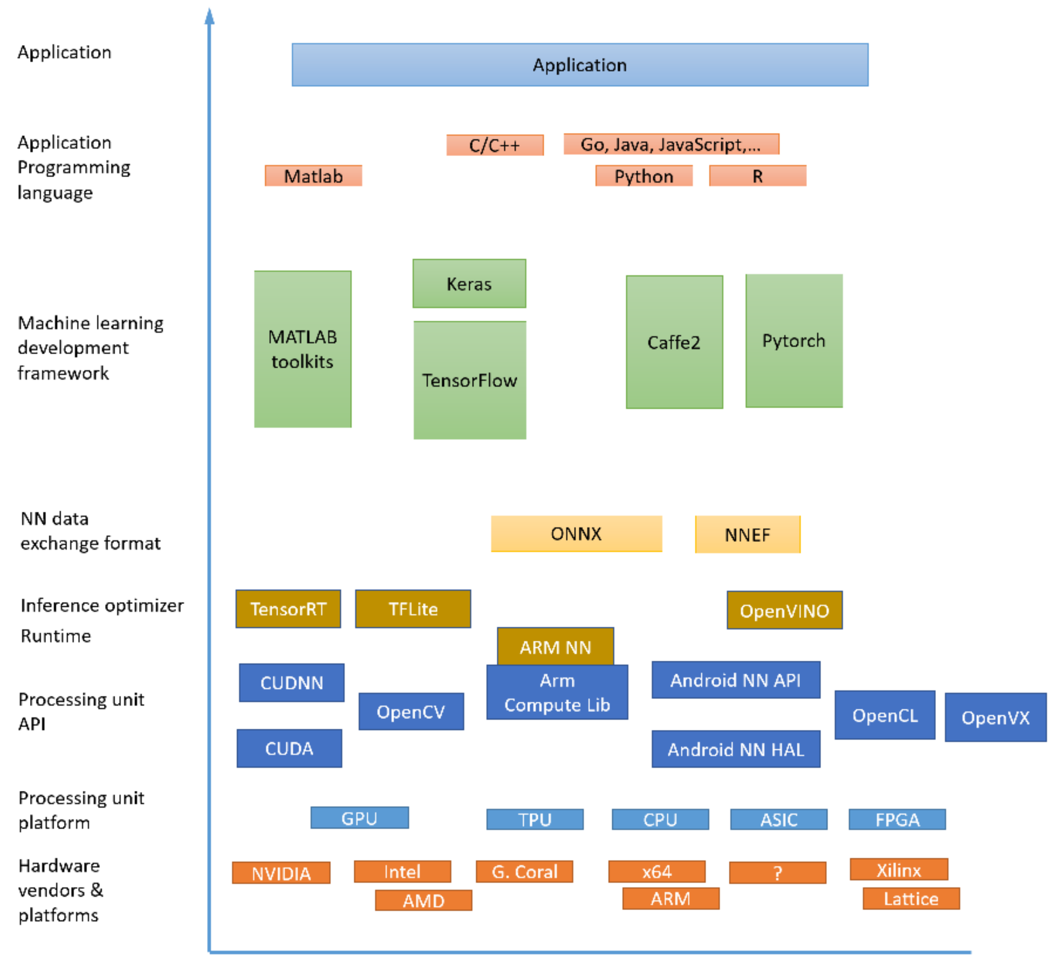


Figure 3.1. *Illustration of tools used in ML and DL development and computing.*

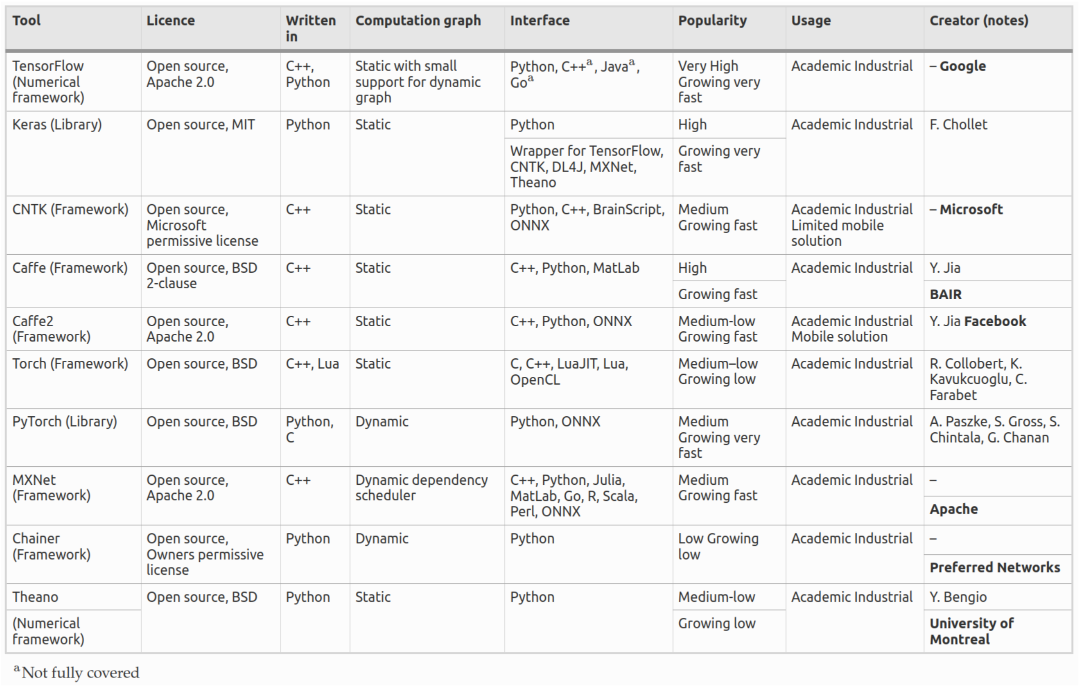
In Figure 3.1 the first layer is application. It can be practically any ML/DL application. However, this work focuses on camera-based applications. The second layer, application programming language determines which programming language is used in any given application. The third layer represents ML training frameworks which will be described in Chapter 3.2.1. The fourth layer determines the NN data exchange format. These tools enable the developer to change the trained model to another format. For example, if a PyTorch developed model is intended to be used with TensorFlow Lite, it must be converted to the TensorFlow format. Two popular NN exchange format tools are introduced in Section 3.2.2. The next layer is the processing unit API which refers to the DL acceleration SW. These are discussed in Section 3.2.4. The last two layers refer to the hardware. The processing unit platform differentiates different types of processing units and the last layer further distinguishes these processing units by architecture and vendor.

In the following chapters, different frameworks for training and deployment are covered. First, ML frameworks are introduced and their advantages and disadvantages are discussed. Second, model deploying frameworks and optimization methods are reviewed.

3.2.1 Deep Learning training frameworks

Frameworks are vital to start DL model development and over the last years, many frameworks have been developed. Therefore, many frameworks have been developed over the years. A unifying factor between all of the DL frameworks is to facilitate the complicated data analysis process. The designed purpose of these tools varies and one framework cannot address every problem. Figure 3.2 shows a summary of GPU-supported frameworks. All the following tools utilize cuDNN and CUDA (introduced in Chapter 3.2.4) for GPU processing.

This chapter reviews software frameworks for DL. Due to the vast amount of frameworks available today, only the most important frameworks are discussed in this chapter with a focus on image and video processing.



Tool	Licence	Written in	Computation graph	Interface	Popularity	Usage	Creator (notes)
TensorFlow (Numerical framework)	Open source, Apache 2.0	C++, Python	Static with small support for dynamic graph	Python, C++[a], Java[a], Go[a]	Very High Growing very fast	Academic Industrial	– **Google**
Keras (Library)	Open source, MIT	Python	Static	Python	High	Academic Industrial	F. Chollet
				Wrapper for TensorFlow, CNTK, DL4J, MXNet, Theano	Growing very fast		
CNTK (Framework)	Open source, Microsoft permissive license	C++	Static	Python, C++, BrainScript, ONNX	Medium Growing fast	Academic Industrial Limited mobile solution	– **Microsoft**
Caffe (Framework)	Open source, BSD 2-clause	C++	Static	C++, Python, MatLab	High	Academic Industrial	Y. Jia
					Growing fast		**BAIR**
Caffe2 (Framework)	Open source, Apache 2.0	C++	Static	C++, Python, ONNX	Medium-low Growing fast	Academic Industrial Mobile solution	Y. Jia **Facebook**
Torch (Framework)	Open source, BSD	C++, Lua	Static	C, C++, LuaJIT, Lua, OpenCL	Medium–low Growing low	Academic Industrial	R. Collobert, K. Kavukcuoglu, C. Farabet
PyTorch (Library)	Open source, BSD	Python, C	Dynamic	Python, ONNX	Medium Growing very fast	Academic Industrial	A. Paszke, S. Gross, S. Chintala, G. Chanan
MXNet (Framework)	Open source, Apache 2.0	C++	Dynamic dependency scheduler	C++, Python, Julia, MatLab, Go, R, Scala, Perl, ONNX	Medium Growing fast	Academic Industrial	–
							Apache
Chainer (Framework)	Open source, Owners permissive license	Python	Dynamic	Python	Low Growing low	Academic Industrial	–
							Preferred Networks
Theano	Open source, BSD	Python	Static	Python	Medium-low	Academic Industrial	Y. Bengio
(Numerical framework)					Growing low		**University of Montreal**

[a]Not fully covered

Figure 3.2. *Summary of the most used DL frameworks with GPU support [48]*

TensorFlow

TensorFlow (TF) is a open-source software library for ML and DL and is developed and maintained by the Google Brain team. TF includes an API for Python and C++ and is one of the most popular frameworks. In addition, it is supported in Google's and Amazon's cloud services. One downside of TF is the rather steep learning curve due to its diverse functionality and low-level implementation. [48] Even though there are tools to convert models between frameworks, this task can be challenging at times as Rubin demonstrated in his article [49]. If, for example, TensorFlow Lite is the desired output model, then TF or Keras can be good options.

Compared to PyTorch, TF works more smoothly on embedded devices. TF Lite, which will be discussed in Chapter 3.2.3, allows the use of ML models on embedded devices. According to [50], TF is the best option for RPi (CPU-only), Intel Movidius, Google Coral and NVIDIA Jetson Nano.

In addition, TF supports not only CPUs but also GPUs and TPUs. Implementation on NVIDIA's GPUs is limited because TF only uses cuDNN and CUDA. If a suitable GPU is available, TF can automatically switch to use it and the acceleration is automated. However, for that reason memory control is not possible. [51] TF is available on 64-bit Linux, macOS, Windows and mobile platforms such as iOS and Android. Python versions 3.5-3.8 are supported.

