POLITECNICO DI TORINO

Master's Degree in ICT for Smart Societies

Master's Degree Thesis

Deep Learning on the Edge: a comparative analysis on computer vision for space applications

Supervisors

Prof. Enrico MAGLI

Dr. Mattia VARILE

Candidate

Francesco TOSETTI

December 2021

Summary

Artificial intelligence is currently one of the topics of most interest in computer science. In particular, artificial neural networks have found applications in many disciplines, like autonomous driving systems, medical diagnosis and many others. An enabling technology for this IT revolution is the available computing capacity provided by the Cloud. In some scenarios, however, execution on external servers is not possible, as in the case of some space missions. The only solution is to run the algorithms directly on the device that generates data. It is, therefore, necessary to migrate system intelligence from the Cloud to "the Edge". To do so, we must optimize the models to minimize the computation and memory required. This document will show the whole development process, starting from network training, through model optimization to deployment on embedded devices. For our tests, we employed the SPEED Dataset and trained three different neural networks for typical tasks in computer vision, such as object detection, semantic segmentation and keypoint detection. The models are optimized following standard techniques already present in the literature like quantization. The devices where we intend to run the algorithms represent the most popular off-the-shelf solutions currently on the market, such as Intel Neural Compute Stick 2, Google Coral Dev Board, Raspberry Pi, Nvidia Jetson Nano and Xilinx FPGA. We will analyze the accelerators equipped on each board and report a comparative analysis on various aspects of the development frameworks. Our results will mainly focus on accuracy, model size, and inference time. We will show how these solutions can effectively enable real-time machine learning applications at the edge. Although each device has its pros and cons, and we cannot say in general which one performs better, with the Coral Dev Board we get the best results. We can speed up about 80 times the same model running on Raspberry Pi 3B+, obtaining inference for semantic segmentation at more than 30 FPS.

Acknowledgements

First of all, I would like to express my sincere gratitude to the whole team of the Aiko company, for helping me during these months. In particular, my research would have been impossible without the aid and support of Mattia and Federico. Further, I would like to thank my academic supervisor, Prof. Magli, for the recommendation and the thoughtful comments on this dissertation. I am also thankful to the Politecnico di Torino and all its member's staff. To conclude, I cannot forget to thank my family, my girlfriend and my friends for all the unconditional support in these very intense academic years.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial intelligence

ASIC

Application specific integrated circuit

CNN

Convolutional neural networks

CPU

Central Processing Unit

DL

Deep learning

FPGA

Fiel Programmable Gate Array

FPU

Floating-Point Unit

GPU

Graphic Processing Unit

IoU

Intersection over Union

ML

Machine learning

MPSoC

Multiprocessor system on a chip

NN

Neural network

NPU

Neural Processing Unit

PTQ

Post training quantization

QAT

Quantization Aware Training

RISC

Reduced instruction set computer

SBC

Single Board Computer

SIMD

Single Instruction Multiple Data

SoC

System on a chip

SSD

Single shot detector

TPU

Tensor Processing Unit

VPU

Visual Processing Unit

Chapter 1

Introduction

1.1 Introduction

Artificial intelligence and deep learning in particular are being adopted in almost every industry nowadays. In 2020 this market yielded \$12.3 billions, but profits are estimated to rise to \$60.5 billions by 2025 [1]. Although the algorithms at the base of neural networks have been proposed in the 60s but practically ignored until the 80s [2], it is only in recent years that they have reached a wide use. This is due to the enormous availability of data, but above all, thanks to the powerful computing power provided by servers in data centers. The execution of deep learning models requires complex matrix calculations, and for this reason these models are often hosted by the Cloud. In this approach, intelligence is therefore centralized, but this solution is not optimal for all applications. The data is so exposed on the internet, it is necessary to have enough bandwidth to transmit information and the round trip time induces latency. A new direction that is taking the development of deep learning applications is the migration of intelligence from the Cloud to the Edge. The basic idea is to run neural networks directly where inputs are generated, deploying the "brain" of the system directly into embedded or mobile devices. However, the models developed over the years are incredibly cumbersome and rely on floating-point operations that are not well-supported by all devices. In order to have performances similar to those ensured by the Cloud, it is necessary to employ dedicated hardware, but also to optimize the models to make them lighter. It is also essential to minimize the operational cost of these networks to increase energy efficiency and make them suitable for battery powered devices. This opens the door to infinite new applications, such as ADAS systems or virtual assistants. The goal of this thesis, carried out in collaboration with AIKO [3], is to employ optimization techniques and ad-hoc hardware to execute three common tasks in computer vision such as Object Detection, Image Segmentation and Keypoint Detection on Edge devices. We will take as an example a possible application in the field of space missions, an area that could take incredible advantages from this revolution in deep learning, since the round-trip time for servers located on Earth becomes a bottleneck. The spacecraft becomes able to take the autonomous decisions on how to carry out their mission goals, even if not in contact with ground. We will evaluate the performance, especially in terms of speed for inference, for the main available off-the-shelf hardware solutions. Our study will also extend to general purpose devices such as Arm processors, GPUs and FPGAs.

1.2 Thesis outline

The structure of this document is the following:

- Chapter 2 summarize some state-of-the-art convolutional neural network and deep learning model optimization.
- Chapter 3 describes the employed dataset.
- Chapter 4 shows the training procedures for the employed neural networks.
- Chapter 5 presents the edge device where the models are going to run into.
- Chapter 6 summarizes the available frameworks for network optimization.
- Chapter 7 analyzes the different results and compare achieved performance.
- Chapter 8 is dedicated to final conclusions and future possible implementations.

Chapter 2

AI and Deep Learning

Artificial intelligence (AI) is one of the computer science fields of greatest interest in recent years. It allows programming electronic systems in such a way that they adopt characteristics that are typically associated with human intelligence, such as decision-making skills or images interpretation. A subcategory of particular interest in our days of AI is Deep Learning (DL), which is based on artificial neural networks. This discipline is a subfield of Machine Learning (ML) and consists of unsupervised learning of multiple hierarchical levels of data features. The features of higher level are derived from those of lower level. This is possible thanks to various layers of non-linear units cascaded to perform tasks of features extraction and transformation. The input of each layer is conditioned by the output of the previous one. Deep Learning is now used in almost every application, from health to space missions. The deep learning, bases its operation on the classification and selection of the most relevant data, to reach a conclusion, vaguely inspired by the human brain. It employs artificial neural networks, mathematical models consisting of interconnections of information.

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model composed of artificial neurons interconnected with each other. A typical problem in machine learning is to define a good model for a specific task. It is in fact difficult to write a computer program that "describes" the data. ML is limited to the definition of the model, use example data to fit this and then determine the decision rule. The neural networks, in the paradigm of DL, instead are able to learn the model and the decision rules in an autonomous way. This is done with mathematical models which implement a huge multidimensional non-linear function. These are organized into layers, each consisting of nodes or neurons. In the feedforward phase, these neurons receive the inputs, process them and pass the result to the following layer.

From a mathematical point of view, in a feedforward network, each neuron performs a scalar product between a vector of weights $\vec{w}$ and inputs. The result is passed to a non-linear function (activation function). In the simplest case, it is compared with a threshold, called bias. If the product is greater than the bias, then the output is 0, otherwise it is 1. The output of a layer is forwarded to the following one. To complete advanced tasks and to achieve good accuracy, it is necessary to implement many hidden layers. Each is formed by several neurons and in the case of dense connections, the number of parameters (weights and biases) become several million. The final structure of a deep neural network consists of:

1. An input layer, which takes in input the data to be processed and passes it to the network.
2. Multiple hidden layers, which process the data in a hierarchical manner, extracting features.
3. An output layer that takes decisions based on what is calculated in the hidden layers.

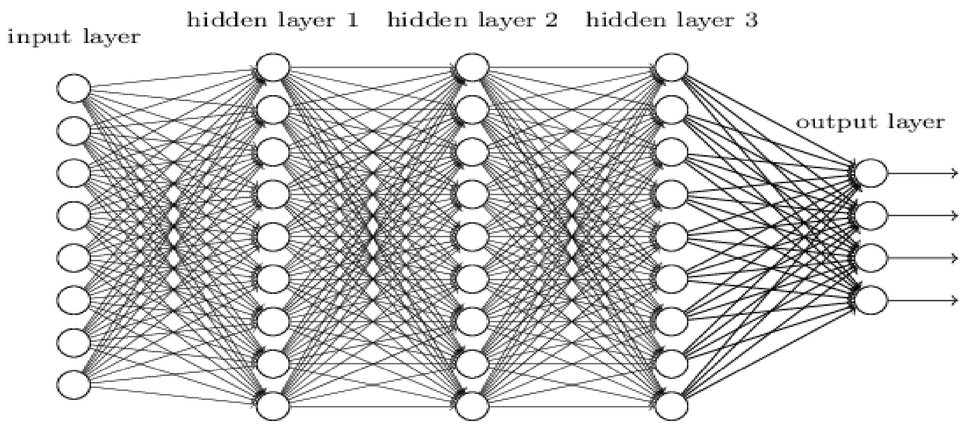


Figure 2.1: Fully connected deep Neural Network

Learning phase works by changing the weights of the vectors $\vec{w}_i$, being $\vec{w}_i$ the vector of the weights for the i-th layer. Changing $\vec{w}_i$ in fact also the output of the layer itself changes, and therefore the input of the following one. The weights are adjusted step by step in the so-called training phase, using labeled data. At each iteration, the loss function (e.g. the MSE) is evaluated by comparing the output with the expected value and $\vec{w}_i$ are changed to minimize it. This requires understanding how changing weights and biases affects cost. This happens thanks to the Error Back-propagation [2] algorithm that is the key for the implementation of these models. This training phase requires both huge computational capabilities and many available labelled data.

Once NN are trained, they can take in input data samples and return a prediction. What the end user is then interested in is the prediction of the networks. The purpose of this thesis is to optimize the inference process, which is simpler and faster than the training one. From now on therefore we will consider more this aspect of the networks.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNN) are a specialized type of neural network that uses convolution in place of general matrix multiplication in at least one of their hidden layers. The main application area of such networks is computer vision, especially for image classification, image segmentation, object detection and many other tasks. One of the challenges of computer vision problems is that images can be large, but we want a fast and accurate algorithm to work on that. The key factor behind CNNs is the application of Convolutional Layers, so called because of the mathematical function they are based on, the convolution. These layers convolve the input and pass the result to the following layer. One of the main concepts behind CNNs is the so-called Local Receptive Field. Whereas in fully connected NNs every input is connected to every hidden neuron, in CNNs each neuron receives input from only a restricted area of the previous layer, called the local receptive field. For images this is not a limitation since the content is usually "local" and there are few long-term correlations among pixels that are very distant. We slide then the local receptive field over the whole image. Furthermore, the connections from the local receptive field to each neuron have all the same weights, so all neurons in the first layer detect exactly the same feature, at different locations in the image. This reduces the probability of overfitting and the memory footprint of the network because a single bias and a single vector of weights are shared across all the local receptive fields. Furthermore, this exploits shift-invariance of image content. A convolutional layer act as a filter, convolving the input and passing the result to the following layer. In this way, they abstract images to a feature map. Another crucial element of CNNs is the Pooling layer. This layer, placed after the convolutional layer, simplifies the output information, performing a sort of down sampling. This reduces the size of outputs, speeding up the computation and making some of the features it detects more robust. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant. There are two types of Pooling layers, Max and Average Pooling. The first one returns the maximum value among the considered pool, while Average Pooling evaluates the average of all the values inside the considered image portion. A combination of several convolutions and pooling layers represents the feature extraction portion of the network. In the end, we can apply fully connected layers and a Softmax function for the classification task.