TF is commonly compared to PyTorch because they are the most used DL frameworks. In general, TF is more focused on industrial use cases and PyTorch on research [51].

```python
import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D
from tensorflow.keras.layers import MaxPooling2D, Flatten

model = Sequential()
model.add(keras.Input(shape=(640, 360, 3)))
model.add(Conv2D(32, 5, activation="relu", padding="same"))
model.add(Conv2D(16, 3, activation="relu", padding="same"))
model.add(MaxPooling2D(4))
model.add(Flatten())
model.add(Dense(units=10, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Program 3.1. *Keras model definition and training*

Google uses TF in many of its products, e.g. search results and autocompletion, speech-to-text and voice technology, image recognition and classification as well as spam detection for Gmail.

Keras

On top of TensorFlow, there is a high-level API called Keras. The development of Keras was driven by the idea that it would be easy to use and fast prototyping would be possible. [52] This is also Keras' weakness: Keras is less flexible and optimal for researching new architectures. One of the greatest advantages of Keras, as well as TensorFlow, is that they are supported by large companies (TF by Google, Keras by Google and Microsoft). [48]

Keras provides easy-to-use DL tools. Because Keras is a library running on top of TensorFlow, it is easy to convert Keras models to TensorFlow Lite and deploy it to embedded devices.

Program 3.1 demonstrates how easy it is to create a Keras model and train it.

First, all the necessary packages are imported. Then, the model instance is created. “Sequential” is the simplest type of model because it is a linear stack of layers. Two dense layers are added with different parameters and to finalize the learning process, it is configured with the compile function. The training is done with a simple fit function where the training parameters, such as the number of epochs and batch size, are determined. Parameter x_train is a list of training data (e.g. pictures) and y_train a is list of the correct labels for the training data. The model.summary() prints model shape as shown below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 640, 360, 32)      2432
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 640, 360, 16)      4624
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 160, 90, 16)       0
_________________________________________________________________
flatten (Flatten)            (None, 230400)            0
_________________________________________________________________
dense (Dense)                (None, 10)                2304010
=================================================================
Total params: 2,311,066
Trainable params: 2,311,066
Non-trainable params: 0
```

In the next section, a similar python script is used to show a PyTorch comparison.

PyTorch

PyTorch is, next to TensorFlow, one of the most used DL frameworks and was developed by Facebook's AI research group. Horace He claimed in his article from 2019 that TensorFlow and PyTorch were the best choices for DL at this point. [53].

Similar to TF, PyTorch also uses cuDNN and CUDA for an optimized GPU usage. However, PyTorch released version 1.8 on 4.3.2021 [54] which also supports the AMD ROCm. This allows PyTorch to use AMD's GPUs in addition to NVIDIA's. Not all AMD GPUs support ROCm, however.

Even though TF is generally better for embedded device development, mobile phones are an exception. Following version 1.3, PyTorch has supported an end-to-end workflow deployment on iOS and Android mobile devices. For mobile GPU usage, PyTorch uses Metal with iOS. Metal is an API developed by Apple which enables near-direct access to the GPU. For Android phones, PyTorch uses Vulkan with Android phones. [55] Vulkan is a Khronos-developed API for the same purpose as Metal. It has similar unmodified functionalities from OpenGL and OpenCL [56] which are discussed more in Chapter 3.2.4.

PyTorch is widely used in commercial products such as Facebook (e.g. facial recognition,object detection, spam filtering,fake news detection, as well as news feed automation

```
import torch
from torch import nn
import torch.nn.functional as F
import torch.optim as optim

# Define model
class Net(nn.Module):
        def __init__(self):
                super().__init__()
                self.conv1 = nn.Conv2d(3, 32, 5)
                self.conv2 = nn.Conv2d(32, 16, 3)
                self.pool = nn.MaxPool2d(4)
                self.flat = nn.Flatten()
                self.fc = nn.Linear(160*90*16, 10)

        def forward(self, x):
                x = self.pool(F.relu(self.conv1(x)))
                x = self.pool(F.relu(self.conv2(x)))
                x = F.softmax(self.fc(x))
                return x

net = Net()
print(net) # Dispalys the model's architecture

def train_model()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    for epoch in range(2):  # loop over the dataset multiple times
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

Program 3.2. *PyTorch model definition and training. Modified from PyTorch website [57]* and friend suggestions) [51] and the Tesla Autopilot [58].

PyTorch is simple to use and debug and it is very "pythonic". Unlike for example in TF, developers can place a python debugging breakpoint in the middle of the PyTorch model. In comparison, TF debugging is more complicated. The PyTorch API is great because it is better designed and TF has been switching APIs many times over the last years. The following is an example of the PyTorch model definition and training [53] There is example model definition and training [57] similar to the one made in Keras Section.

At first, the model definition is a bit more complicated with PyTorch than it is with Keras. However, the considerable difference is model training. While there was only one "fit" command in Keras, in PyTorch there is a whole self-made function for it. Whereas in Keras the number of epochs is defined in fit function parameters, in PyTorch it is defined in the for-loop. This is to enable debugging but requires more understanding about DL and the training process which is not necessarily a bad thing.

Caffe

Caffe is a DL framework developed by Yangqing Jia at the Berkeley Artificial Intelligence Research. It is optimized for speed, modularity and expression [59]. To achieve, that Caffe is written in C++. New custom layers must be written in C++ as well. Caffe is highly suitable for image processing and there are a lot of retrained networks available in the Caffe Model Zoo. [48]

In 2017, Caffe 2 was announced and a year later it was merged into PyTorch. Thus, Caffe 2 is part of Facebook's PyTorch ML framework [60].

MATLAB

MATLAB differs from the other DL frameworks in many ways. It is not a DL framework but rather a platform for programming and numeric computing by MathWorks. Whereas almost all other frameworks are open-sourced, MATLAB is a commercial product that is quite expensive for private use. For this reason, it is less popular and has less community support. However, community support is still considered good. The MATLAB language is matrix-based which allows the natural expression of computational mathematics. It is used by engineers and scientists for data analysis, algorithm development, and simulations. MathWorks has two main product families: MATLAB and Simulink. Simulink is a platform for simulations and model-based design and it not relevant considering this work.

MATLAB toolboxes are packages that are not automatically included. There are several useful toolboxes for ML and DL purposes. These packages are reviewed in the book "Practical MATLAB Deep Learning A Project-Based Approach" [61] by Michael Paluszek and Stephanie Thomas. Below, some of the most relevant toolboxes for DL and visual tasks are listed.

- Deep Learning toolbox
- Statistics and Machine Learning Toolbox
- Computer Vision System Toolbox
- Parallel Computing Toolbox

The **Deep Learning Toolbox** is the main toolbox for MATLAB DL. It can be used to design, build, and visualize CNNs. The **Statistics and Machine Learning Toolbox** is made for data analytic methods, gathering trends, and patterns. It provides tools for classification, regression, and clustering which can be used to understand the data and thus for DL model development. The **Computer Vision System Toolbox** includes functions for computer vision systems such as feature detection and extraction. It also supports 3D vision and 3D motion detection as well as stereo camera processing. The **Parallel Computing Toolbox** is not directly a DL/ML toolbox but it is very useful for accelerating computing. It enables multicore CPU processing and GPU processing. Some Deep Learning toolbox functions can utilize parallel computing. In addition, high-level programming constructs such as parallel for-loops are easy to implement. There are also MATLAB open source tools by MATLAB users such as the Deep Learn Toolbox by Rasmus Berg Palm [62], the Deep Neural Network by Masayuki Tanaka [63] and the Pattern Recognition and Machine Learning Toolbox (PRMLT) by Mo Chen [64] among others. [61]

The reason why MATLAB is considered in this thesis is its ability to deploy applications for embedded boards. With the **MATLAB Coder** toolbox C and C++ code can be generated for both PC and embedded HW. The use of the **Embedded Coder** with MATLAB Coder enables C and C++ code generation and optimization for embedded processors. Embedded Coder supports a wide variety of processors. To deploy a standalone application, **MATLAB Compiler** can be used. Software component generation for integration with other programming languages can be done with the **MATLAB Compiler SDK**. [65, 66]

Table 3.1 presents the summary of introduced frameworks.

Table 3.1. *Summary of the most used DL frameworks*

Framework	Creator	Initial release	Open Source	Interface	CUDA support	Easy-to-use	Flexi-bility
Tensorflow	Google Brain	2015	Yes	Python, R, Java, C/C++ JavaScript	Yes	+	+++
Keras	François Chollet	2015	Yes	Python, R	Yes	+++	+
PyTorch	Facebook	2016	Yes	Python, Julis, C++	Yes	+++	+
Caffe	Berkeley Vision	2013	Yes	Python, MATLAB, C++	Yes	++	++
MATLAB	MathWorks	2013	No	MATLAB	Yes	++	+

3.2.2 File exchange

ONNX Ecosystem

Open Neural Network Exchange (ONNX) is a format to represent ML and NN models originally developed by Facebook and Microsoft. The idea behind ONNX is to enable developers to work with different frameworks by allowing them to convert models to another framework as shown on the left side of Figure 3.3. Later ONNX was open-sourced on GitHub and others have joined the development. The advantage is that the developer is not tied to the framework selected at the beginning of the project. ONNX supports all major ML frameworks such as PyTorch, TF, Keras, scikit learn and MATLAB. [67, 68]

```python
import torch
pre_trained = VAE(encoder, decoder)
pre_trained.load_state_dict(torch.load('trained.pt'))
pre_trained.eval()
ex_input = torch.zeros(1, 28, 28)
torch.onnx.export(pre_trained, ex_input, "onnx_model.onnx")
```

Program 3.3. *Converting PyTorch model to ONNX [70]*

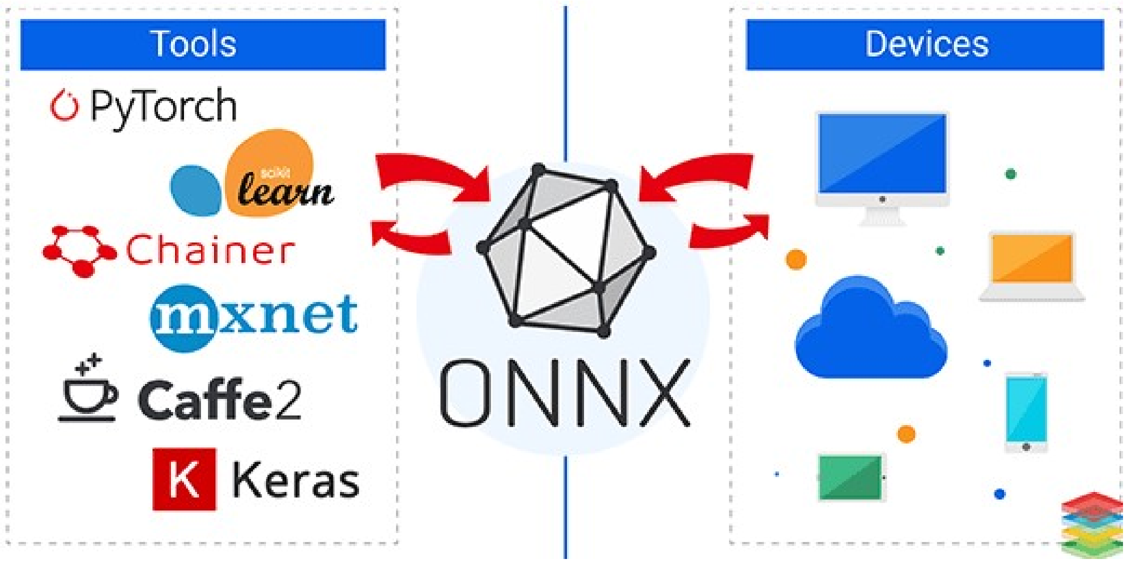


Figure 3.3. *ONNX enable conversions between frameworks and model optimization for various hardware [69].*

Regardless of the ML framework, all of them share most of the mathematical operations which allows almost any model to be easily converted to ONNX and vice versa. Program 3.3 illustrates how to convert a PyTorch model to ONNX. First, the model is instantiated and then weights are loaded. Following that, it's put into evaluation mode, the PyTorch model is converted to ONNX and exported. [68] Since the export runs the model, the example input is required. Any values can be used in the example if the type and size are right. [70]

ONNX Runtime

In companies, there can be several teams using different training frameworks and targeting different deployment options. This causes scattered solutions and compatibility issues. At the end of 2018, Microsoft provided ONNX Runtime as a solution. ONNX Runtime is an inference engine and training accelerator for ML and NN models in ONNX format. [71]

Neural Network Exchange Format

Neural Network Exchange Format (NNEF) is a neural network data exchange format developed by the Khronos Group and is complementary to ONNX. The Khronos Group is a non-profit standardization organization open to universities and companies that can join and participate in the standardization process. The first version of NNEF was launched in 2017. The main goal of the NNEF is to enable data scientists and engineers to use training frameworks of their choice and import trained networks on inference engines from hardware vendors as shown in Figure 3.4. In addition, NNEF can be used for communication between training frameworks. [72]

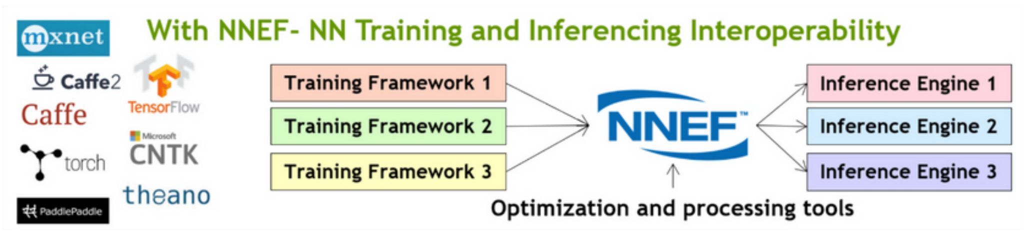


Figure 3.4. *Neural Network Exchange Format supports the most popular training frameworks and is a intermediate step to different inference engines [73].*

The development of mobile and embedded processor architectures with the objective to increase speed and lower power consumption, has been very active in recent years. Because of this development, there is a risk that embedded neural net processing becomes so fragmented that barriers between different solutions make it impossible for developers to accelerate inferencing engines on multiple platforms. NNEF aims to solve this problem.

NNEF and ONNX are complementary but they have some functional and structural differences. Whereas ONNX is used more in training interchange, NNEF is designed for embedded inferencing import. In short, NNEF is more hardware-oriented while ONNX focuses more on software stack flexibility. [72, 73]

3.2.3 Deploying neural network models on embedded device

To be able to use trained DL models on embedded devices, they first need to be optimized before deployment. Model optimization methods, such as combining layers, quantization and elimination of unused layers, reduce the model size and thus memory requirements which allows faster inference. In this chapter, different optimization and inference engines are discussed.

TensorFlow Lite

TensorFlow Lite (TFLite) enables ML inference on mobile and embedded devices. It consists of two components: TFLite converter and TFLite interpreter. [74] The TFLite converter only converts TF-based models [50] (page 431).

The TFLite converter converts TF models into an efficient form by reducing the binary and modelsize. The binary size is decreased by reducing the size of dependencies and the model size by quantizing weights [75]. TFLite interpreter is used to perform an inference with a TFLite model. The interpreter supports inference using C++, Java, and Python.

TensorRT

NVIDIA has released its framework, TensorRT, for neural network model optimization and inference. TensorRT was developed for NVIDIA's GPUs. It can be used complementary to training frameworks such as PyTorch, TF, and Caffe. In addition, it is possible to import existing models from Caffe, ONNX, or TF. [76]

TensorRT allows the creation of optimization profiles. With these profiles, developers can determine configurations such as maximum workspace size, the minimum acceptable level of precision, etc. [76].

OpenVINO

OpenVINO is a toolkit developed by Intel. It includes a Model Optimizer and Inference Engine. The model optimizer supports Caffe, TF, ONNX, MXNet, and Kaldi models. The inference engine supports the execution of Intel's products such as CPU, Integrated Graphics, and Movidius Neural Compute Stick. Intel Movidius can only be used with OpenVINO [50] (page 55).

Arm NN

Arm NN is an inference engine that enables the use of neural networks on Arm Cortex CPUs, Arm Mali GPUs, and the Arm ML processor. Supported NN are TF, TFLite, Caffe, and ONNX. Arm NN optimizes the inference by analyzing the NN and replacing its operations with implementations particularly designed for specific hardware. [77] Arm NN is build on top of ARM Computer Library [78] which is introduced later in Chapter 3.2.4.

3.2.4 Processing unit API

Processing unit APIs are the means to communicate with hardware devices. Figure 3.5 shows the relationship and abstraction level of some APIs. Next, these APIs are discussed.

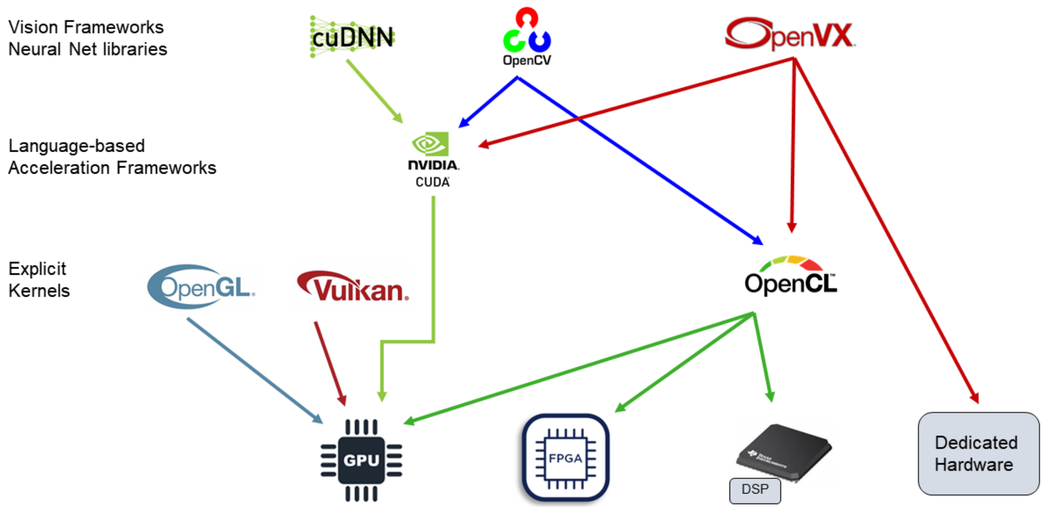


Figure 3.5. *Overview of processing unit APIs. Abstraction level increases towards the top. Modified from [79]*

OpenCL

Application development for heterogeneous parallel processing platforms is challenging because programming approaches for these processors vary a lot. OpenCL (Open Computing Language) addresses this problem. It is a low-level, royalty-free standard for heterogeneous parallel programming from supercomputers to embedded devices by the Khronos Group. It can leverage CPUs, GPUs. and other processors such as FPGA to accelerate parallel computation. OpenCL enables the writing of accelerated portable code across different devices and processor architectures. This low-level abstraction layer relieves developers from studying details of the device. [80, 81]

In addition to computing language, OpenCL includes API, libraries and a runtime system. It supports the writing of a general-purpose program that can be run on a GPU without any additional mapping onto a 3D graphics API if extreme efficiency is not required. [81]

OpenVX

OpenVX is Khronos Group-developed standard for the cross-platform acceleration of computer vision applications. The design goal of OpenVX is to provide optimized computer vision processing for embedded and real-time applications. It is intended for implementation by hardware vendors. Compared to OpenCL, OpenVX has a higher abstraction to enable performance portability across various processing unit architectures. [82]

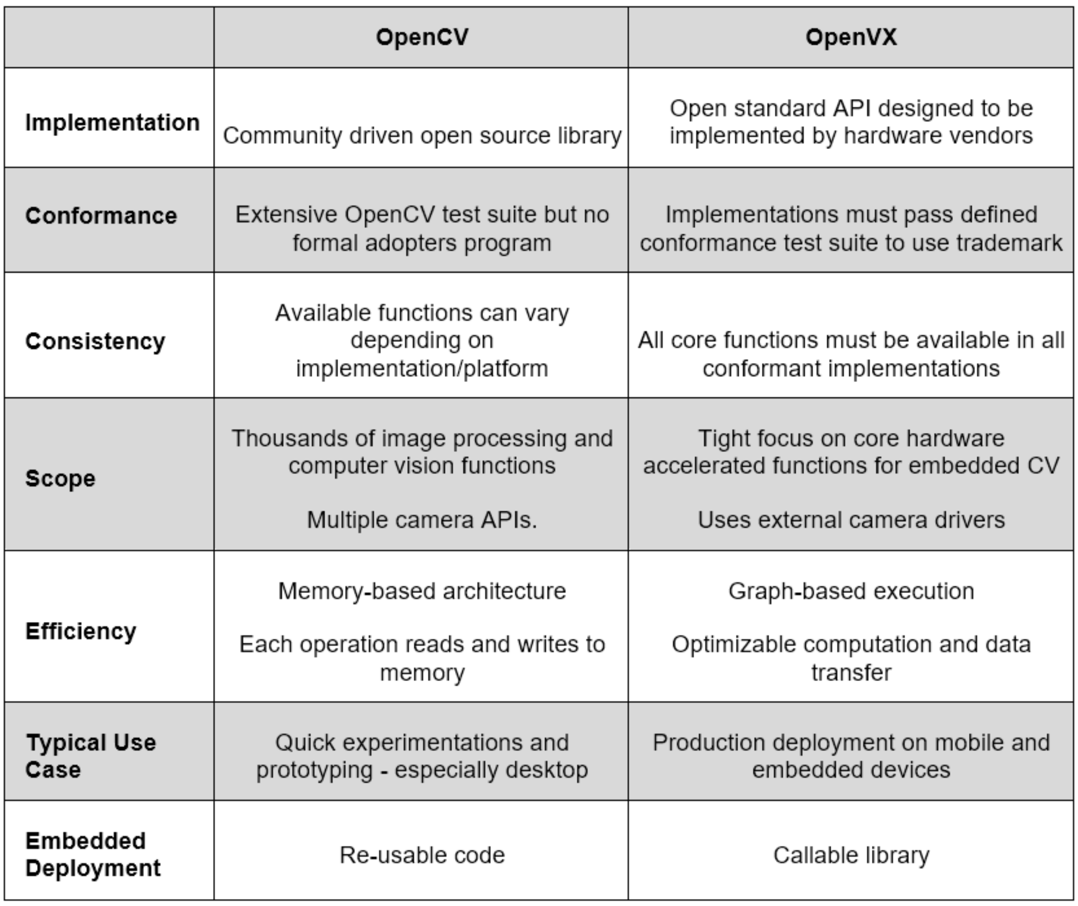
OpenCV

OpenCV (Open Source Computer Vision Library) is a widely used computer vision library.

It has thousands of computer vision functions and it is open-source. OpenCV is written in C/C++ but there are APIs for Python and Java as well.

OpenCV is not a kernel's API but in its OpenCV 3 release, a transparent API was added. This means that the implementation of OpenCV will look for OpenCL on the system. If OpenCL is found, it will begin to dispatch some of the primitives down into OpenCL. The programmer doesn't need to change any of the code. It happens automatically. [79]

OpenCV is complementary to OpenVX. However, they have been designed to perform slightly different tasks. Next, the main differences between these two are discussed (Figure 3.6).



	OpenCV	**OpenVX**
Implementation	Community driven open source library	Open standard API designed to be implemented by hardware vendors
Conformance	Extensive OpenCV test suite but no formal adopters program	Implementations must pass defined conformance test suite to use trademark
Consistency	Available functions can vary depending on implementation/platform	All core functions must be available in all conformant implementations
Scope	Thousands of image processing and computer vision functions Multiple camera APIs.	Tight focus on core hardware accelerated functions for embedded CV Uses external camera drivers
Efficiency	Memory-based architecture Each operation reads and writes to memory	Graph-based execution Optimizable computation and data transfer
Typical Use Case	Quick experimentations and prototyping - especially desktop	Production deployment on mobile and embedded devices
Embedded Deployment	Re-usable code	Callable library

Figure 3.6. *The main differences between OpenCV and OpenVX. Modified from [79].*

First of all, both compared APIs are very useful and have their specific use cases in ML. The purpose of this comparison is to provide reasons for choosing one over the other depending on a given situation. OpenCV is an open-source library maintained by a community. OpenVX is a Khronos-defined API with full conformance tests. It means that even though OpenCV has a lot of tests, it is not regulated by any organization. Khronos, on the other hand, regulates the whole system of OpenVX and every new implementation must pass certain conformance tests. OpenVX is consistent in a way that all core functions must be available in all conformant implementations. OpenCV functions may vary depending on the platform. The drawback of OpenVX is that because of its conformance and consistency limitations, it cannot provide as many functionalities as OpenCV. But OpenVX focuses on optimized hardware-accelerated functions. To sum it up, OpenCV is more suitable for rapid demos and OpenVX for production deployment.

Arm Compute Library

Arm has created a collection of low-level functions for ML which are optimized for Cortex-A CPU and Mali GPU architectures and called Arm Compute Library [83]. ML acceleration is done for Mali GPU using OpenCL [83]. For Cortex-A CPU, the acceleration is done by Neon or Scalable Vector Extension (SVE) which are low-level tools for ARM CPU processing [83].