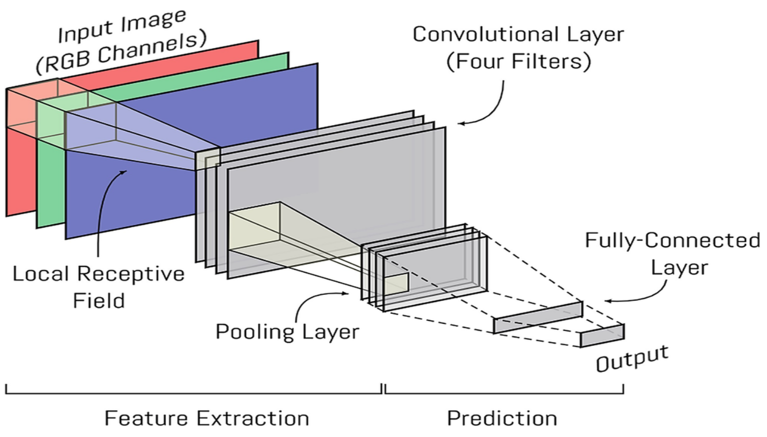


Figure 2.2: Convolutional Neural Network working schema [4]

2.3 Object detection network

Convolutional Neural Networks are commonly employed in object detection tasks. Object Detection consists of identifying and locating objects of a certain class within digital images. It has applications in many areas of computer vision, including face detections and object tracking. There are several approaches to the problem, but we will deal with the implementation using Deep Learning techniques, especially using CNNs. In the literature, there are two main families of NN implementations for this purpose: one-stage methods and two-stage methods. The first ones are oriented towards the speed of inference and include models such as YOLO (You Only Look Once)[5] or SSD (Single Shot Detector)[6]. The two-stage methods instead are centered on accuracy at the expense of a slower inference. These include R-CNNs (Regional-based Convolutional Neural Networks)[7] family models. Our main objective is to perform real-time inference on devices with reduced capacity, so we will consider mainly single-stage methods. An object detection network takes in input an image and returns to the output the coordinates of the bounding box containing the objects (localization) and the class of the identified object (classification). It is possible to represent the bounding boxes in different ways. We are using normalized coordinates, scaled with respect to the image size, reporting the bottom left and the top right corners of the box. A metric commonly used to evaluate the goodness of the algorithm is the Intersection over Union (IoU). It computes the size of the intersection and divides it by the union.

$$IoU = \frac{Area\ of\ overlap}{Area\ of\ union} \tag{2.1}$$

IoU is a measure of the overlap between two bounding boxes, the predicted box and the ground truth one. The possible values of IoU are in the range [0,1] and the higher is the IoU the better is the accuracy.

2.3.1 Single-Shot Detector

Single-Shot Detector (SSD)[6] is a particular architecture for object detection. This approach, discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. The main advantage of this method is that we need a single shot to identify multiple objects in the same image. Other networks instead need two shots, the first for the detection of the region and the second to classify the object contained. For this reason, SSD is decidedly faster and therefore suitable for real-time applications.

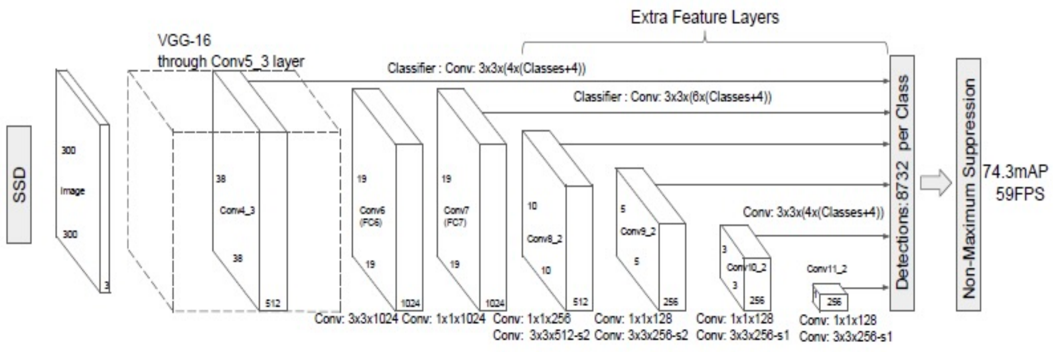


Figure 2.3: SSD architecture [6]

2.4 Semantic segmentation network

Semantic segmentation is the task of clustering parts of an image together that belong to the same object class. A semantic segmentation network then acts as a classifier for every pixel, instead of the entire image. Similarly to the object detection network, it receives an image as input, but it provides a mask as output. CNNs are commonly used in these tasks, and they are present in several applications such as autonomous vehicles, or medical image diagnostics. The training requires as input an image, and a mask that reports the class of each pixel represented. As a loss function we cannot use the pixel accuracy, consisting in evaluating the percentage of pixels that has been correctly classified, because of the class imbalance. In fact, the dataset we will use is not well-balanced and the background predominates. For this reason we used the Jaccard Index, which essentially evaluates the intersection over union as for the object detection, and it is computed as in Formula (2.2).

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{2.2}$$

2.4.1 U-Net architecture

U-Net is a Fully Convolutional Network (FCN) proposed in 2015 for biomedical image segmentation [8]. It is so-called because of the typical U-shaped structure, and the distinctive feature of this model is the use of an encoder/decoder structure that enables learning few latent variables. In summary, the first part (encoder) of the U-Net, the contraction path, down-samples the input image into a feature map, through pooling layers, extracting the key elements. Subsequently, in the expanding path, the network up-samples the feature map into an image, using the deconvolution levels (decoder).

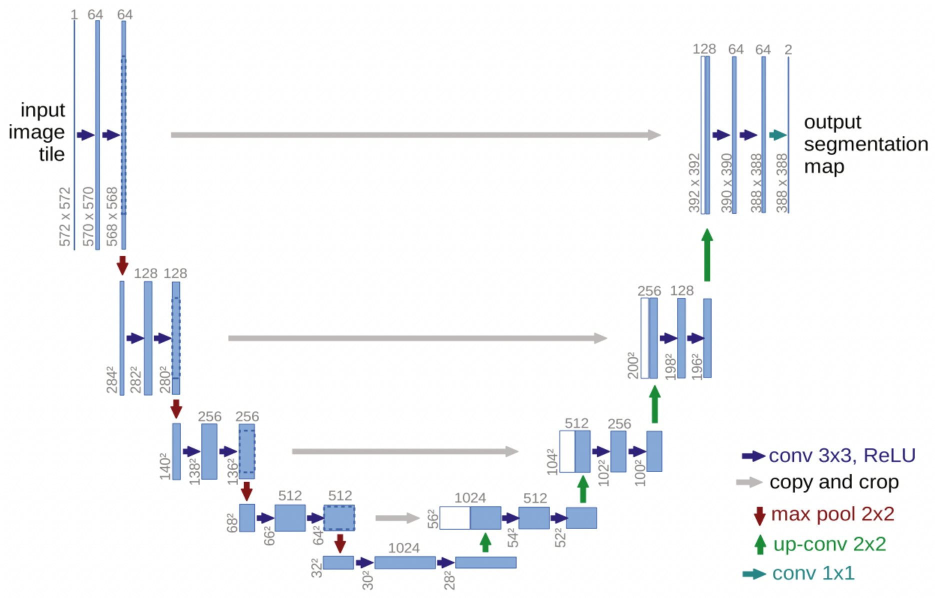


Figure 2.4: U-Net architecture [8]

2.5 Keypoint detection network

Keypoint detection consists of locating key object parts. Typical applications of this task include pose estimation or face detection. In the face detection scenario, for instance, key parts can be the corners of the mouth, eyes, eyebrows and so on. Keypoint detection can be used for on board estimation of the pose of some object (e.g. the relative position and altitude of a known non-operative satellite using monocular vision).

2.6 Deep learning for space

Artificial Intelligence and Deep Learning play an increasingly important role in the space domain. Examples [9] of areas where AI is heavily employed are satellite operations, to direct operations on a large constellation of satellites but also space missions. They are in fact used to analyze huge amounts of data coming from the missions. The data from some Mars rovers is being transmitted using AI, and these rovers have even been taught how to navigate by themselves [10]. Unfortunately, to be honest, still the ML lacks reliability and adaptability required for some applications. These aspects need to be improved before being widely employed in the space industry. However, the biggest space companies such as ESA and NASA are heavily investing in research for the advanced implementation of such algorithms. Currently, spacecrafts must communicate with the Earth to perform their work. Being able to make them fully autonomous, implementing artificial intelligence to take care of themselves, would be crucial to explore new parts of the solar system and to reduce the cost of missions. Other possible DL applications are automatic landing and intelligent decision taking. To do so, it is necessary to process a lot of data in a short time, directly on board. As we said before, however, neural networks can be very complicated and require large computing capacity. For this reason, it is fundamental to optimize them.

2.7 Intelligence at the Edge

The execution of deep learning algorithms is not a simple task, especially when applied to relatively complex data such as images and videos. In addition, the models must contain the weight and biases of a huge number of neurons, so the networks can be incredibly heavy. This implies that neural networks can only run on devices that have good computational capabilities and no particular memory limitations, such as workstations. Since not all users have powerful computers, Cloud-based applications are commonly employed to perform inference on complex networks. In this way, the "Intelligence" of the system resides in the Cloud. Any device with an internet connection can forward a request to the Cloud. In a reasonably short time, it can perform incredibly complex calculations, without needing memory to host the deep learning model. This sounds like a great solution, and indeed it is for certain scenarios. In other cases, however, this approach cannot be considered. To perform inference in the Cloud, an internet connection is needed, as said before. It may seem like something obvious nowadays, but sometimes this is not available. Another obstacle could be the device power consumption. Often in mobile devices, the biggest energy consumers are the antennas for data transmission. Some applications may need to transmit continuously and this would have a devastating impact on battery life. In addition, data, which could also be strictly personal, such as some medical parameters for instance, must necessarily travel over the Internet and thus become exposed to cyberattacks. Another crucial aspect concerns real-time applications. If in Cloud-based inferences, the speed of execution of models may, in some cases, become negligible, the latency in communication is not and represents an insuperable bottleneck. Even more so for space applications, with spacecraft that must transmit at hundreds of thousands of km, thus with delays of several minutes. Take as an example an autonomous driving system for a rover on Mars, this cannot ask the Cloud for every single operation. In these situations, the only possibility is to migrate the "intelligence" of the system, from the Cloud, directly into the Edge devices and for this we refer to "Intelligence at the Edge". This concept is becoming more and more popular and is used in numerous applications, even closer to our daily lives than what space missions can be. Classic examples are self-driving cars or even virtual assistants such as Apple's Siri or Amazon's Alexa that implement hybrid solutions, responding to what they can locally and querying the Cloud only for more complicated tasks. Furthermore, for applications with high duty cycles, the final cost in the long run will be much lower than forwarding every request to the Cloud. This migration is therefore an important step towards the deployment of deep learning in more and more applications. To make this feasible, we have two options. Scaling up the computational capacity of the devices, exploiting ad-hoc hardware architectures such as GPUs, FPGAs or ASIC, or scaling down the neural network's complexity to speed up their execution. It should be noted that only the inference process can be moved to embedded devices. The training process is too complex and inevitably requires large datasets, and therefore a lot of available memory, and huge computational capacity. Moreover, the network is trained only once and there are no particular reasons to perform it on the edge.

2.8 Network Optimization

As we have already mentioned, the models to be executed can be heavy and complicated. We therefore need to act on the models to enable the execution of neural networks on devices that have low computation and memory capabilities and often are battery-powered. This optimization process aims to shrink the model as much as possible. Generally, this reduction in complexity also leads to less accurate models. As an advantage, however, we are able to incredibly reduce the size of the network, the time required for the inference and the energy consumption. In the following chapters we will show as the drop in accuracy in some cases is not so critical, and therefore it does not compromise the success of the inference. In addition, some optimizations allow the use of specialized hardware for accelerated inference. There are particular hardware accelerators in fact, that we have used, and we will show later, as the Edge Tensor Processing Unit (TPU) or the Intel Visual Processing Unit (VPU) that can run inference extremely fast with models that have been correctly optimized. There are several optimization techniques, but the most popular are Quantization, Pruning and Weight Clustering. Although these techniques are mainly designed and used for embedded devices, they can now also be used on models stored on the Cloud. In fact, they reduce remarkably the dimension of the model and the operating cost of the network, decreasing therefore also the carbon footprint.

2.8.1 Neural Network Quantization

Quantization [11] refers to techniques for performing computations and storing tensors at lower bit widths than the original model data type. A quantized model executes some, or all of the operations, on tensors with integers rather than floating-point values. This allows for a more compact model representation and the use of high performance vectorized operations on many hardware platforms.

The networks, after the training process, are generally represented in FLOAT32 (Floating-point 32-bit). This means that each weight of the network is encoded with 32 bits, 1 bit for the sign, 8 for the exponent and 23 for the significand. The encoded value can be estimated as in Formula (2.3)

$$value = signifcand \cdot base^{exponent} \tag{2.3}$$

In this way, we can represent real numbers. Using instead integer data types, such as INT8 (Integer 8-bit) or UINT8 (Unsigned integer 8-bit), which use only 8 bits per value, we can represent only integer number in a range of $2^8 = 256$ values. So it is obvious that with 32-bit we can represent a larger number of possible values and with greater precision. Performing basic operations such as multiplication or addition is therefore totally different whether we adopt one or the other type.

Consequently, the execution speed also changes depending on the data format. We cannot say in general terms which is faster, but it strictly depends on the hardware architecture. For example, in modern desktop CPUs, floating-point operations are generally faster than integer ones. On the other hand, in simpler CPUs like those embedded on Single Board Computer (SBC), integer operations are generally faster. In fact, these devices lack adequate hardware support to efficiently perform floating-point operations. In other cases, floating-point operations may not be supported at all, such as for the Edge TPU.