CUDA and cuDNN

Nowadays, most of the DL research is done with GPUs [48], more specifically NVIDIA GPUs, because of their speed and flexibility. Also NVIDIA was one of the first vendors that started to provide DL acceleration tools for users and it has a better framework and community support. For example TF supports only NVIDIA GPU cards [84]. PyTorch just released version 1.8 that supports ROCm which is AMD's solution for GPU-accelerated computing.

CUDA (Compute Unified Device Architecture) is a platform for parallel computing for NVIDIA's GPUs. It includes GPU-accelerated libraries, a compiler, development tools and CUDA runtime. [85]

NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library for DL development. It provides highly optimized DL operations such as forward and backward convolution, pooling, normalization, and activation layers. This allows deep learning researchers and framework developers to focus on application/software development rather than low-level GPU performance tuning. cuDNN is used in many DL frameworks including Caffe2, Keras, MATLAB, PyTorch and TF. [86]

3.3 Deep Learning Framework Compatibility

Compatibility is one of the most important aspects when choosing a framework for embedded inference. For example, PyTorch models cannot be used in Google Coral since it support only Google's TFLite-based models.

Even if models can be converted from one framework to another with a file exchanger, it can sometimes be tricky [87]. Likewise, even though TensorRT supports ONNX models, it was found out that TensorRT works more seamlessly with TF [50] (page 56). Therefore, Rosebrock et al. [50] (page 431) recommend using TF because it is supported in many different hardware and frameworks such as OpenVINO, TensorRT and TFLite. This makes it easy to switch between hardwares and still use the same trained TF model.

Compatibility issues can become very expensive for companies that use a variety of frameworks and different hardware vendors. The vendors would prefer their users to only buy their HW solutions. For this reason, vendor-specific frameworks are always a risk. As mentioned later in Chapter 4.4 NXP has created a vendor-specific framework for S32V234 board development. This tool was hard to use and the program written for the other boards was not compatible with it. For companies and end-users, it would be more convenient if the same program was used in every hardware. The Khronos Group is enabling this by providing computing standards for free to any company, institution, or individual [88]. Several HW and SW tools are utilizing Khronos standards.

When choosing the hardware it is important to keep in mind which features are needed by the user and which tools are supported by the manufacturer. For example which DL data formats and frameworks are supported and which versions of those. The most popular ones are Keras, TF, TensorFlow Lite, PyTorch, Caffe2 and ONNX. Users should only focus on the higher-level tools. Manufacturers ensures that the hardware is used in the most optimal way.

4 HARDWARE TEST MATERIALS AND METHODS

In this section test methods and tested hardware are presented. First, the test algorithm is introduced. Then a selection of embedded AI devices is evaluated based on development experience and performance. The hardware performance results are presented in Chapter 5.

4.1 Test methods

The test application used in these boards is called "emotion detection". The detection is done for each frame separately and previous predictions do not affect to the next ones. Most video-based algorithms work this way but sometimes average filters could be used to filter one frame misclassification out. This application uses RGB images and does image processing to it so that the image is compatible with the detection algorithms. The aim was to use test applications consisting of typical application development workflows for which publicly available and well-supported solutions can be utilized (i.e. face region detection) combined with custom-developed proprietary inference models using camera stream as input data. The objective was to test how different frameworks were supporting the workflows and identify typical problems encountered by developers when transferring the execution to edge devices.

First, the image is fed to the Haar Cascade object detector (HCOD) which is trained to find faces in the image. Haar Cascade is an ML model that is trained by using positive and negative images. It was first proposed by Paul Viola and Michael Jones in 2001 [89].

- **Positive images** contain images of objects that are supposed to be identified.
- **Negative images** contain images of everything else except for objects that are intended to be identified.

After the classifier is trained, features need to be extracted from the image patch. That is done with Haar features such as shown in Figure 4.1 that are similar to convolutional kernels. The feature is a single value calculated by subtracting the pixel values under the white area from the pixel values under the black area. The greater difference is proportional to the probability of the edge in that region [90, 91].

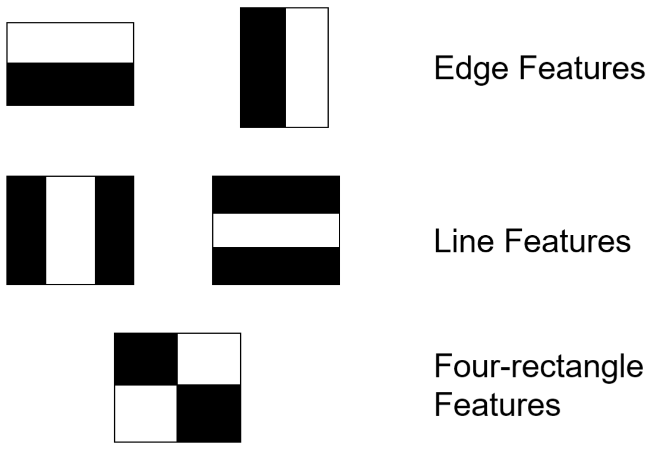
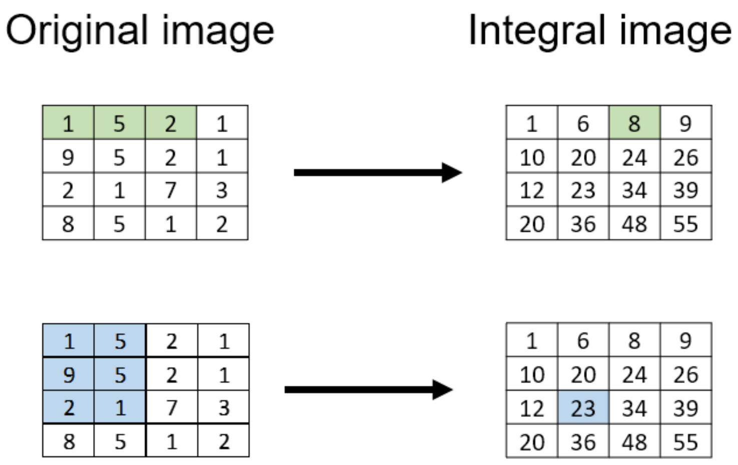


Figure 4.1. *Haar Cascade features*

Unfortunately, it would be too heavy operation for computers to calculate all possible sizes and locations for each feature. Viola and Jones proposed in their paper that it could be done with an integral image. An integral image is constructed from the original image by summing pixel values from left to right and top to bottom e.g. the fifth pixel of the first row in integrated image is the sum of the first five pixels of the row in the original image. This is illustrated in Figure 4.2. [90, 91]



1	5	2	1
9	5	2	1
2	1	7	3
8	5	1	2

1	6	8	9
10	20	24	26
12	23	34	39
20	36	48	55

Figure 4.2. *Every pixel in the integral image is sum of pixels in its left and above.*

Now only four value additions are needed for any feature size. This reduces the complexity of feature calculations dramatically. In the original paper Viola and Jones had 180 000 features from which they used AdaBoost to find the approximately 6000 best ones. Running 6000 features for every section of the image is still too computationally expensive for real-time face detection. For this Viola and Jones introduced the Cascade Classifier which is a multi-stage classifier that applies a set of features in every stage. The first stages contain rougher features compared to later stages that are more complex to find smaller details of the face. If any given stage claims that there is a face in the section, that section is passed on to the next stage. After the section is passed by all stages, a face is confirmed. On the other side, as soon as even one stage states that there is no face in that section, it is discarded right away. This method saves a lot of time and lets the algorithm spend more time in other sections of the image. [89, 91]

Once all the faces are detected, images are cropped, resized, and fed to an emotion detection algorithm. The emotion detection algorithm is proprietary to BHTC and it won't be explained in detail due to confidentiality reasons. The emotion detection model was developed with Keras and, depending on the target device, it is converted to another framework. There are seven output emotions: anger, disgust, fear, happy, sad, surprise, and neutral. When the emotion is detected, a rectangular is drawn around the face and emotion and imprinted above the rectangular. For the comparison the displayed values are face detection time, emotion detection time, and processed frames per second (FPS).

Because HW accelerators such as GPU or TPU cannot be used with HCOD, alternative solutions for face detection were studied for optimized and faster performance. However, HCOD is a good benchmark because it can be used as-is with most of the boards.

In the following sections, the tested hardware is introduced. The development experience is also discussed with the unique properties of each board.

4.2 Google Coral

Google Coral is a general-purpose board for ML applications. Google has a few products that feature a Tensor Processing Unit (TPU) such as Coral Dev Board and USB Accelerator (Figure 4.3). The USB accelerator is like a USB stick that can be inserted into a Raspberry Pi, for example. The Dev board is a single-board computer that doesn't need anything else to be used, except a camera in case images or videos are required.

Coral boards are comparably cheap. The USB accelerator costs around USD 60 and the Dev Board is about USD 130. Both only support TF Lite models which is a disadvantage compared to solutions. In this work, only the Coral Dev Board was evaluated. This board features an NXP i.MX 8M SoC (quad Cortex-A53) CPU, an integrated GC7000 Lite Graphics GPU, and 1 GB RAM. The ML accelerator is a Google Edge TPU coprocessor which is an ASIC designed by Google. [92]

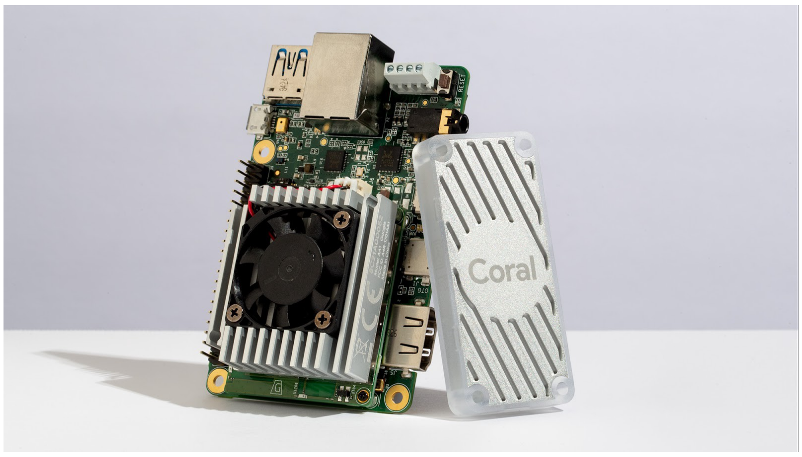


Figure 4.3. *Google Coral Dev Board (left) and USB accelerator (right) [93]*

Getting started with Coral Dev Board was easy. Google [94] and other authors provide several guides on how to set up the board. The first step is to flash the Mendel Linux image to the board. Mendel Linux is a lightweight derivative of Debian Linux. This was easy to do with instructions from the Coral website [94]. After flashing the operating system, the board was up and running. The only challenge was the installation of OpenCV which is an open-source CV and ML software library. Because building OpenCV is computationally heavy and it requires more RAM than there is available on the board. To tackle this issue, instructions by Petar Jalušic' were referred to [95]. An advantage of this board is the broad community support which can support development work.

The camera used for testing was Google's 5-megapixel camera module designed for Coral boards. The test script was done with Python. Coral does not have a graphical interface, thus the video stream was displayed with a Flask [96] server. Flask is a lightweight Web Server Gateway Interface (WSGI) framework. In Coral, Flask was used to create web applications where the output video stream can be displayed. The emotion detection Keras model (.h5) had to be converted to TFLite format before it could be used.

4.3 NVIDIA Jetson Nano

NVIDIA Jetson Nano Developer Kit is a small low-powered single-board computer, that can run multiple NNs in parallel. This board has a Quad-core ARM A57 CPU, a 128-core Maxwell GPU that can run up to 472 GFLOPS (floating point operations per second) and 4 GB of RAM. [97], [50] (page 265). This board is compatible with a Raspberry Pi camera so that was used in the tests.

NVIDIA's TensorRT (Section 3.2.3) is a crucial part of the development with Jetson Nano. It allows optimization of performance, power and efficiency. Developing on NVIDIA Jetson Nano has been made easy because the Maxwell GPU supports OpenGL, Vulkan and CUDA. OpenGL is a processing unit API developed by the Khronos Group. Thus,

Figure 4.4. *NVIDIA Jetson Nano Developer Kit [97]*

OpenCV can use CUDA for more optimized operations. In theory, some PC applications could be used on Jetson Nano as well. However, better performance can be obtained if models and scripts are optimized for the device.

Building the development environment was especially easy with this board due to the ready-made image by PyimageSearch. The image was included in the Complete Bundle of Raspberry Pi for Computer Vision book. The only requirement before starting the development work was to write the image to the SD-card and boot the device. The image included:

- System-level packages
- OpenCV compiled from source (CUDA-capable)
- NVIDIA's distribution of TensorFlow 1.13.1
- TensorRT
- Python libraries as needed

Python packages were installed in virtualenv which is one of the best Python package managers. The PyimageSearch team estimated that installing all the required software by hand would take approximately three to five days. In addition to the ready image, they provided instructions for setting up the Jetson Nano board [98]. There are guides on how to set up important software such as SSH, OpenCV, Tensorflow and TensorRT.

4.4 NXP S32V234

The S32V234 is NXP's second-generation vision processing family with ARM Quad Cortex-A53 cores processors. It has a 3D GPU (GC3000) with OpenCL, OpenGL, and OpenVG and dual APEX-2 vision accelerators for deep learning inference enabled by OpenCL, APEX-CV and APEX graph tool [99]. This board is much more expensive than the previous two with a price of around USD 750.

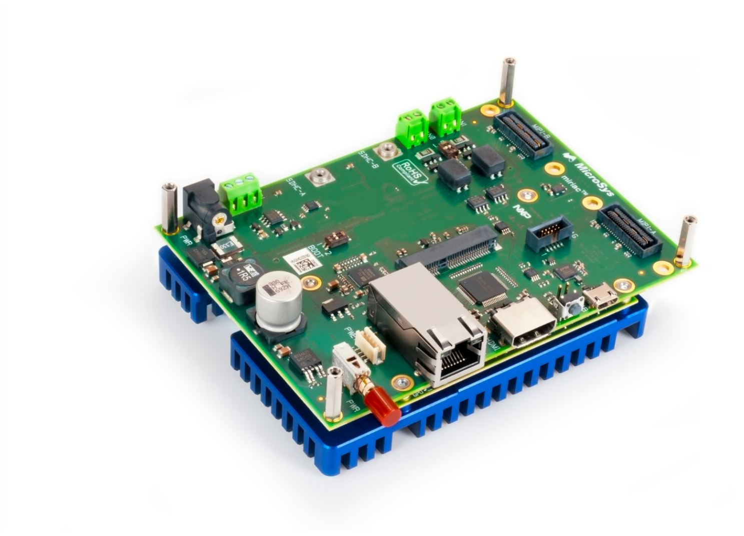


Figure 4.5. *NXP's S32V234 Vision and Sensor Fusion Evaluation Board [100]*

Developing with this board is challenging because NXP uses vendor-specific software development frameworks. These frameworks have steep learning curves and some features require additional coding. In this case, this system does not use Video4Linux for camera controlling. Video4Linux is a set of device drivers for video stream capturing on Linux systems. This means that controlling cameras require self-made drivers which is laborious to do. In addition, some components, such as Yocto and S32 Design Studio, require old versions of Ubuntu and other software.

Building Linux with Yocto

Yocto Project is an open-source collaboration project that is designed for creating custom Linux operating systems for embedded devices. The advantage of this approach is that the Linux system includes only the essential components so that it won't waste any memory or performance on unnecessary processes. Yocto is based on the layer model. Each layer gives the OpenEmbedded build system instructions on what to do. There are lots of layer repositories that can be modified for the specific needs of a given product. [101]

Instructions on how to build Linux can be found in NXP's user manual for S32V234 on their website [102]. However, building with Yocto for this board was somewhat problematic. In the user manual there are instructions for building the Linux BSP (Board support package) using Yocto. The Linux kernel used is version 4.14 which is already a few years old. Furthermore, there are old versions of some components such as OpenCV. Updating to a newer Linux kernel (version 5.4) is challenging because the NXP-provided Advanced Driver Assistance Systems (ADAS) have some drivers that are incompatible with Linux kernel 5.4. This hampers the use of the most recent SW tools.

Software Development

NXP's S32 Design Studio for Vision is an Integrated Development Environment (IDE) for software development for S32V234 based on Eclipce. The development experience with this was not very good. for instance, the S32 Design Studio installation guide is only for Ubuntu 14 and 16. However, the installation was successful on Ubuntu 18.04 by adapting the original guide. [103] NXP encourages to use of the S32 Design Studio for the S32 Platform which supports also Ubuntu 18. The learning curve with Design Studio is rather steep and it is hard to get started with it especially because community help is almost nonexistent.

Another way to develop software for this board is to use NXP's Vision Toolbox for MATLAB. In principle, the code can be developed in MATLAB and then generate and deploy the C/C++ code to the target device. In addition to the C/C++ code the output includes Makefile to build the application on the target device, S23V234 in this case. This MATLAB add-on can be installed by following instructions in the NXP Support Package for S32V234 found in MATLAB add-ons. It requires free-of-charge license which can be downloaded from NXP's website and the NXP Vision SDK software package.

However, MATLAB development is not easy. First, not all MATLAB functions can be used with the MATLAB Coder and Embedded Coder. Second, it is unclear how or if it is even possible to generate Makefile for the target device. The generated Makefile is made for the host computer, Windows PC in this case. It is not possible to use Linux because all the needed tools have no Linux support.

There is a demo application for face detection provided by NXP. It was used to optimize the face detection time. Another reason was that the face detection algorithm development would have taken too much time and it wouldn't have been time efficient. This is also the reason why there is no HCOD test.

4.5 Raspberry Pi 4

Raspberry Pi is a series of the most popular single-board computers by the Raspberry Pi Foundation. The first Raspberry Pi was released in 2012 and since then there have been updated releases every year. In this work, the used board is the Raspberry Pi 4 with 4 GB RAM. It costs around EUR 65. It has an ARM Quad Core Cortex-A72 CPU with build-in WiFi and Bluetooth. Raspberry Pi is not primarily designed to be used for DL so it doesn't have any HW acceleration components. That is why only the HCOD test was done with it. However, the Google Coral USB accelerator can be used with Raspberry Pi. Adrian Rosebrock considers this topic in his book "Raspberry Pi for Computer Vision" [50].

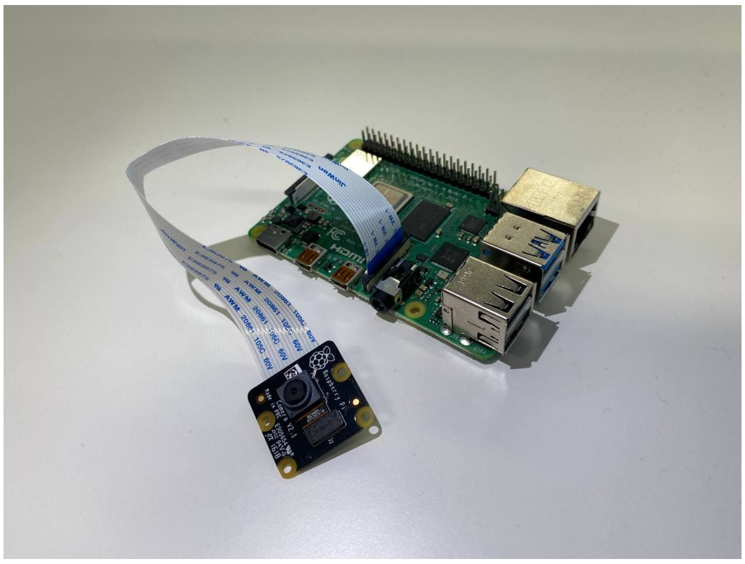


Figure 4.6. *Raspberry Pi 4 with Camera Module V2*

With that book [50] there was an Raspberry Pi 4 Linux image provided with the most important software for DL and CV development including OpenCV. According to this book, Raspberry Pi works well with many different DL frameworks. However, if OpenVINO is used for optimization then TF and Caffe are the best choices [50]. Because the emotion detection model used was already converted to TFLite for the Google Coral implementation it was interesting to test how well it would work on the Raspberry pi 4.

The only Python package that was not installed in the image was TFLite runtime. That was straight forward with the TF's instructions [104].

In the Table 4.1 is shown a summary of the SBCs specification.

Table 4.1. *Summary of the SBCs used in the performance test*

SBC	CPU	RAM	OS	Recommended FW
Google Coral	Quad Cortex-A53	1 GB	Mendel Linux	TensorFlow Lite
NVIDIA Jetson Nano	Quad-core ARM A57	4 GB	Ubuntu 18.04	Caffe, PyTorch and TensorFlow (Use TF take advantage of TensorRT)
NXP S32V234	Quad Cortex-A53	-	Custom Linux built with Yocto	MATLAB
Raspberry Pi 4	Quad core Cortex-A72	4 GB	Raspberry Pi OS	Tensor-Flow, Caffe

5 HARDWARE TEST RESULTS

The hardware performance test results are shown in the tables below. There were two changing variables, face detection algorithm and input data source. The input sources were camera stream and recorded video. The recorded video used was the same in every test. This leads to four combination of tests. All values shown in the tables are averages of the time taken to process a single frame. First, there is Table 5.1 of HCOD face detection with camera stream input. Second, Table 5.2 is the same but with a standard video. In the third Table 5.3, face detection was done with the more optimal way. The implementation was depending on the board. Finally, Table 5.4 contains results of optimal face detection with standard video. All tests could not be executed. For example Raspberry Pi does not have any DL hardware accelerator so considerable improvements were not expected with another implementation. The NXP S32V234 development was so difficult that it would have been too time-consuming to perform all the tests with it.

Table 5.1. *HCOD test with camera stream: the values are averages of one frame processing.*

Haar Cascade, Stream	Face Det (ms)	Emotion Det (ms)	Frames per Sec (FPS)
Google Coral	319.77	73.69	2.37
Jetson Nano	200.20	38.19	4.19
NXP S32V234	-	-	-
Raspberry Pi	254.89	10.56	3.42

Table 5.2. *HCOD test with standard video: the values are averages for one frame processing*

Haar Cascade, Video	Face Det (ms)	Emotion Det (ms)	Frames per Sec (FPS)
Google Coral	313,23	72,36	2,51
Jetson Nano	260,01	18,33	3,59
NXP S32V234	-	-	-
Raspberry Pi	255,98	10,05	3,40

Table 5.3. *Optimal test with camera stream: the values are averages for one frame processing.*

Optimal, Stream	Face Det (ms)	Emotion Det (ms)	Frames per Sec (FPS)
Google Coral	7,60	14,89	18,91
Jetson Nano	95,08	32,09	10,94
NXP S32V234	22,74	16,15	25,38
Raspberry Pi	-	-	-

Table 5.4. *Optimal test with standard video: the values are averages for one frame processing*