The quantization process consists of mapping 32-bit weights (FLOAT32) to lower precision weights (e.g. INT8, FLOAT16, etc.). This conversion has mainly 3 advantages. First, it reduces the model by almost 4 times, if converted to INT8, or even more with smaller data types. It also speeds up inference on some architectures by reducing the cost of network execution, and as a final advantage it allows execution on devices that cannot perform floating-point operations. To understand how it works, we consider an example network layer, with values in the range $[-a, b]$, with a and b being any real number. A possible implementation consists of mapping all values in the range [−128,127] and round them to the closer integer. This results in only 256 possible values, thus representable with 8 bits. Obviously, in the rounding process we are losing information and therefore this action is not reversible. The quantization is an approximation process, but often the loss in accuracy is acceptable, sometimes even negligible.

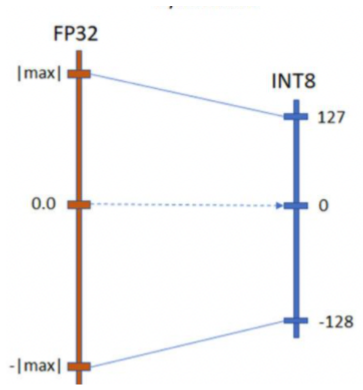


Figure 2.5: Quantization mapping example

Networks could not be directly trained to 8 bits for stochastic gradient descent, which requires high precision. In fact, during training, the values are adjusted continuously by applying many tiny nudges to the weights. These small increments typically need floating-point precision to work, otherwise one can fall into problems like vanishing gradient [12]. In practice, there are two main quantization techniques that are based on this theoretical principle: Quantization-Aware Training (QAT) and Post Training Quantization (PTQ).

2.8.2 Post Training Quantization

Post-Training Quantization (PTQ) is the simplest technique for performing quantization. It consists of manipulating the weights of the network, after it has been trained. It receives as input a network in FLOAT32 and returns the same model, but with the weights stored with fewer bits. We are able to reduce the size of the network by almost 75% with INT8 data type, also if we have approximation errors, which can be more or less significant depending on the application and the specific case. The main advantage of this technique is that the model does not require retraining. This is a huge advantage in all those situations where we lack the original dataset or computational capacity to train the networks. In addition, the training process sometimes takes a very long time. Moreover, this technique is the easiest to implement and there are several frameworks that can help us. We can convert already existing models without too much effort or pre-processing stages. There are several studies and applications on this methodology and, at the moment, it is definitely the most used for the network quantization. The PTQ in fact introduces an approximation, as we explained before, without caring much about the final result. The other technique, QAT, takes into account this loss already during training.

2.8.3 Quantization Aware Training

As we have seen, the PTQ can have losses, even significant, in accuracy. Another approach, definitely more complicated, requires considering already during the training of the neural networks the loss introduced in quantization. Ideally, if during the training we already consider that the weights must then be quantized, we can already manage in the loss function this approximation, mitigating the error. The result of this process will be a more accurate network, but, as it can be understood, the implementation is much more complicated. This technique is very interesting but less used than PTQ. In fact, to adopt this strategy we must necessarily have the original dataset, and we need to retrain the network from scratch. We cannot therefore work on pre-existing models, as we usually did in our tests. In the frameworks and tools we investigated, it is not always stable and fully supported. Moreover, a preliminary investigation we have performed shows that the results are not always so much better than PTQ.

2.8.4 Network Pruning

Network Pruning [13][14] is a technique to reduce the network complexity and overfitting. The idea is that among the many parameters in the network, some could be redundant, therefore they do not contribute significantly to the output. The approach is therefore to cut out nodes that do not play an important role. To do this, we must then classify the neurons in the network according to how much they contribute to the generation of the output, and remove the less important ones. The classification can be done following different metrics such as their mean activations, the number of times a neuron was not zero on some validation set, and other creative methods. As for the quantization, the accuracy decreases and the loss depends on the number of nodes we prune, the goodness of the metrics used to classify the neurons and the network itself.

The correct procedure for pruning requires retraining the network at each iteration. Therefore, first the model is trained normally, then the less significant connections are removed, and the network is fine-tuned. The loop of fine-tuning and pruning is performed until the desired results are achieved [15].

This technique was introduced as early as 1989 [16] and has since been widely studied to compress CNN models. In more recent studies [17], state-of-the-art CNNs have been pruned with no loss in accuracy. In a well-known 2016 paper [18], the number of connections of VGG-16 and AlexNet are reduced by 13x and 9x, respectively.

2.8.5 Weight Clustering

Weight clustering [19] reduces the size of the model by replacing similar weights in a layer, with the same value. These values are found by running a clustering algorithm over the model's trained weights. Like the other techniques we have shown above, the goal of weight clustering is to reduce the size of the model. It is similar to network pruning, but the benefit of compression is achieved by reducing the number of unique weights, whereas for pruning we set the weights below a certain threshold to zero. Just like pruning, we therefore have a loss in accuracy, which depends on the network and the results we want to achieve in terms of speed of inference and size of the model.

2.8.6 Other optimization techniques

Model optimization and compression techniques are certainly not limited to those presented so far. Being able to bring artificial intelligence algorithms to as many devices as possible is a challenge in which more and more researchers are participating. For this reason in literature [20] it is possible to find other examples of techniques, that at the cost of low accuracy, squeeze the model until it weighs a few KB. For this reason we talk about TinyML [21], a new branch of Machine Learning that aims to bring on microcontrollers, or "tiny" devices in general, algorithms that once could only be performed on powerful computers. To do this we must reduce models of the order of tens, if not hundreds of MBs in networks of a few hundred KBs, reducing the models of almost 100 times. In this scenario, we are

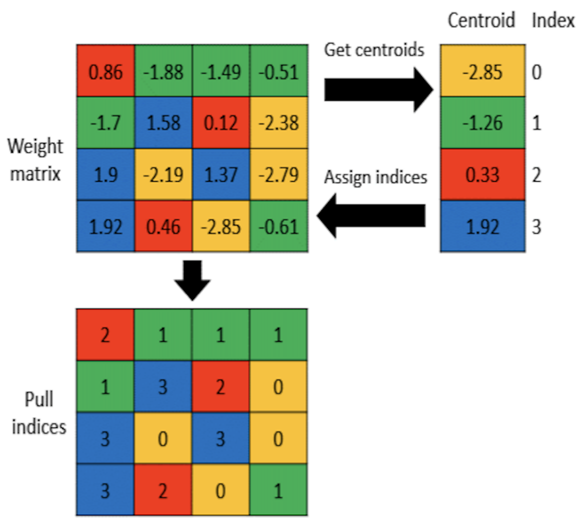


Figure 2.6: Weight Clustering working example

able to perform Machine Learning tasks on inexpensive devices, such as a few dollar microcontrollers might be. With such simplified models, we might be able to have devices with batteries that last even months. Since these devices are present in almost every aspect of daily life, their use is limited only by our imagination. They can already be used in industrial predictive maintenance [22], security devices, healthcare or agriculture [23] to name a few.

Chapter 3

Dataset generation

3.1 Dataset description

Training a deep neural network usually requires huge labelled image datasets. Collecting these data is very time-consuming and often complicated. For this, we relied on a public dataset, the Spacecraft Pose Estimation Dataset (SPEED) [24] released for a challenge, proposed in 2019 in collaboration with the European Space Agency (ESA). The main subject is the Tango spacecraft from the PRISMA [25] mission. The original Dataset contains a total of 15300 monocular images, derived from two sources. The first one consists of 15000 rendered images created with Augmented Reality software. The second source instead consists of real images of a mock-up spacecraft, acquired with a camera sensor, under optimal lighting conditions. These are only 300 and cover too few scenarios. In fact, the spacecraft is always positioned on a black background, easier to distinguish than other situations, such as with the Earth in the background. Moreover, the spacecraft is always positioned in similar poses and the light conditions are optimal. The images are therefore not enough explanatory of the problem, and to train the network only with real images would make the network not robust and lazy in more complex situations. The spacecraft renders instead make the situation much more realistic, representing the subject sometimes in front of the Earth, moving it in different poses and with various light situations. For this reason, both for training and for testing the algorithms that we will propose later, we consider only rendered images. Image's size is 1920x1200. SPEED is released under the CC BY-NC-SA 3.0 license [26], so it can be copied, modified and redistributed, but it is not allowed to use these data for commercial purposes. SPEED does not report also the labels of the images, so we had to label the dataset manually for each network. To do this, we employed specific free tools.

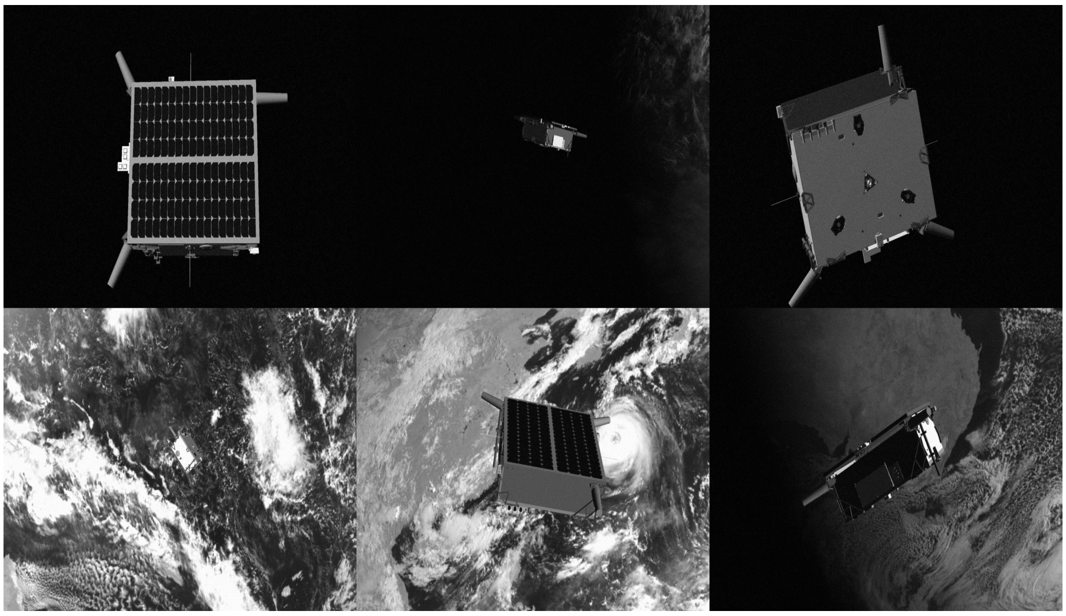


Figure 3.1: Example images from SPEED Dataset

3.1.1 Object detection labelling

To effectively train object detection CNN, we need to generate the ground truth bounding boxes. To do so, we used a utility function offered by the TensorFlow Object Detection API. Object Detection API is the open-source framework we used to train our network. Recalling that the main objective of this paper is not to provide a particularly accurate network, but to optimize it for devices with limited capacity, we decided to label a subset of only 80 images instead of the whole dataset, which would have taken a long time. The results in terms of accuracy will certainly be lower, but labelling and training processes are much faster. The same applies to the labelling of networks for Pose Estimation and Semantic Segmentation, which will be discussed later. These 80 images represent the largest number of possible situations. We considered the spacecraft at different distance from the camera, in many poses and both with and without the Earth as a background. The resulting labelled dataset is divided into 2 parts. The first one, containing 57 images, is used for training, while the remaining 23 are used for testing purposes. To represent the bounding boxes, it is necessary to use 4 values. For example, a possibility is to indicate the coordinates of the center of the box, the height and the width. There are several of them, but we decided to report the coordinates of the bottom left point and those of the top right point. We decided to use the normalized coordinates, evaluated as in Formula (3.1) so they do not require modification if we need to scale the images.

$$(X_{norm}, Y_{norm}) = (\frac{X}{W_{image}}, \frac{Y}{H_{image}}) \tag{3.1}$$

With H_{image} and W_{image} that are the height and width of the image respectively.

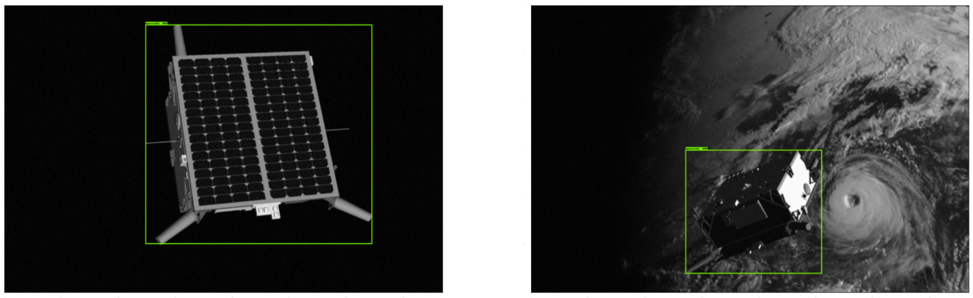


Figure 3.2: Object detection labelled images

3.1.2 Semantic segmentation labelling

The labelling process for the segmentation network is slower than for object detection. We no longer have to simply generate bounding boxes but masks. It is therefore necessary to manually classify every single pixel of the image, and there are several online tools available for this purpose. We used LabelBox [27], an online utility which is very simple and offers a free version, which obviously has some limitations but is still sufficient for our purpose. In this case, we used only 56 images, keeping approximately 70% of them for training (39 images) and the remaining 17 for testing. The possible classes to which each pixel can belong to are 2, one represents the background and the other the spacecraft. For this reason, we are talking about a binary segmentation.

3.1.3 Keypoint detection labelling

The image labelling we performed for keypoint detection was always done with LabelBox. The processed images are this time 60, 42 for training and 18 for testing. In this phase we must obviously first identify the points, then label them, specifying what point each of them represents. We considered 11 keypoints, 8 of which identify the vertices of the body of the Tango spacecraft, while the remaining 3 the tip of the RF antennas. Depending on the pose of the spacecraft, not all points are visible, so we need to indicate which ones are hidden. We labeled each keypoint, with the element it represents, and divided them into three classes. The first represents

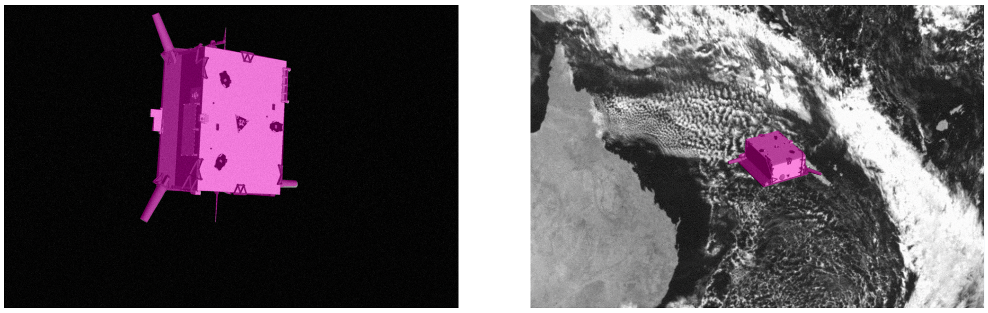


Figure 3.3: Segmentation labelled images example

the vertices of the front body of the spacecraft, the one with the solar panels, and therefore includes 4 points. The second one, composed by other 4 points, instead identifies the 4 vertices of the rear part of the spacecraft, while the last class is composed by 3 points that identify the antennas. In Figure 3.4, the first class is represented in red, the second in yellow and the third in blue.

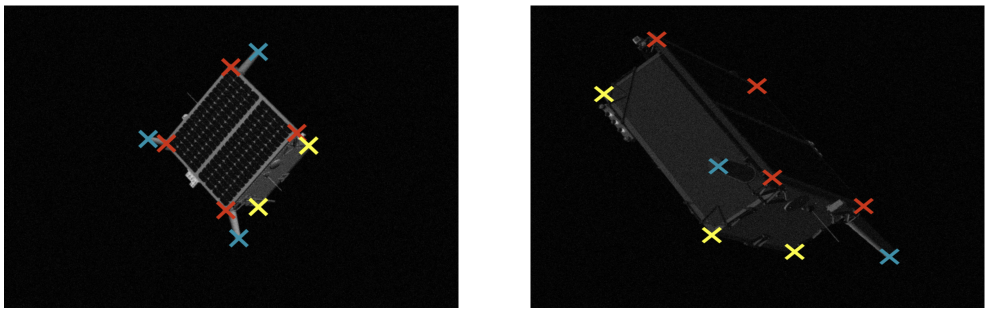


Figure 3.4: Keypoint Detection labelled images example

Once we have identified each point, we can then reconstruct the spacecraft and figure out its orientation and estimate its distance from the camera. Figure 3.5 shows a stylized 3D model of the spacecraft that we can reconstruct from our points, while in Figure 3.6 it is shown a montage of the wireframes model on a real image.

3.2 Data augmentation

Data Augmentation consists of techniques used to increase the diversity of the employed training set by applying random, but realistic, transformations. Applying these techniques we are able to increase the amount of data, adding to the original

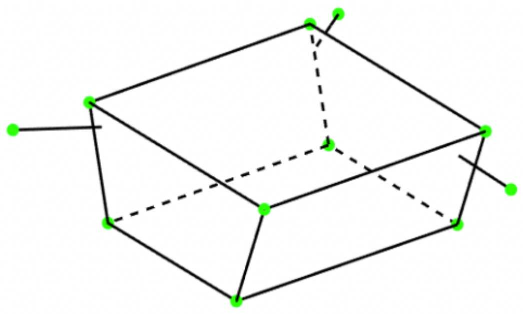


Figure 3.5: Reconstructed 3D model of the spacecraft [28]

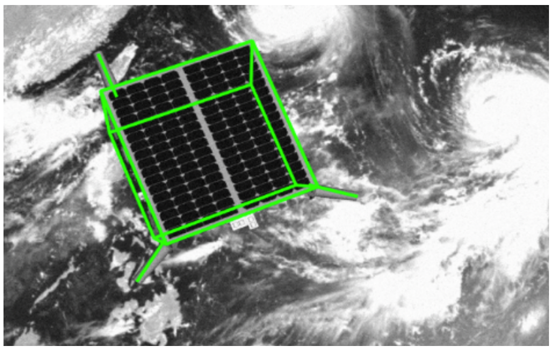


Figure 3.6: Montage of the model on actual image [28]

dataset slightly modified copies of existing images. It acts as a regularizer and helps reduce overfitting when training a machine learning model [29]. Faced with situations of data scarcity, data augmentation is very important. In our case, we increased the size of the dataset by almost 4 times. This then brought better values in accuracy, without struggling to label more data. In the end, we used 57 images for object detection which became 250 with data augmentation, while for segmentation we raised the number from 39 to 156 and for keypoint from 42 to 160. These techniques can be applied to different kind of data. In our case, that we are dealing with images, we can apply various types of transformation. For example, we can add Gaussian noise, rotate or crop images, modify the contrast, act on the color scale and many others. Thanks to the Albumentations library [30], everything can be done in Python. During the pre-processing stage, we only need to set the pipeline for the data transformation and, of course, provide the original images.

Chapter 4

Networks Training

Artificial neural networks require training to operate. This process is sometimes complicated and can take a long time. The networks in fact need to "learn" from some example, each of which is characterized by an input, as it can be an image in our case, and by an expected output that depends on the type of desired network. The training of a neural network from a given example is usually conducted by determining the difference between the predicted output of the network and a provided target output. This difference is basically the error that the network commits and can be measured with different metrics. The objective in this phase is to adjust the weights of the network in such a way as to minimize the error. The adjustment of the weights is done by applying small corrections at each iteration, using a specified training strategy. The main purpose of our work is not to train networks for innovative tasks or to improve the already existing ones, but to define basic models to be then optimized. For this reason, the training procedures that we will show will be quite simple and straightforward, without any particular tricks that could have improved performance. In this chapter, we will introduce the frameworks we used, and we will analyze in detail the training process of the three networks on which our work will be focused on. All the trained networks are CNN, and they are concerned with Object Detection, Semantic Segmentation and Keypoint Detection tasks. We will train the models using the datasets described in section 3. In each case, we developed the scripts for training in such a way that they can be reused by changing both the training dataset, but also the network architecture and hyperparameters.

4.1 Introduction and employed frameworks

The training of the networks was done using TensorFlow 2 [31]. TensorFlow (TF) is a leading software library for machine learning and artificial intelligence. Although it can be used for different tasks, it is mainly used for the development of deep neural networks. It was developed by Google, initially for internal use, but then in 2015 it was publicly released as a free open-source version. TF is available on 64-bit Linux, macOS and Windows platforms. Its computations are expressed as stateful data flow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays, which are referred to as tensors. The reason why we use this library is that there is a large community behind it that develops models for different tasks, like the ones we are interested in, and it is extremely flexible. Its architecture in fact allows the deployment of computation across a variety of platforms (CPUs, GPUs, TPUs) that we will use for inference and describe in the next chapters. Moreover, all frameworks that we will use later for model optimization are natively compatible with TF trained networks. An additional advantage of this library, is the TensorFlow Lite (TF Lite) extension, a software stack specifically designed for mobile deployment. TF Lite allows to convert models, natively designed for 64-bit desktop environment, to any supported mobile device, such as Android or iOS based devices but also 32-bit Linux based embedded systems. The limitation of TensorFlow Lite is that it cannot perform model training, which must be done on 64-bit compatible machines, but only inference.

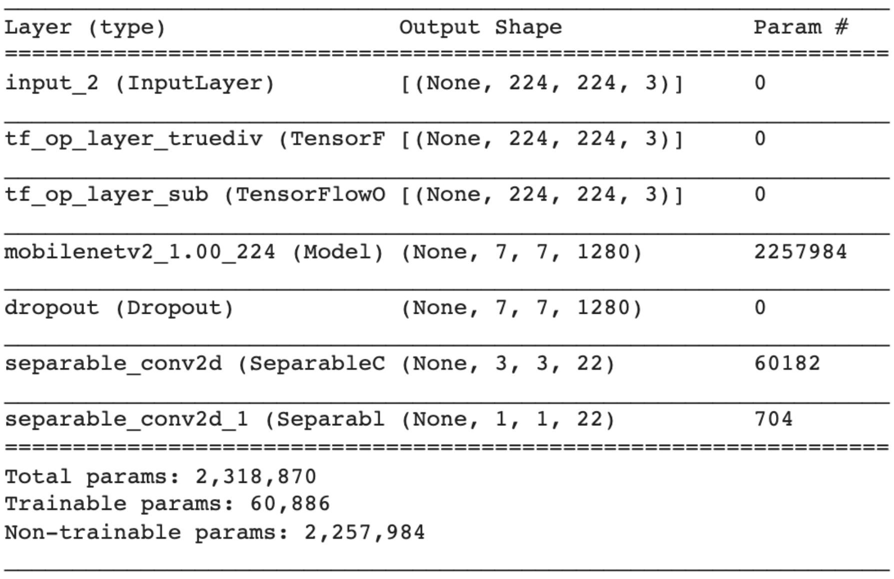
TF allows to efficiently execute low-level tensors on a wide range of architectures, to scale computation to GPUs clusters and to export programs to external runtimes such as servers or mobile devices. We can say that it acts as a back-end for model execution. For a human developer it is not always easy to define the model to be used and that is why we use Keras [32]. It is a deep learning API written in Python, running on top of TensorFlow. It was developed with a focus on enabling fast experimentation. It contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizers, and a set of tools to make working with image and text data easier to simplify the coding necessary for writing deep neural network code. Keras is therefore a more high-level library, easier to use, flexible and extremely powerful.

All of our models were trained using version 2.3 of TensorFlow and version 2.4 of Keras. The codes are entirely written in Python.