Optimal, Video	Face Det (ms)	Emotion Det (ms)	Frames per Sec (FPS)
Google Coral	7,31	14,55	23,22
Jetson Nano	155,19	13,09	6,06
NXP S32V234	-	-	-
Raspberry Pi	-	-	-

Google Coral

The first emotion detection test was done with the HCOD. The performance wasn't very good, but this was expected because the HCOD was calculated with a CPU. The nput size for the Haar Cascade was 640x480 pixels. In the stream test, a 500 frame average was calculated and for face detection the average inference time was 319,8 ms and for emotion detection 73,7 ms. The results for the standard video were very similar: face detection time was only 6,5 ms lower and emotion detection time was 1,3 ms lower. The difficulty in evaluating HCDO is that the algorithm calculation time varies a lot depending on the face location. If the face is in the bottom right corner the inference time is around 200 ms. The other extreme placement is the top left corner where the inference time is close to 400 ms. In the video stream test the face was in the middle of the image. This board was the worst in the HCOD test, however, not significantly. One thing that can affect its perfomance is the Flask. It is hard to determine how much it reduces the performance.

The second face detection model for Coral is called MobileNet SSD v2 (Faces) [105]. This model is pre-compiled by Google's Coral team thus the performance should be rather close to the absolute optimal.

MobileNet SSD v2 (Faces):

- Detects the location of human faces
- Dataset: Open Images v4
- Input size: 320x320

This model accelerated the inference time significantly. The face detection time dropped from 319,8 ms to 7,6 ms. This performance boost is mostly due to TPU usage which this model could utilize. Interestingly, also the emotion detection time decreased from 73,7 ms to 14,9 ms. This might have been due to the slightly different bounding box of these models' outputs. There was almost no difference between the camera stream and the standard video tests.

The downside of the MobileNet model was that it was not particularly robust. For example, the HCOD and Dlib (used with Jetosn Nano) face detection gave almost the same bounding box for every frame if the face was still. For MobileNet, the bounding box was inconsistent and it varied a lot between frames. Another considerable difference is that the input image for MobileNet was 320x320 whereas for HCOD it was 640x480. This means that were three times more pixels in the HCOD than in the MobileNet face detection. The HCOD face detection test was also made with the same image size as the one with MobileNet. This reduced the face detection time to 118 ms. The emotion detection time was similar to before (72 ms) which was expected. This demonstrates that different implementations are not easily comparable. As shown by the results, there is always some kind of trade-off between accuracy, robustness, and speed.

NVIDIA Jetson Nano

The first test (HCOD) application was the same for the Google Coral. This time, Flask was not used because Jetson Nano supports Linux OS with a GUI. The Haar Cascade test performance was much better on the Jetson Nano than it was on Coral. The face detection time was abetter by a third (200,2 ms) compared to Coral's performance in the camera stream test. As pointed out before, the face location affects the HCOD's face detection time, thus the standard video test provides more comparable results. However, there is a significant difference between Google Coral and NVIDIA Jetson Nano. The emotion detection time was 38,2 ms which was almost two times faster than it was on Coral with the HCOD test. Still, emotion detection was over two times slower than it was with Coral's optimal test.

To be able to run the Keras model (.h5) on Jetson Nano it has to be converted to a frozen graph. Frozen graphs reduce the size of a model by removing useless information, e.g. information saved in the checkpoint files such as gradients. As the name implies, a frozen graph cannot be trained afterward because of lost information. Now, everything for inferencing on a PC would be ready. However, to achieve better performance on an embedded device, the frozen graph is further optimized with TensorRT.

Various different methods for a more optimal face detection were tested. A GPU-based test was done with Dlib's CNN face detection model [106]. This resulted in a heavy application and the RAM started to restrict the performance. The application displayed the following warning: "Allocator (GPU_0_bfc) ran out of memory trying to allocate 69.09MiB. The caller indicates that this is not a failure, but may mean that there could be performance gains if more memory were available". Nevertheless, this resulted better performance than the HCOD as the face detection time was reduced to 95,08 ms and the emotion detection time to 32,09. In principle, the emotion detection time should not be affected by the face detection algorithm but different algorithms result in a slightly different bounding box which can be seen with a varying emotion detection time.

Even though the performance was reduced by memory, Dlib's face detector was able to detect the face better than HCOD. It was highly accurate, very robust and it could detect the face from varying angles.

NXP S32V234

The NXP S32V234 board had the best performance compared to any other tested board. This NXP board was also the most expensive and it is not designed for private use but rather for more demanding tasks such as driver monitoring systems. The dual APEX-2 vision accelerators performance can be seen from an FPS count. Even if the face and emotion detection times are slower than with the Coral board, the NXP board was still 34 % faster in FPS. This is due to more optimized image processing.

Again, these results are not directly comparable between other boards because the NXP face detection application uses an HD image (1280x720 pixels) which is more than 6 time bigger than the image used with Coral's MobileNet. Thus, in reality, the NXP S32V234 is the most powerful board of all tested in this thesis.

Raspberry Pi 4

Raspberry Pi 4 doesn't have an accelerator for DL calculus so only the HCOD test was performed with it. The camera stream and standard video tests showed very similar results. The performance of the NVIDIA Jetson Nano and Raspberry Pi was very similar. The Raspberry Pi is faster with emotion detection and the Jetson Nano with face detection. FPS counts with standard videos were around 3,5 for both.

6 DISCUSSION AND CONCLUSIONS

The aim of this thesis was to review software tools needed in embedded deep learning applications, use them to develop applications, and run the applications on selected Single-board Computers (SBC). The most used frameworks and tools were compared and evaluated. In addition, four SBCs were tested successfully even though all applications weren't tested on the NXP board due to development difficulties. Based on that it can be said that the aims have been achieved. This field of science is very rapidly progressing which means that statements regarding software tools covered in this thesis may not be valid anymore in a few years.

All the most popular frameworks introduced in this thesis are similar in terms of performance. However, the developing workflow varies a lot. PyTorch is good for developing new architectures and easy to debug but it may be hard for beginners. TF has a steep learning curve as well. TF is good for embedded deep learning because TF Lite is commonly supported in mobile phones and embedded devices. To make TF model development easier, Keras is the best option. Keras is a high-level API operating on top of TF. MATLAB has been developing its deep learning toolboxes a lot in the past few years. There are some advantages over TF and PyTorch such as code generation functionalities. However, as experienced in this thesis it requires learning and it is not as easy and convenient as it sounds.

Compatibility between software and hardware is crucial for companies because it can be very expensive if the software has to be rewritten for every hardware separately. Standardized APIs, such as Khronos Group's APIs, are helping hardware manufacturers to make such boards which are easy to program and enable reusing the code. When choosing the hardware supported frameworks, this needs to be checked to ensure that desired model can be run on the device.

Tested SBCs were Google Coral, NVIDIA Jetson Nano, NXP S32V234 and Raspberry Pi 4. The test algorithm was emotion detection which included two parts. First, the face was detected and cropped, and then the face was fed to the emotion detection model. Each board was tested with two different face detection algorithms, Haar Cascade object detector (HCOD) and a more optimal one (depending on the board). A camera stream and recorded video were used as input. These two variables, face detection algorithm and input source, produced four tables of results. The emotion detection algorithm the was same in every test. HCOD tests were comparable because the input image size and other aspects were similar for each board. Especially, HCOD with pre-recorded videos were comparable because head angle and position in the image affect the processing time. The more optimal tests were so different that they were not comparable. For example, Google Coral with MobileNet took in a 320x320 image and NXP S32V234 took an HD image (1280x720). This means that the NXP face detection processed 9 times more pixels than Google Coral.

It was not possible to perform all the scheduled tests. With Raspberry Pi 4 only the HCOD test could be done because it did not have any parallel computing processors. In addition, only the optimal test was done with NXP because of the development issues. It took too much time and resources to get even one algorithm working so the decision was made to not even try others. The development experience with other boards was much better. The Google Coral Dev Board had great instructions on its web-site. Also, there were multiple good guides from users for various tasks such as installing OpenCV. Nvidia Jetson Nano and Raspberry Pi 4 had great community support as well and most importantly Adrian Rosebrock's Linux images saved a lot of time. In the images all the necessary tools such as OpenCV, TF etc. were already installed.