4.2 Networks fine-tuning

Training a network from scratch is a task that requires a lot of time, huge computational power and above all a very large dataset. Adjusting the weight of each layer of a network can require many calculations, and it is necessary to have a lot of data available to avoid overfitting. However, as we showed in Chapter 3, we did not have many images available. The only possible solution in these cases is Transfer

Learning [33]. This technique consists of taking features learned on one problem, and leveraging them on a new, similar one. For instance, features from a model that has learned to identify cars may be useful to start a model meant to identify spacecrafts. This is done taking the layers from a model already trained and freeze them in order to avoid modifying the values of the parameters during the training. At this point, it is possible to add more layers on top of the frozen ones. These will be the ones we will train and for this reason the number of trainable parameters is much lower as we have to deal only with the last layers. As an example, we can consider the Keypoint model shown in Figure 4.1, that we will discuss shortly. We apply two convolutional layers on top and only train the parameters of these. In this way we have to adjust only 60886 parameters out of more than 2 million. This makes the process much faster, requires fewer images and helps to avoid overfitting.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
tf_op_layer_truediv (TensorF [(None, 224, 224, 3)]     0
_________________________________________________________________
tf_op_layer_sub (TensorFlowO [(None, 224, 224, 3)]     0
_________________________________________________________________
mobilenetv2_1.00_224 (Model) (None, 7, 7, 1280)        2257984
_________________________________________________________________
dropout (Dropout)            (None, 7, 7, 1280)        0
_________________________________________________________________
separable_conv2d (SeparableC (None, 3, 3, 22)          60182
_________________________________________________________________
separable_conv2d_1 (Separabl (None, 1, 1, 22)          704
=================================================================
Total params: 2,318,870
Trainable params: 60,886
Non-trainable params: 2,257,984
_________________________________________________________________
```

Figure 4.1: Keypoint detection model summary

Alternatively, it is possible to choose not to freeze the last layers and only re-train those. The starting models to apply fine-tuning technique are trained very carefully on very populous Datasets. In our case, the networks we will use as starting point are all trained on the ImageNet Dataset [34] composed of more than 14 million images of different types of objects. It is often used for transfer learning [35] and indeed also our images fit well, providing good results.

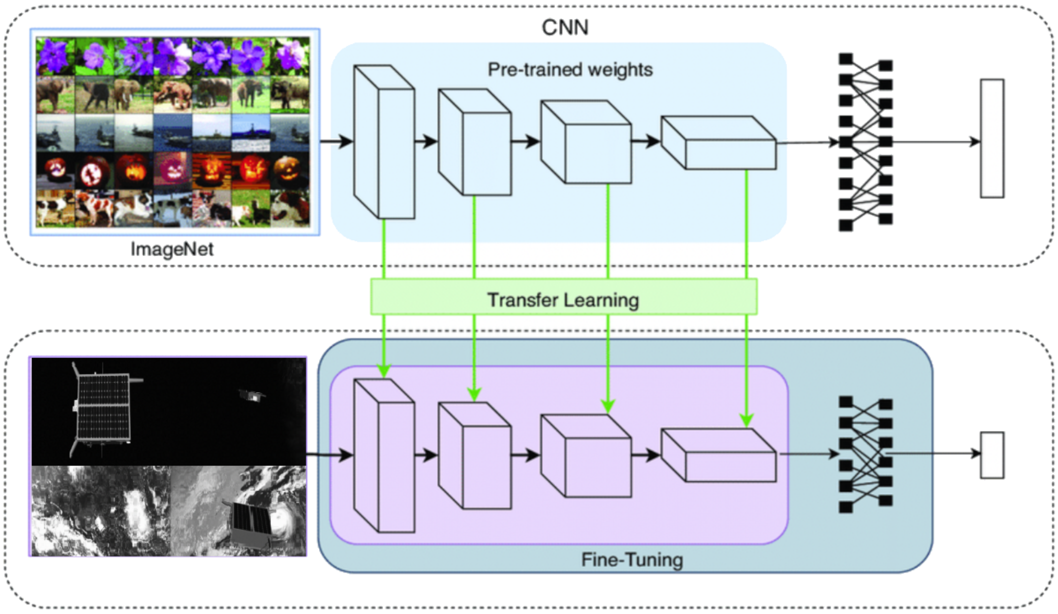


Figure 4.2: Transfer Learning example

4.3 Object detection training

The first network we have trained is a CNN that deals with Object Detection. It is a SSD (Single Shot Detector) network based on the MobileNet v2 architecture. It takes as input 640 x 640 images. The training dataset is composed by 57 images that pass through an augmentation process. To define this model, we relied on the TensorFlow Object Detection API, an open source framework built on top of TensorFlow that makes it easy to build, train, and deploy object detection models. Network training was done using the following parameters:

- Epoch = 1000
- Batch Size = 8
- Optimizer = Stochastic gradient descent (SDG)
- Learning Rate = 0.001
- Momentum = 0.9
- Classifier activation = Softmax

4.4 Semantic segmentation training

Regarding the segmentation network, instead we have relied on Segmentation Models [36], a Python library for Image Segmentation based on Keras and TensorFlow. We decided to use this library because it provides a high level API that allows to quickly create a model for segmentation, it also supports the U-Net architecture that we intend to use and an already pre-trained MobileNet v2 as backbone. It also provides an implementation of the Jaccard Loss needed to train and evaluate the performance of the model. In this case, we use 39 images for training, also going through a data augmentation process. The images are scaled to 224 x 224 resolution and because of this, when the spacecraft is far away, and is therefore represented by only a few pixels in the image, it becomes too blurry to be meaningful. In the data augmentation pipeline, we therefore cropped a 224 x 224 square from the original images near the spacecraft so that we did not have to compress the image and lose resolution. At inference time, something similar can be done, by first applying the object detection network and then cropping the predicted area from the network in the original image. This one is a CNN as well, more specifically a U-Net with MobileNet v2 as backbone architecture. Training parameters are the following:

- Epoch = 200
- Batch Size = 8
- Optimizer = Adam
- $\gamma = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$
- Loss Function = Jaccard loss
- Classifier activation = Sigmoid

4.5 Keypoint detection training

For the development of the keypoint detection network, we did not rely on any other high-level library. We have therefore used only Keras and TensorFlow. Here we take again as starting network the MobileNet v2, and we added at the end a Dropout layer and two layers of Depthwise separable 2D convolution. Our custom network is fully-convolutional, which makes it more parameter-friendly than the same version of the network having fully-connected dense layers. This solution is inspired by an example implementation provided by the Keras development team. Also in this case, the network was pre-trained on the ImageNet dataset.

The training dataset consists of 42 images, again 224 x 224 as for the segmentation. We intend to crop some images during data augmentation before passing them to the network to have higher resolution images of the spacecrafts similarly than before.

Training parameters now are:

- Epoch = 100
- Batch Size = 4
- Optimizer = Adam
- $\gamma = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$
- Loss Function = Mean Square Error (MSE)
- Classifier activation = Sigmoid

Working with the model, we realized that the images are too few to obtain satisfactory results. It turns out to be a task therefore more complicated than the others that we have tested. Modifying the augmentation pipeline and using more complex backbone networks does not give the desired results. To train networks of this type require thousands of images and for this reason the network fails to learn properly. We will use the model for optimization to test if it can be distributed on the edge and to test the gain in inference time. We will still evaluate the loss of accuracy, but it will not be as meaningful as for the other models.

Chapter 5

Edge devices

This chapter will describe the devices we are going to use to perform our tests. One of the main goals of this work is to demonstrate the ability to perform inference on limited capabilities devices. Examples could be mobile devices, or SBC, which may also be battery-powered. We are therefore talking about general purpose systems, which are not specifically designed to execute neural networks but are employed for a multitude of other tasks. We will also focus on systems based on architectures that parallelize operations, such as GPUs and FPGAs. These are widely used for various tasks due to their operational efficiency and adaptability. Moreover, we have also taken into account ASIC (Application Specific Integrated Circuit) systems that are designed to quickly and efficiently execute machine learning algorithms, such as the Edge TPU by Google. Each hardware architecture has its own pros and cons, and we cannot state in advance which one is better than the others. Moreover, since any platform process data in different way, also the time required for the prediction changes substantially, depending on the type of network we are dealing with. Generally CPUs work quite well with Recurrent Neural Networsk (RNN), GPUs are ideal for fully-connected networks while TPUs are ideal for CNNs and image processing in particular [37]. ASIC-based devices, being specifically designed for these purposes, can achieve better performance, both in terms of speed of inference and in terms of energy efficiency, a key factor for integrated battery-powered systems. If we had to design an artificial intelligence-based system from scratch, these devices would be the best choice. As we will show later, the results we obtain highlight that the networks we trained earlier are able to run at several frames per second. Thus, thanks to dedicated hardware, we are able to run real-time deep learning applications without querying the Cloud. This opens the door to infinite new applications in each domain. One field in particular that could reap incredible benefits is autonomous space missions, where we can perform complex real-time computer vision tasks and more, without being continuously connected to ground. Our analysis focuses on the major off-the-shelf solutions currently on the market. The devices we used are relatively inexpensive and represent some entry-level solutions in the world of AI accelerators. In fact, except for the FPGA, the devices we are discussing are easily affordable at around $100 or so. In detail, we are going to deal with:

1. Raspberry Pi 3B+
2. Rasberry Pi 4
3. Intel Neural Compute Stick 2
4. Google Coral Dev Board
5. Nvidia Jetson Nano
6. Xilinx UltraScale+ MPSoC

5.1 Generic embedded device - TensorFlow Lite runtime

As a baseline scenario for our analysis, we take a generic embedded device without recurring to any kind of hardware acceleration. The network then runs on a CPU. As example devices, we considered a Raspberry Pi 3B+ and a Raspberry Pi 4. They are two of the most popular SBCs available on the market, and they equip slightly different SoCs (System on a Chip). In this first analysis, we intend to run the networks using only the CPU, employing the TensorFlow Lite runtime without any kind of hardware acceleration. What will really make the difference in terms of execution speed is therefore the clock frequency of the processor and the floating-point execution features. This analysis help us to define a starting point and reasonably, the performance that we get in this way are the worst. Moreover, we use two devices and not only one to show how CPUs with different characteristics behave differently. Besides, we installed on the Pi 3B+ a 32-bit operating system (Ubuntu 18.04 32-bit) demonstrating therefore that the TensorFlow Lite runtime runs without compromise also on 32-bit systems. Raspberry Pi 4 runs instead the 64-bit version of the same OS.

5.2 Raspberry Pi: 3B+ and 4

The first device we looked at is a Raspberry Pi 3 Model B+ Rev 1.3. It is a Single Board Computer (SBC) released in 2016 and still commonly used in various applications. It is equipped with a quad-core ARM Cortex-A53 CPU that can deliver up to 1.4 GHz. It is based on 64-bit ARMv8 micro-architecture. It has 1

GB of LPDDR2 memory (900 MHz), shared with the integrated GPU, which is a Broadcom VideoCore IV. The running operating system is Ubuntu 18.04. We have installed a version of the operating system that makes the Raspberry as a device equipped with a 32-bit Armv7 CPU. This was done to have a wider range of devices and to evaluate the actual behavior. In fact, we have also the Raspberry Pi 4, based on Armv8 operating at 64-bit. This one is the successor model to the 3B+ and because of that it is more powerful in every way. It equips 4 GB of LPDDR4 SDRAM, a more powerful CPU and even the integrated GPU achieves higher performance. Some differences of the considered SoCs are shown in Table 5.1

	Raspberry Pi 3B+	Raspberry Pi 4
SoC	BCM2837B0	BCM2711