However, enough results were gathered to draw conclusions. The NXP's board had the best performance. With HCOD all the boards (except NXP) had very similar performances. The Coral had a slightly worse (FPS 2,37) and the Jetson Nano a little better (FPS 4,19) performance, but they were in the same class. Although the results from the optimal test are not well comparable the NXP was clearly the fastest board. The face and emotion detection times were not the best but the rest of the image processing was so fast that the NXP board can be determined to be the best.

This thesis overviewed DL tools and compared them on a general level. Thus, most of the tools were not actually used. For further studies, it would be interesting to test deep learning frameworks in terms of training. Training a model architecture with the same data set with different frameworks would give information about the training process and performance. Nowadays, the embedded device most used by people is the mobile phone. The use of mobile phones in e.g. Driver Monitoring System (DMS) applications would be worth researching. In addition, similarly to deep learning frameworks, also vendor-specific SDKs such as Nvidia and Arm NN SDK would be interesting to test and compare.

REFERENCES

[1] Mraz, S. *Prediction: Market for artificial intelligence in cars will grow 1,200% in next six years*. English. 2020.

[2] Hussein, R., Palangi, H., Ward, R. and Wang, Z. J. Epileptic Seizure Detection: A Deep Learning Approach. English. (Mar. 2018). URL: https://arxiv.org/abs/1803.09848.

[3] Yang, F., Poostchi, M., Yu, H., Zhou, Z., Silamut, K., Yu, J., Maude, R. J., Jaeger, S. and Antani, S. Deep Learning for Smartphone-Based Malaria Parasite Detection in Thick Blood Smears. English. *IEEE journal of biomedical and health informatics* 24.5 (Sept. 2019), 1427–1438. DOI: 10.1109/JBHI.2019.2939121. URL: https://ieeexplore.ieee.org/document/8846750.

[4] Copeland, J. *What is Artificial Intelligence?* Accessed: 28.10.2020. May 2000. URL: http://www.alanturing.net/turing_archive/pages/Reference%5C%20Articles/What%5C%20is%5C%20AI.html.

[5] Gavrilova, Y. Artificial Intelligence vs. Machine Learning vs. Deep Learning: Essentials. (Apr. 2020). Accessed: 30.10.2020. URL: https://serokell.io/blog/ai-ml-dl-difference.

[6] *The Difference Between Deep Learning Training and Inference*. Accessed: 30.9.2020. URL: https://www.intel.com/content/www/us/en/artificial-intelligence/posts/deep-learning-training-and-inference.html.

[7] Shridhar, K., Lee, J., Hayashi, H., Mehta, P., Iwana, B. K., Kang, S., Uchida, S., Ahmed, S. and Dengel, A. ProbAct: A Probabilistic Activation Function for Deep Neural Networks. English. (May 2019). URL: https://arxiv.org/abs/1905.10761.

[8] Doshi, C. Why Relu? Tips for using Relu. Comparison between Relu, Leaky Relu, and Relu-6. *Medium* (June 2019). URL: https://medium.com/@chinesh4/why-relu-tips-for-using-relu-comparison-between-relu-leaky-relu-and-relu-6-969359e48310.

[9] Ramadhan, L. Neural Network: The Dead Neuron. *Towardsai* (Nov. 2020). URL: https://pub.towardsai.net/brain-damage-on-artificial-intelligence-37bc53023ab8.

[10] Kumar, S. S. Why Sigmoid?: *Medium* (Nov. 2019). URL: https://medium.com/n%5C%C3%5C%BAcleoml/why-sigmoid-ee95299e11fd.

[11] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. English. (Sept. 2014). URL: https://arxiv.org/abs/1409.1556.

[12] Model optimization. (). Accessed: 21.1.2021. URL: https://www.tensorflow.org/lite/performance/model_optimization.

[13] Albawi, S., Mohammed, T. A. and Al-Zawi, S. Understanding of a convolutional neural network. English. IEEE, Aug 21, 2017, 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186. URL: https://ieeexplore.ieee.org/document/8308186.

[14] Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K. and Zhang, J. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. English. *Proceedings of the IEEE* 107.8 (2019), 1738–1762. DOI: 10.1109/jproc.2019.2918951. URL: https://search.datacite.org/works/10.1109/jproc.2019.2918951.

[15] Shi, W., Cao, J., Zhang, Q., Li, Y. and Xu, L. Edge Computing: Vision and Challenges. English. *IEEE internet of things journal* 3.5 (Oct. 2016), 637–646. DOI: 10.1109/JIOT.2016.2579198. URL: https://ieeexplore.ieee.org/document/7488250.

[16] Kavanagh, S. *5G vs 4G: No Contest*. Accessed: 28.5.2021. Mar. 2020. URL: https://5g.co.uk/guides/4g-versus-5g-what-will-the-next-generation-bring/.

[17] Caulfield, B. *What's the Difference Between a CPU and a GPU?* 2009. URL: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/.

[18] *What Is a GPU?* Accessed: 1.10.2020. URL: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html.

[19] Wang, X., Han, Y., Leung, V. C. M., Niyato, D., Yan, X. and Chen, X. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. English. *IEEE Communications surveys and tutorials* 22.2 (2020), 869–904. DOI: 10.1109/COMST.2020.2970550. URL: https://ieeexplore.ieee.org/document/8976180.

[20] *Field Programmable Gate Array (FPGA)*. Accessed: 1.10.2020. URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html#:~:text=Field%5C%20Programmable%5C%20Gate%5C%20Arrays%5C%20(FPGAs,or%5C%20functionality%5C%20requirements%5C%20after%5C%20manufacturing..

[21] Wang, T., Wang, C., Zhou, X. and Chen, H. An Overview of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. English. (2019), 1674–1681. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00229. URL: https://ieeexplore.ieee.org/document/8855594.

[22] Görner, M. *What are Tensor Processing Units (TPUs)?* Accessed: 18.5.2021. May 2021. URL: https://codelabs.developers.google.com/codelabs/keras-flowers-data/#2.

[23] *Europe agrees on mandatory safety systems for 2022*. Accessed: 2.10.2020. 27 March, 2019. URL: https://autovistagroup.com/news-and-insights/europe-agrees-mandatory-safety-systems-2022.

[24] NHTSA's Overview of Motor Vehicle Crashes in 2019. (Dec. 2020). URL: https://www.nhtsa.gov/risky-driving.

[25] Jung, S.-J., Shin, H.-S. and Chung, W.-Y. Driver fatigue and drowsiness monitoring system with embedded electrocardiogram sensor on steering wheel. English. *IET intelligent transport systems* 8.1 (Feb. 2014), 43–50. DOI: 10.1049/iet-its.2012.0032. URL: http://digital-library.theiet.org/content/journals/10.1049/iet-its.2012.0032.

[26] Boyraz, P., Acar, M. and Kerr, D. Multi-sensor driver drowsiness monitoring. English. *Proceedings of the Institution of Mechanical Engineers. Part D, Journal of automobile engineering* 222.11 (Nov. 2008), 2041–2062. DOI: 10.1243/0954407\0JAUTO513. URL: https://journals.sagepub.com/doi/full/10.1243/09544070JAUTO513.

[27] Kumar, A. and Patra, R. Driver drowsiness monitoring system using visual behaviour and machine learning. English. (Apr 2018), 339–344. DOI: 10.1109/ISCAIE.2018.8405495. URL: https://ieeexplore.ieee.org/document/8405495.

[28] Jo, J., Lee, S. J., Kim, J., Jung, H. G. and Park, K. R. Vision-based method for detecting driver drowsiness and distraction in driver monitoring system. English. *Optical Engineering* 50.12 (Dec. 2011), 127202. DOI: 10.1117/1.3657506. URL: http://dx.doi.org/10.1117/1.3657506.

[29] Galarza, E. E., Egas, F. D., Silva, F. M., Velasco, P. M. and Galarza, E. D. Real Time Driver Drowsiness Detection Based on Driver's Face Image Behavior Using a System of Human Computer Interaction Implemented in a Smartphone. English. Proceedings of the International Conference on Information Technology & Systems (ICITS 2018) (Jan. 2018), 563–572. DOI: 10.1007/978-3-319-73450-7_53. URL: http://link.springer.com/10.1007/978-3-319-73450-7_53.

[30] Reddy, B., Kim, Y.-H., Yun, S., Seo, C. and Jang, J. Real-Time Driver Drowsiness Detection for Embedded System Using Model Compression of Deep Neural Networks. English. (Jul 2017), 438–445. DOI: 10.1109/CVPRW.2017.59. URL: https://ieeexplore.ieee.org/document/8014793.

[31] Schwarz, M. *Look Who's Driving*. 2019.

[32] Shuttleworth, J. *SAE Standards News: J3016 automated-driving graphic update*. Jan. 2019. URL: https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic.

[33] Parrick, M. *The Future of Car Insurance*. Accessed: 26.5.2021. Nov. 2018. URL: https://brownandjoseph.com/blog/future-car-insurance/.

[34] Hendrickson, J. What Are the Different Self-Driving Car "Levels" of Autonomy?: (Jan. 2020). URL: https://www.howtogeek.com/401759/what-are-the-different-self-driving-car-levels-of-autonomy/.

[35] Hetzner, C. Audi quits bid to give A8 Level 3 autonomy. (Apr. 2020). URL: https://europe.autonews.com/automakers/audi-quits-bid-give-a8-level-3-autonomy.

[36] Davies, A. BMW Takes Self-Driving to the Next Level. (June 2020). URL: https://www.autoweek.com/news/technology/a32852529/bmw-takes-self-driving-to-the-next-level/.

[37] Honda launches next generation Honda SENSING Elite safety system with Level 3 automated driving features. (Mar. 2021). URL: https://hondanews.eu/eu/et/corporate/media/pressreleases/329456/honda-launches-next-generation-honda-sensing-elite-safety-system-with-level-3-automated-driving-feat.

[38] Dickson, B. Sorry, Elon: Fully Autonomous Tesla Vehicles Will Not Happen Anytime Soon. *PC Magazine* (July 2020). URL: https://www.pcmag.com/opinions/sorry-elon-fully-autonomous-tesla-vehicles-will-not-happen-anytime-soon#:~:text=achieve%5C%20that%5C%20goal.-,Level%5C%205%5C%20Autonomy,National%5C%20Highway%5C%20Traffic%5C%20Safety%5C%20Administration..

[39] Kalmet, P. H. S., Sanduleanu, S., Primakov, S., Wu, G., Jochems, A., Refaee, T., Ibrahim, A., Hulst, L. V., Lambin, P. and Poeze, M. Deep learning in fracture detection : a narrative review. English. *Acta orthopaedica (2020)* (2020), 1. DOI: 10.18154/rwth-2020-02484. URL: https://search.datacite.org/works/10.18154/rwth-2020-02484.

[40] Coller, M., Fu, R., Yin, L. and Christiansen, P. ARTIFICIAL INTELLIGENCE: Healthcare's New Nervous System. (2017). URL: https://www.accenture.com/t20171215t032059z__w__/us-en/_acnmedia/pdf-49/accenture-health-artificial-intelligence.pdf.