CPU Architecture	4-core Cortex-A72	4-core Cortex-A53
CPU Frequency	1.4 GHz	1.5 GHz
GPU Frequency	0.40 GHz	0.50 GHz
iGPU - FP32 [GFLOPS]*	20	32

*theoretical computing performance of the internal GPU in GFLOPS

Table 5.1: Raspberry Pi 3B+ and 4: SoC comparison

These are therefore both based on ARM, a RISC architecture commonly used on mobile and embedded systems and less on desktop environments. One of the advantages of this architecture is the energy efficiency with which it performs operations, which allows it to have small power consumption and keep the working temperature quite low.

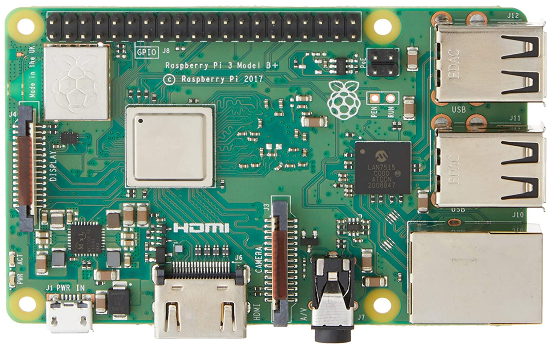


Figure 5.1: Raspberry Pi 3B+

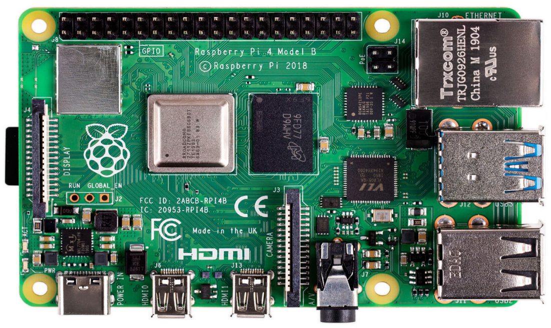


Figure 5.2: Raspberry Pi 4

The System on a Chip (SoC) of the Raspberry Pi, like many other embedded devices, cannot efficiently perform floating-point operations. In fact, although it features a Floating-Point Unit (FPU), these processors work best with integer data types. This is very common in mobile or embedded systems. Non-quantized 32-bit networks are therefore slowed down on this kind of devices. As we will show later, just by simply quantizing the model we can get almost 70% faster performance. Although these are therefore fairly inexpensive processors, they still have a hardware accelerator, called Neon. Also, if not specifically designed to perform accelerated inference, this technology can be used for this purpose as well. It is a combination of 64 and 128-bit SIMD (Single Instruction Multiple Data) instructions to accelerate and standardize the handling and processing of multimedia signals. It performs the same operation simultaneously for multiple data items. SIMD technology is crucial for performing vector operations, so also for inference with neural networks. What speeds up execution is the parallelization of operations.

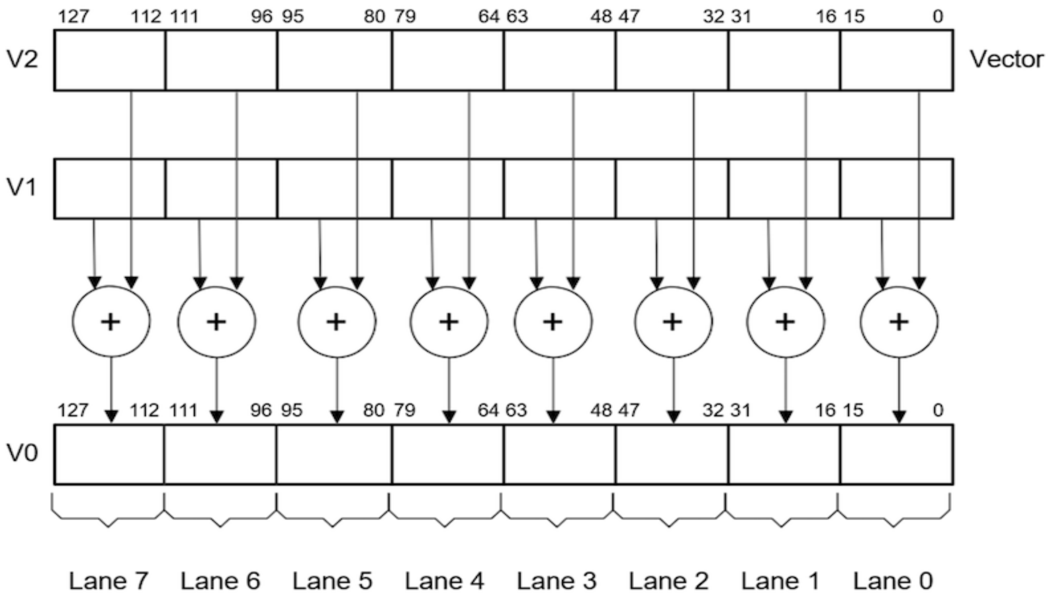


Figure 5.3: Example of addition instruction with Neon [38]

The Neon [38] unit is fully integrated into the CPU and shares the processor resources for integer operation, loop control, and caching. This significantly reduces the area and power cost compared to a hardware accelerator. It also uses a much simpler programming model, since the Neon unit uses the same address space as the application. The Neon registers contain vectors of elements of the same data type. The same element position in the input and output registers is referred to as a lane. Usually, each Neon instruction results in N operations occurring in parallel, where N is the number of lanes that the input vectors are divided into.

There cannot be a carry or overflow from one lane to another. The number of lanes in a Neon vector depends on the size of the vector and the data elements in the vector. Furthermore, some Neon instructions act on scalars together with vectors. It can handle up to 16 operations simultaneously. Figure 5.3 shows as the addition between two vectors is performed by the Neon accelerator. ARM is an extremely popular architecture, so being able to optimize neural networks for such platforms would have multiple benefits. The networks could be accelerated on millions of devices that are deployed every day. Additionally, although this architecture has historically been used primarily in mobile devices, in recent years it is also finding use in desktop environments such as in the latest Apple computers.

5.3 Coral Dev Board

The Coral Dev Board is a Single-Board-Computer (SBC) designed for the accelerated execution of Machine Learning algorithms. It is equipped with the Coral System-on-Module (SoM) that provides a fully-integrated system, including NXP's iMX 8M SoC, eMMC memory, LPDDR4 RAM, Wi-Fi, and Bluetooth, but its unique power comes from Google's Edge TPU coprocessor. More detailed technical specifications are shown in Table 5.2.

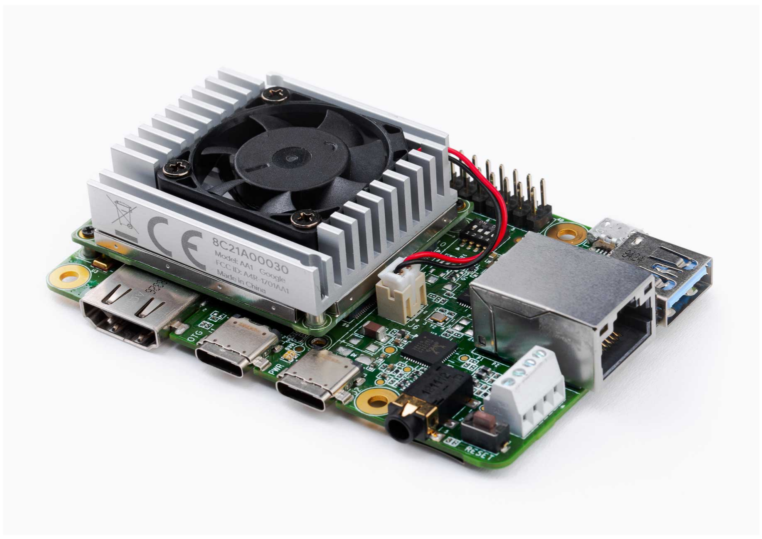


Figure 5.4: Coral Dev Board

The Edge TPU is a small ASIC designed by Google that provides high performance ML inferencing for low-power devices. It is capable of performing 4 trillion operations per second (TOPS), using 0.5 watts for each TOPS (2 TOPS per watt). These accelerators offer obviously lower computational capabilities than Cloud

TPUs hosted in Google data centers, which remain the best choice for training complex ML models. Edge TPU on the other hand is designed to run on small, low-power devices, and is primarily intended for model inferencing. In fact, it is only capable of accelerating forward-pass operations, which means it is primarily useful for performing predictions. Edge TPU is capable of executing deep feed-forward neural networks (DFF) such as CNN but with some limitations in the supported model architectures.

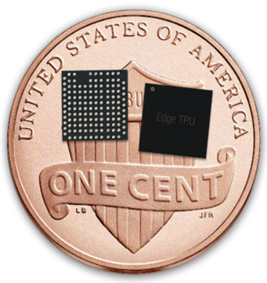


Figure 5.5: Edge TPU: size comparison with a dollar cent

In fact, to fully exploit the capabilities of the Coral Dev Board it is necessary that the model is quantized, the parameters must be in UINT8 or INT8. Moreover, tensors cannot have more than 3 dimensions and not all operations are fully supported [39]. The TPU edge cannot perform floating-point operations and therefore, if a non-quantized model is provided, the execution will be much slower since they will be delegated to the CPU. Coral dev Board is available from 2019 with a retail price of $129.

Coral Dev Board				
CPU	GPU	ML Accelerator	Memory	Size
NXP i.MX 8M SoC (quad Cortex-A53, Cortex-M4F)	Integrated GC7000 Lite Graphics	Google Edge TPU coprocessor: 4 TOPS (int8)	RAM: 1 GB LPDDR4 & Flash: 8 GB eMMC	88 mm x 60 mm x 24 mm

Table 5.2: Coral Dev Board Tech Specs

5.4 Intel Movidius Neural Compute Stick 2

Another off-the-shelf development kit created for running deep learning algorithms is the Intel Movidius Neural Compute Stick 2 (INCS2). This device launched in late 2018 is built on the Intel Movidius Myriad VPU featuring 16 programmable shave cores and a dedicated neural compute engine for hardware acceleration of deep neural network inferences. The main difference with other accelerators we have used is that it is a plug-in USB device. It can therefore be equipped with compatible devices either on x86_64 environments, or on Linux-based SBC. In our case, we are using the Raspberry Pi 3B+, described above. By coupling highly parallel programmable compute with workload-specific hardware acceleration in a unique architecture that minimizes data movement, Movidius VPUs achieve a balance of power efficiency and compute performance.

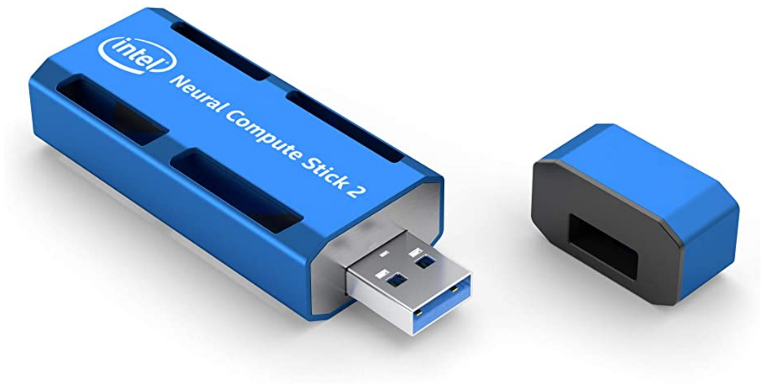


Figure 5.6: Intel Neural Compute Stick 2

This architecture provides ultra-low power consumption, that make it suitable for inference on Edge. The INCS2 is capable of delivering a total performance of over 4 TOPS, like Edge TPU. The models, which must necessarily be converted using Intel OpenVINO, which we will discuss in detail in the next chapters, cannot be quantized to integers. In fact, unlike the Coral, this board works only in floating-point. The models that we can provide are therefore only stored in FLOAT32 and FLOAT16 formats. This device is also particularly suitable for CNNs and is also employed in industrial applications and smart security cameras. Just like the Edge TPU presented before, it only speeds up the forward-propagation process and for this reason it cannot be used for training, but only during the inference phase. Not all operations are supported and, differently from the Coral Dev Board, those cannot be delegated to the CPU, so in case of unsupported operations, it is not possible to convert the model at all. This is also easily available for purchase and Intel's recommended customer price is $69.

5.5 NVIDIA Jetson Nano

The last deep learning specific solution we tested is the Jetson Nano Developer kit provided by NVIDIA. It is a small, powerful computer that lets us run multiple neural networks in parallel for applications like, object detection, segmentation, and speech processing. Similar to Coral, it is based on an embedded system-on-module (SoM) that includes an integrated 128-core Maxwell GPU, a 64-bit quad-core ARM A57 CPU, 4 GB of LPDDR4 memory, along with support for MIPI CSI-2 and PCIe Gen2 high-speed I/O. It is designed for deployment of deep learning algorithms in computer vision and runs on a 64-bit Linux-based operating system. The dimensions are similar to those of the Coral. It has a passive heat sink but is not equipped with a fan and for heavy workloads it can overheat. It supports only floating-point operations and for this reason, as in the case of INCS2, we will test only FLOAT32 and FLOAT16 models. In FLOAT16 it can deliver up to 472 GFLOPS with power consumption between 5 and 10 W. The accelerator is based on CUDA-X, which is built on top of CUDA and consists of a collection of libraries, tools, and technologies for parallel execution. Accelerated inferences are provided by the 128-core NVIDIA Maxwell architecture-based GPU. It is designed for Edge AI, so also this device makes efficiency one of its strengths and runs in as little as 5 Watts. This was released in March 2019 at a retail price of $99 and is also easily available for purchase.

5.5.1 GPU for Deep Learning inference

Graphic Processing Units (GPUs) provide much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. They run at lower frequencies than CPUs and can provide high energy efficiency. The performance difference between CPUs and GPUs exists because they are designed for different purposes. While CPUs are very good at executing sequences of operations (threads) and can do a few tens of them at once, GPUs are designed to execute thousands of threads in parallel. This allows them to mitigate slow single thread performance to achieve higher throughput. Something similar can be told for FPGAs, which bring similar benefits, as we will see in section 5.6.1, but offer much less programming flexibility. GPUs are designed in such a way that more transistors are dedicated to data processing, rather than data caching and flow control. In this way, we are able to perform more floating-point computations in parallel. A schematic example of the distribution of CPU and GPU chip resources is shown in figure 5.7.

Executing deep learning algorithms is a task that requires high parallelism, so we can get the most out of GPUs. This is true for the training phase, the most computational intensive task, but also for inference, especially on large batch sizes.

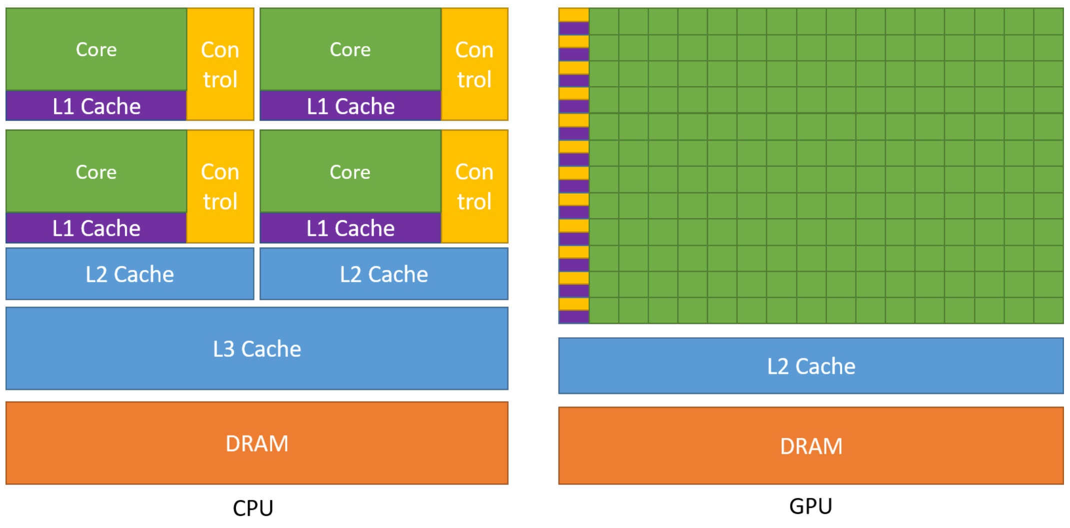


Figure 5.7: GPU devotes more transistors to data processing [40]

In our tests, we will rely on a GPU based on Compute Unified Device Architecture (CUDA), created by NVIDIA. CUDA gives developers access to a native instruction set for parallel computation of CUDA GPU elements. Using CUDA, the latest Nvidia GPUs effectively become open architectures like CPUs. This approach to troubleshooting is known as GPGPU (General Purpose GPU). The board used for testing features 128-CUDA cores, based on NVIDIA's Maxwell architecture. This microarchitecture is particularly suited for low-power applications.

Figure 5.8: Nvidia Jetson Nano

5.6 FPGAs - Xilinx Zynq Ultrascale+ MPSoC

The last device we considered is the Myir FZ3 Card [41]. This SBC is based on Xilinx Zynq UltraScale+ ZU3EG Multiprocessor system on a chip (MPSoC) [42], which features a 1.2 GHz quad-core ARM Cortex-A53 64-bit application processor, a 600MHz dual-core real-time ARM Cortex-R5 processor, a Mali400 embedded GPU and rich FPGA fabric. Besides, it integrates 4 GB DDR4, 8 GB eMMC, 32 MB QSPI Flash and 32 KB EEPROM as well as many peripherals. This enables accelerated execution of deep neural networks, and the manufacturer claims computational capabilities up to 1.2 TOPS. Like all the other boards we tested, also this one runs on a Linux distribution. The list price of this product is higher than the others and amounts to $399 [41].

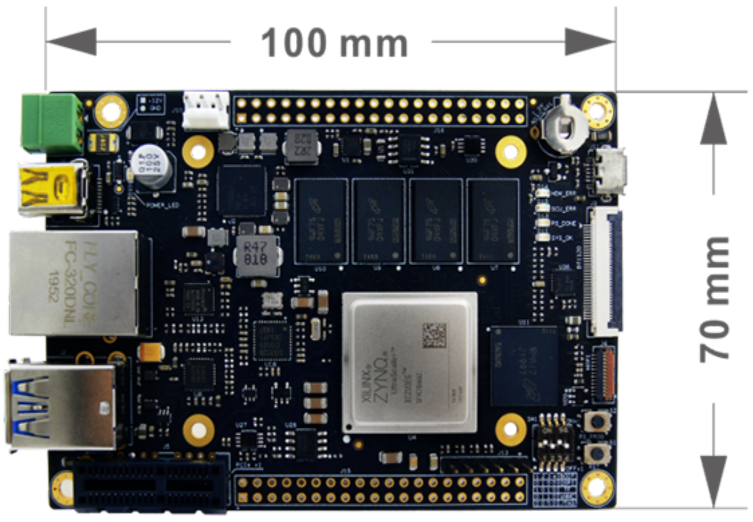


Figure 5.9: Myir FZ3 Card

5.6.1 FPGA for Deep Learning

The heart of this device is based on Field Programmable Gate Arrays (FPGA). These systems accelerate the execution of CNNs due to their ability to maximize parallel operations while maintaining low power consumption. In fact, they are programmable devices that mainly consist of a set of configurable logic blocks (CLBs) connected together through programmable interconnections. At the edge of this matrix there are input/output blocks, (IOB). The CLBs implement the logical functions, the set of interconnections connects them, while the IOBs interface the circuit with the outside. They have characteristics similar to ASICs, but are more flexible. Due to their flexibility and ability to handle computationally intensive tasks, they have been used as application accelerators. Moreover, thanks to their energy efficiency, they are excellent for both battery-powered devices and servers. Recently, they have also been used in the execution of neural networks [43]. In fact, FPGAs can create customized hardware circuits that are deeply pipelined and inherently multithreaded, which are very valuable for inference [44]. The benefits of these devices are not limited to flexibility. In fact, FPGAs support a full range of data types, including INT8 and FLOAT32.

Moreover, FPGAs can offer performance advantages over GPUs when the application demands low latency and low batch sizes. FPGA-based inference is invariant to batch size, while GPU-based inference is not [45]. This limitation of the GPU is inherent in its architecture. In fact, they consist of a large array of ALUs, which are considered single instruction multiple thread (SIMT), ideally similar to the SIMD we have seen in Intel's Neon technology. The workload is divided into thousands of threads in parallel, and numerous threads is required to prevent some ALUs from being idle. Only certain tasks can be efficiently mapped to such an architecture, however, and if we cannot take advantage of all the ALUs, energy efficiency decreases. For sequential, moderately parallel, or sparse workloads, the efficiency offered by the GPU can even be lower than that offered by a CPU [46]. Therefore, to optimize GPU-based inference, it is necessary to provide multiple images together. Buffering images clearly introduces latency, and therefore you cannot have both high throughput and low latency at the same time. FPGAs, on the other hand, can process input data as soon as it is available, having both high throughput and low latency simultaneously.

Furthermore, they can interface with several sensors and handle multiple input data streams smoothly, which is why they are used in systems such as ADAS or robotics. By accelerating data ingestion, FPGAs can speed the entire AI workflow, overcoming I/O bottlenecks, a common problem in AI applications. Unlike the other accelerators we have seen, FPGAs speed up the entire process. Since we do not have a specific application to address, we cannot take these aspects into account, but we will only evaluate the time required for inference. Considerations of this nature should still be done in the design phase to decide which is the most suitable device, so we will take into account this aspect for the final comparison.

Although this architecture seems very interesting for AI applications and is increasingly used, it also has limitations. First, they are not always easy to program, and sometimes require the developers to be familiar with hardware descriptive language (HDL). Furthermore, the implementation is even more complicated due to the lack of libraries, unlike the other devices shown above. It must be said, however, that there are solutions, such as the one we have adopted, that provide development environments that simplify the situation a bit. Moreover, they are definitely expensive. The cost of the board itself, in combination with the development costs, makes it a solution not feasible for small projects.

Now that we have introduced FPGAs and why they can be a key architecture for running Deep Learning, we can report the technical details of our device. It is based on the Xilinx Zynq Ultrascale+ MPSoC and the technical details are listed in Table 5.3 This kind of devices features both programmable logic (PL) and processing system (PS) on a single chip.

Ultrascale+ MPSoC ZU3EG	
Process (nm)	16
Memory (Mb)	7.6
System Logic Cells (k)	154
CLB Flip-Flops (k)	141
CLB LUTs (k)	71
DSP Slices	360

Table 5.3: Xilinx Ultrascale+ MPSoC ZU3EG overview

5.7 Devices comparison

The boards we have introduced represent some of the most popular off-the-shelf solutions on the market. The accelerators we have examined are excellent starting points for prototyping deep learning applications for embedded systems. Excluding the single Raspberry, in fact, they all have accelerators that can also be purchased separately and used on large-scale products. Although, excluding the FZ3 board, they have comparable costs and similar declared performance, the underlying architectures are different. The described accelerators are all particularly well suited for image processing and working with CNN. We did not investigate power consumption, but the values reported by the manufacturers are similar. We are going to evaluate a wide range of devices then. From SoCs without particular accelerators, to ASICs designed specifically for deep learning, passing through devices that are in between, such as FPGAs. We expect the best results from the Coral Dev Board or FPGAs. This is because GPUs do not work well with small batch sizes. Our inferences will in fact be on one image at a time. From the standpoint of power supply, temperatures at which they can operate, and size, they are all very similar, except for the INCS2 which looks like a USB flash drive. What can make the difference is the difficulty in setting up the boards and the support for the various models, which we will discuss in the next chapter. FPGAs in fact are generally more complex to program and require specific skills. Also, as we will show in Chapter 6, not all models are fully supported and this affects the performance. As we intend to investigate such architectures for use in space missions, it is important to evaluate other aspects as well. It is essential that the devices are resistant to the extreme conditions expected at the satellite board. A problem, however, beyond low Earth orbit is ionizing radiation. For this reason, Radiation hardening is generally performed. Radiation hardening [47] is the process of making electronic components and circuits resistant to damage or malfunction caused by high levels of ionizing radiation (particle radiation and high-energy electromagnetic radiation) especially for environments in outer space. None of the devices we use adopts these measures, but Xilinx produces also FPGAs with radiation hardening, then ready for aerospace use.

Chapter 6

Network Optimization

In the previous chapter, we introduced the devices on which we intend to run the algorithms. All these systems have obvious limitations compared to the huge data centers where Cloud-based inference can be performed on. We have to deal with two main constraints, the first one concerns the computational capabilities and the second one is about the available memory. Moreover, as we have seen, each device is based on deeply different architectures, with processing units that have different performance profiles and data precision they tend to operate best in. Although we used standard techniques already known in the literature that we discussed in Section 2.8, the pipeline for the optimization of neural networks changes, depending on the target device. In this chapter, we will show how the optimization processes take place, using the main frameworks currently available. In the case of proprietary platforms, they are released with free licenses directly from the manufacturer.

6.1 TensorFlow Lite optimization

As we will show later, many of the solutions we have studied work with TensorFlow Lite models. Converting the trained model from TF, to TF Lite is therefore an essential step. The models in TF Lite have multiple platform support, covering Android and iOS devices, microcontrollers and embedded Linux. A key point of this framework is that TensorFlow Lite binary is about 1 MB when all supported operators are linked (for 32-bit ARM builds), making it an ideal choice for devices with limited available memory. A TF Lite model is represented by an efficient portable format, known as Flat Buffers. This gives several advantages over the Protocol Buffers model format that the standard version of TensorFlow employs. In fact, it provides a smaller size and especially accelerated inference, since the model can be accessed directly without an extra parsing step. There are several ways to create a TensorFlow Lite model, but we always get them by converting a model from TensorFlow. To do so, we need to use the TF Lite converter. At this stage, we can already apply optimizations, such as quantization. Alternatively, it can also be done later. Since the model has already been trained, we are referring to Post Training Quantization.

TF Lite supports a number of TensorFlow operations used in common inference models. As they are processed by the TensorFlow Lite Optimizing Converter, those operations may be elided or fused, before the supported operations are mapped to their TensorFlow Lite counterparts [48]. Since the TensorFlow Lite built-in operator library only supports a limited number of TF operators, not every model is convertible. The best way to understand how to build a TF model that can be used with TensorFlow Lite is to carefully consider how operations are converted and optimized, along with the limitations imposed by this process. In addition, a number of TensorFlow operations can be processed by TensorFlow Lite even though they have no direct equivalent. This is the case for operations that can be simply removed from the graph, replaced by tensors or fused into more complex operations. To convert the models, we prepared a Python script, which can be used via command-line interface (CLI). It takes as input the TensorFlow 2 model to be converted and the desired data precision. If integer data types are chosen, it is necessary to provide also example images for calibration. Unlike constant tensors such as weights and biases, variable tensors such as model input, activations and model output cannot be calibrated unless we run a few inference cycles. As a result, the converter requires a representative dataset to calibrate them. In addition to quantization, other optimization processes can be performed with TensorFlow, thanks to the Model Optimizer library. It is possible to exploit both Weight Clustering and Pruning. These two techniques, however, require that the model be retrained, which is often not possible. Moreover, in this way the models are simply lighter and do not give a boost on the execution speed. They primarily aim to reduce the size of the model, losing points in accuracy. In our case, the accuracy is very important, and the models have already size that can be stored in each device we will work on.

6.2 Arm NN SDK

For the development of Arm platforms, we employed the Arm NN Software Developer Kit (SDK). It consists of a set of open-source Linux software tools that enables machine learning workloads on power-efficient devices. This inference engine provides a bridge between existing neural network frameworks and Arm Cortex-A CPUs, Arm Mali GPUs and Ethos NPUs. We have available only the Arm Cortex-A CPU, embedded on the Raspberry, so our study is limited only to this family. The development for GPUs and NPUs would be similar anyway. The importance of this tool is given by the fact that Arm architectures are widely used in many mobile devices, employed in all industries. Arm NN SDK utilizes the Compute Library, a collection of low-level machine learning function optimized for programmable cores, such as Cortex-A CPUs. The maintenance of Arm NN is presently guaranteed by Linaro Machine Intelligence Initiative, and it is now open source.