[41] Teikari, P., Najjar, R. P., Schmetterer, L. and Milea, D. Embedded deep learning in ophthalmology: making ophthalmic imaging smarter. English. *Therapeutic advances in ophthalmology* 11 (Mar. 2019), 251584141982717–2515841419827172. DOI: 10.1177/2515841419827172. URL: https://search.datacite.org/works/10.1177/2515841419827172.

[42] Holzinger, A., Biemann, C., Pattichis, C. S. and Kell, D. B. What do we need to build explainable AI systems for the medical domain? English. (Dec. 2017). URL: https://arxiv.org/abs/1712.09923.

[43] Nguyen, A., Yosinski, J. and Clune, J. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. English. IEEE, Jun 2015, 427–436. ISBN: 1063-6919. DOI: 10.1109/CVPR.2015.7298640. URL: https://ieeexplore.ieee.org/document/7298640.

[44] Su, J., Vargas, D. V. and Sakurai, K. One Pixel Attack for Fooling Deep Neural Networks. English. *IEEE transactions on evolutionary computation* 23.5 (Oct. 2019), 828–841. DOI: 10.1109/TEVC.2019.2890858. URL: https://ieeexplore.ieee.org/document/8601309.

[45] Bologna, G. and Hayashi, Y. Characterization of Symbolic Rules Embedded in Deep DIMLP Networks: A Challenge to Transparency of Deep Learning. English. *Journal of Artificial Intelligence and Soft Computing Research* 7.4 (Oct. 2017), 265–286. DOI: 10.1515/jaiscr-2017-0019. URL: http://www.degruyter.com/doi/10.1515/jaiscr-2017-0019.

[46] Warden, P. and Situnayake, D. *TinyML*. Bejing ; Boston ; Farnham ; Sebastopol ; Tokyo: O'Reilly, 2020. ISBN: 9781492052043.

[47] Leung, H. and Haykin, S. The complex backpropagation algorithm. English. *IEEE transactions on signal processing* 39.9 (Sept. 1991), 2101–2104. DOI: 10.1109/78.134446. URL: https://ieeexplore.ieee.org/document/134446.

[48] Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., García, Á. L., Heredia, I., Malík, P. and Hluchý, L. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. English. *The Artificial intelligence review* 52.1 (Jan. 2019), 77–124. DOI: 10.1007/s10462-018-09679-z. URL: https://search.datacite.org/works/10.1007/s10462-018-09679-z.

[49] Rubin, R. My Journey in Converting PyTorch to TensorFlow Lite. (Sept. 2020). Accessed: 28.4.2021. URL: https://towardsdatascience.com/my-journey-in-converting-pytorch-to-tensorflow-lite-d244376beed.

[50] Rosebrock, A., Hoffman, D., McDuffee, D., Thanki, A. and Paul, S. *Raspberry Pi for Computer Vision*. 1.0.0. PyImageSearch.com, 2019.

[51] Dancuk, M. PyTorch vs TensorFlow: In-Depth Comparison. (Feb. 2021). Accessed: 29.4.2021. URL: https://phoenixnap.com/blog/pytorch-vs-tensorflow.

[52] Chandram, R. *The What's What of Keras and TensorFlow*. Accessed: 22.10.2020. Apr. 2019. URL: https://www.upgrad.com/blog/the-whats-what-of-keras-and-tensorflow/.

[53] He, H. The State of Machine Learning Frameworks in 2019. *The Gradient* (2019).

[54] *PyTorch 1.8.0 Release Notes*. Accessed: 30.4.2021. Apr. 2021. URL: https://github.com/pytorch/pytorch/releases.

[55] *PyTorch Mobile*. Accessed: 4.5.2021. URL: https://pytorch.org/mobile/home/.

[56] Khronos. *Vulkan 1.2.179 - A Specification (with all registered Vulkan extensions)*. Accessed: 26.5.2021. Apr. 2021. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/chap1.html.

[57] PyTorch: Training a Classifier. (). Accessed: 3.5.2021. URL: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

[58] Karpathy, A. *PyTorch at Tesla*. Accessed: 3.5.2021. Nov. 2021. URL: https://www.youtube.com/watch?v=oBklltKXtDE.

[59] *Caffe*. Accessed: 21.10.2020. URL: https://caffe.berkeleyvision.org/.

[60] *Caffe2 and PyTorch join forces to create a Research + Production platform PyTorch 1.0*. Accessed: 29.10.2020. May 2018. URL: https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html.

[61] Paluszek, M. and Thomas, S. *Practical MATLAB Deep Learning : A Project-Based Approach*. English. Apress, Jan. 2020. URL: https://library.biblioboard.com/viewer/9f7b0068-c392-11ea-8d48-0ae0aa0d175d.

[62] Palm, R. B. Prediction as a candidate for learning deep hierarchical models of data. MA thesis. 2012.

[63] Tanaka, M. Deep Neural Network. *MATLAB Central File Exchange* (2021). Accessed: 17.5.2021. URL: https://www.mathworks.com/matlabcentral/fileexchange/42853-deep-neural-network.

[64] Choen, M. Pattern Recognition and Machine Learning Toolbox. *GitHub* (2021). Accessed: 17.5.2021. URL: https://github.com/PRML/PRMLT.

[65] MATLAB Coder. (). Accessed: 17.5.2021. URL: https://se.mathworks.com/products/matlab-coder.html.

[66] Embedded Coder. (). Accessed: 17.5.2021. URL: https://se.mathworks.com/products/embedded-coder.html.

[67] *What is ONNX?* Accessed: 6.10.2020. Dec. 2019. URL: https://microsoft.github.io/ai-at-edge/docs/onnx/.

[68] Ippolito, P. P. ONNX: Easily Exchange Deep Learning Models. (Sept. 2020). Accessed: 6.10.2020. URL: https://towardsdatascience.com/onnx-easily-exchange-deep-learning-models-f3c42100fd77.

[69] Gill, J. K. What is ONNX? Open Neural Network Exchange Advantages. (May 2019). Accessed: 6.10.2020. URL: https://www.xenonstack.com/blog/onnx/.

[70] (OPTIONAL) EXPORTING A MODEL FROM PYTORCH TO ONNX AND RUNNING IT USING ONNX RUNTIME. (2017). Accessed: 09.10.2020. URL: https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html.

[71] Xu, F. ONNX Runtime is now open source. (Dec. 2018). URL: https://azure.microsoft.com/en-us/blog/onnx-runtime-is-now-open-source/.

[72] Group, N. W. *NNEF and ONNX: Similarities and Differences*. Feb. 2018. URL: https://www.khronos.org/blog/nnef-and-onnx-similarities-and-differences.

[73] *Neural Network Exchange Format (NNEF)*. Accessed: 5.3.2021. URL: https://www.khronos.org/nnef.

[74] *TensorFlow Lite guide*. Accessed: 3.11.2020. URL: https://www.tensorflow.org/lite/guide.

[75] *Model optimization*. Accessed: 21.1.2021. URL: https://www.tensorflow.org/lite/performance/model_optimization.

[76] *NVIDIA TensorRT Documentation*. Accessed: 3.10.2020. URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[77] Roddy, S. *Arm NN: the Easy Way to Deploy Edge ML*. Jan. 2019. URL: https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/arm-nn-the-easy-way-to-deploy-edge-ml.

[78] *Arm NN and Arm Compute Library*. Accessed: 31.5.2021. URL: http://software-dl.ti.com/processor-sdk-linux/esd/docs/05_03_00_07/linux/Foundational_Components_ArmNN.html.

[79] *OpenVX & OpenCL BOF - SIGGRAPH 2016*. Accessed: 22.3.2021. July 2016. URL: https://www.youtube.com/watch?v=aml-CBmOZ4g.

[80] *OpenCL*. Accessed: 21.3.2021. URL: https://www.khronos.org/opencl/.

[81] Howes, L. The OpenCL Specification. (Feb. 2018). URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf.

[82] *OpenVX*. Accessed: 22.3.2021. URL: https://www.khronos.org/openvx/.

[83] *Arm Compute Library*. Accessed: 31.5.2021. URL: https://developer.arm.com/ip-products/processors/machine-learning/compute-library.

[84] *GPU support*. Accessed: 28..4.2021. URL: https://www.tensorflow.org/install/gpu.

[85] *NVIDIA CUDA toolkit*. Accessed: 28.4.2021. URL: https://developer.nvidia.com/cuda-zone.

[86] *NVIDIA cuDNN*. Accessed: 28.4.2021. URL: https://developer.nvidia.com/cudnn.

[87] Rubin, R. My Journey in Converting PyTorch to TensorFlow Lite. (Sept. 2020). Accessed: 12.11.2020. URL: https://towardsdatascience.com/my-journey-in-converting-pytorch-to-tensorflow-lite-d244376beed.

[88] *Khronos, Developer Resource Hub*. Accessed: 9.6.2021. URL: https://www.khronos.org/developers.

[89] Viola, P. and Jones, M. Rapid object detection using a boosted cascade of simple features. English. 1 (2001), I. DOI: 10.1109/CVPR.2001.990517. URL: https://ieeexplore.ieee.org/document/990517.

[90] *Face Detection using Haar Cascades*. Dec. 2020. URL: https://docs.opencv.org/4.5.1/d2/d99/tutorial_js_face_detection.html.

[91] Shankar, B. G. *Face Detection with Haar Cascade*. Dec. 2020. URL: https://towardsdatascience.com/face-detection-with-haar-cascade-727f68dafd08.

[92] *Dev Board*. URL: https://coral.ai/products/dev-board.

[93] Situnayake, D. Build AI that works offline with Coral Dev Board, Edge TPU, and TensorFlow Lite. (Mar. 2019). URL: https://blog.tensorflow.org/2019/03/build-ai-that-works-offline-with-coral.html.

[94] *Get started with the Dev Board*. Accessed: 18.2.2021. URL: https://coral.ai/docs/dev-board/get-started.

[95] Jalušić, P. *Doing Machine Vision on Google Coral with OpenCV*. Jan. 2020. URL: https://krakensystems.co/blog/2020/doing-machine-vision-on-google-coral.

[96] *Welcome to Flask*. Accessed: 14.4.2021. URL: https://flask.palletsprojects.com/en/1.1.x/.

[97] *Jetson Nano Developer Kit*. Accessed: 12.4.2021. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[98] Rosebrock, A. *How to configure your NVIDIA Jetson Nano for Computer Vision and Deep Learning*. Mar. 2020. URL: https://www.pyimagesearch.com/2020/03/25/how-to-configure-your-nvidia-jetson-nano-for-computer-vision-and-deep-learning/.

[99] Mula, N. *Vision Processor for Machine Learning Applications*. Accessed: 2.6.2021. Nov. 2017. URL: https://www.eeweb.com/vision-processor-for-machine-learning-applications/.

[100] *SBC-S32V234: S32V Vision and Sensor Fusion Evaluation Board*. Accessed: 18.3.2021. URL: https://www.nxp.com/design/development-boards/automotive-development-platforms/s32v-mpu-platforms/s32v-vision-and-sensor-fusion-evaluation-board:SBC-S32V234.

[101] Rifenbark, S. *Yocto Project Reference Manual*. Accessed: 28.4.2021. 2018. URL: https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html#the-yocto-project-layer-model.

[102] *User Manual Linux S32V_BSP23.1*. Nov. 2020. URL: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32v2-vision-mpus-/s32v2-processors-for-vision-machine-learning-and-sensor-fusion:S32V234?tab=Design_Tools_Tab.

[103] *S32 Design Studio for Vision*. Accessed: 17.3.2021. URL: https://www.nxp.com/design/software/development-software/s32-design-studio-ide/s32-design-studio-for-vision:S32DS-VISION.

[104] Tensorflow for mobile & IoT: Python quickstart. (Mar. 2021). Acessed: 5.5.2021. URL: `https://www.tensorflow.org/lite/guide/python`.

[105] *Models Built for the Edge TPU*. Accessed: 14.4.2021. URL: `https://coral.ai/models/`.

[106] *Dlib Detailed API Listing*. Accessed: 24.4.2021. URL: `http://dlib.net/python/index.html#dlib.cnn_face_detection_model_v1`.