One of the main features of this SDK is support for major NN Frameworks such as TensorFlow, Caffe and ONNX. Thanks to a parser in fact, it is possible to provide the model trained with the framework that the user prefers. In our case, we can then perform inference by simply passing the TensorFlow Lite model, without further pre-processing stages. In many applications, this can be considered an

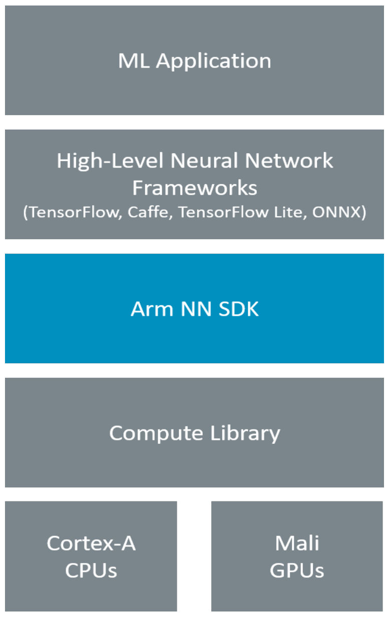

ML Application
High-Level Neural Network Frameworks
(TensorFlow, Caffe, TensorFlow Lite, ONNX)
Arm NN SDK
Compute Library
Cortex-A CPUs
Mali GPUs

Figure 6.1: Arm NN SDK overview

advantage. In fact, other procedures that we will show later, such as optimization for Intel, require that the network is first converted by a model optimizer. This obviously adds load to the development pipeline and requires that a new file is generated, which is only compatible with a limited set of devices. On the other hand, using a model parser, it is necessary to re-run the optimization process every time you load the network. This however turned out to be definitely fast, so this solution implemented by Arm can be considered a plus. To interface with Arm NN, we used the Python API PyArmNN which is built around the public Arm NN headers. PyArmNN does not implement any computational kernel, all operations are delegated to the Arm NN C++ library. Setting up the environment requires enough preliminary work, and building packages on limited devices like our Raspberry Pi takes several hours. On the other hand, the execution and development of inference scripts is simpler and more straightforward. In a few lines of code we were able to parse the network, specify the desired configurations, optimize the model and then run the inference. We have tested parsing TF Lite models, and often they are not compatible. This is a big limitation, especially if one wants to deploy ready-to-use models, which cannot be re-trained with other versions of TensorFlow, which use different operation sets. We are using Arm NN version 21.08 and PyArmNN version 26.0.0.

6.3 Coral optimization - PyCoral

To perform inference on Edge TPU, thus on our Coral Dev Board, we employ the PyCoral API [49]. The Edge TPU is compatible with the standard TensorFlow Lite API, so we can run a model on the Coral Dev Board by changing just a couple lines of code we used to run inferences on a generic device that supports TensorFlow Lite. By default, the TF Lite interpreter executes the model on the CPU, but that fails if the model is compiled for the Edge TPU because an Edge TPU model includes a custom operator. So it is needed to specify a delegate. Then, whenever the interpreter encounters the Edge TPU custom operator, it sends that operation to the accelerator. As such, inferencing on the Coral Dev Board requires only the TensorFlow Lite API. Unlike Arm NN, we then need to convert the model before deploying them to the board. For the model conversion, we use the Edge TPU Compiler, a command-line tool that compiles a TensorFlow Lite model into a file that is compatible with the Edge TPU. The compiler can be installed on any x86 64-bit Debian-based Linux system. Once compiled for the Coral Dev Board, the model cannot run on other devices. If we skip this step instead, the Edge TPU will not be used and the network will run only on the CPU.

	Object Detection	Segmentation	Keypoint detection
TFLite (int8)	4.370 MB	8.530 MB	2.819 MB
TFLite for Edge TPU	4.967 MB	8.762 MB	3.077 MB

Table 6.1: Models size comparison for Edge TPU

The dimensions of the models after conversion are shown in Table 6.1. This process therefore introduces a slight overhead, on the order of a few hundred KB.

The manufacturer, Google, also provides PyCoral, APIs in Python to assist with development. The PyCoral API is a small set of convenience functions that initialize the TensorFlow Lite Interpreter with the Edge TPU delegate and perform other inferencing tasks such as pre-process input tensors and post-process output tensors for common models.

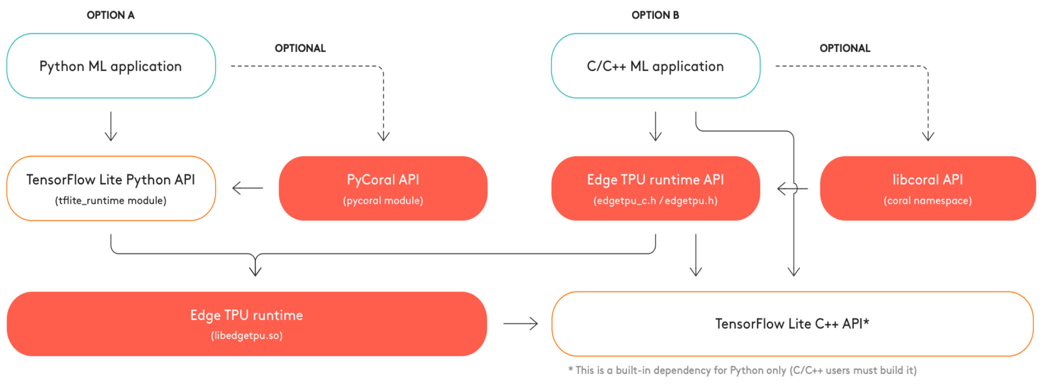


Figure 6.2: Coral Software stack

6.4 Intel OpenVINO

Similar to the process for Coral, deploying to Intel platforms requires model conversion first. The model conversion is done with OpenVINO toolkit, released by Intel. It allows optimizing the AI models for proprietary architectures. OpenVINO toolkit enables CNN-based deep learning inferences on the edge, and it supports heterogeneous execution on many Intel devices, including the Intel Neural Compute Stick 2 we are working with. Furthermore, it includes optimized calls for standard computer vision libraries, including OpenCV, that we are employing for data pre-processing and post-processing. To perform inference on Intel VPUs we must first pass the model to the Model Optimizer. The Model Optimizer is a cross-platform command-line tool that converts a trained neural network from its source framework, in our case it is TensorFlow, to an Intermediate Representation (IR) for use in inference operations. For model optimization, it is also worth mentioning the possibility of using OpenVINO Deep Learning Workbench, another tool that greatly simplifies the process by also providing a graphical user interface. We tried both options and although the DL Workbench proved to be very useful for walking the user through the process, the command-line tool is better for automating the process by running a single command.

Although model optimization has been presented as a simple and straightforward process, this is not always true. We also have to provide additional files, specify certain parameters, and select the right desired configurations. To give a clearer idea of what the complications can be, we can take the development of the object detection model we developed as an example. We had to specify two configuration files for the entire training and execution pipeline of the model. In addition, it is necessary to specify the input and output layers of the network and the size of the input tensors. It is also required to choose the desired accuracy of the model, always remembering that Intel Neural Compute Stick 2 does not have the hardware to perform accelerated inference on INT8 models.

Once we have the IR, in output from the Model Optimizer, we can then perform inference. The IR is composed of two files. The Model.bin file contains the values of the weights and biases, while the Model.xml file describes the topology of the neural network. The size of the IR obviously depends on the converted model and the adopted precision. The overhead compared to the TensorFlow Lite models is negligible. The final model sizes are reported in Table 6.2.

	Object Detection	Segmentation	Keypoint detection
TFLite (fp32)	12.109 MB	32.017 MB	9.124 MB
Intel IR (fp32)	13.260 MB	32.237 MB	9.257 MB
TFLite (fp16)	6.204 MB	16.053 MB	4.614 MB
Intel IR (fp16)	6.842 MB	16.250 MB	4.722 MB

Table 6.2: Models size comparison for Intel platforms

Once the models are converted, we are ready to deploy them. It is necessary to install the OpenVINO Inference engine on the host device to run the models on INCS2. The installation is simple. In addition, the models can be integrated into Python scripts thanks to the API provided by Intel.

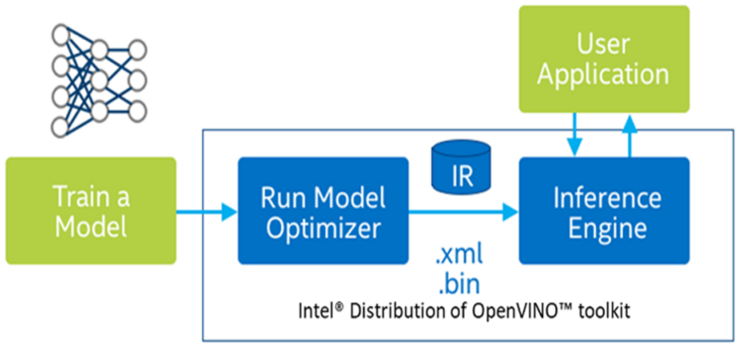


Figure 6.3: OpenVINO deployment overview

OpenVINO also provides an estimate on the theoretical workload to run each model. The results, are shown in Table 6.3. It returns the total number of floating-point operations required to infer a model (in FLOPS), the total number of trainable network parameters excluding custom constant and the number of memory activations. It considers both maximum and minimum theoretical memory consumption. The first one considers the amount of storage that is necessary for inference given that the memory is not reused, which means all internal feature maps are stored simultaneously. The second one instead refers to the amount of memory used by a network for inference when it is reused as much as possible. For both them, unit of measurements, is in number of activations.

	FLOP*	N. of weights	Min Memory	Max Memory
Obj. Det.	$6.89 \cdot 10^9$	$2.85 \cdot 10^6$	$19.7 \cdot 10^6$	$202.7 \cdot 10^6$
Segmentation	$6.88 \cdot 10^9$	$7.93 \cdot 10^6$	$4.0 \cdot 10^6$	$38.3 \cdot 10^6$
Keypoint det.	$0.62 \cdot 10^9$	$2.27 \cdot 10^6$	$2.43 \cdot 10^6$	$22.13 \cdot 10^6$

FLOP*: Floating-Point operations

Table 6.3: Models workload comparison for Intel platforms

The obtained results show that the number of operations required by object detection and semantic segmentation is practically the same. What substantially changes is the memory activation and the overall weight of the networks. Differently, the keypoint network has even an order of magnitude less in terms of operations required. This will positively affect the time needed for inference. Although this data refers to inference with OpenVINO, the networks behave similarly on the other frameworks. With this information, we can already anticipate which model will be faster. Keypoint detection in fact will take much less time than the other two networks. As second fastest we expect instead the semantic segmentation, that at equal FLOPS, must access fewer times to the memory. More in-depth results on inference times and accuracy will be discussed in Chapter 7.

6.5 NVIDIA TensorRT

NVIDIA TensorRT is an SDK for high-performance deep learning SDK based on NVIDIA GPUs. TensorRT is built on CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in GPUs. Accelerated execution is accomplished by a model optimizer. It provides six types of optimization, and they are summarized below:

1. Precision Calibration: It deals with quantization of models in FLOAT16 or INT8. The GPUs we use do not support integer operations, so we can only scale from FLOAT32 to FLOAT16.

2. Layer and Tensor fusion: Some recurring operations in inference graphs can be combined into a single operational step. These graph optimizations do

not change the underlying computation in the graph. Instead, they look to restructure the graph to perform the operations much faster and more efficiently. This reduces the overhead and cost of network execution.

3. Kernel auto-tuning: While optimizing models, there is some kernel specific optimization which can be performed during the process. This selects the best layers, algorithms, and optimal batch size based on the target GPU platform.

4. Multiple Stream Execution: The CUDA architecture is designed to execute data streams in parallel. This is the only way to achieve significant benefits over the CPU. This optimization deals with processing multiple input streams in parallel.

5. Dynamic Tensor Memory: To optimize memory usage, TensorRT allocates tensor space only when needed. This has a twofold advantage by avoiding allocation overhead. It reduces memory footprints and the network runs more efficiently.

6. Time Fusion: Optimizes recurrent neural networks over time steps with dynamically generated kernels.

Unlike the other accelerators we have presented so far, GPUs are commonly employed to execute deep learning algorithms even in data centers or professional workstations. We are talking about devices much more powerful than the Jetson Board we are using. The development workflow is anyway the same.

TensorRT (TRT) supports all major frameworks, therefore also TensorFlow. TensorFlow-TensorRT (TF-TRT) is an integration of TF and TRT that allows to deliver substantial performance gains on NVIDIA GPUs with minimal effort. When TF-TRT is enabled, in the first step, the trained model is parsed in order to partition the graph into TensorRT-supported subgraphs and unsupported sub-graphs. Then each TensorRT-supported subgraph is wrapped in a single special TensorFlow operation (TRTEngineOp). In the second step, for each TRTEngineOp node, an optimized TensorRT engine is built. The TensorRT-unsupported sub-graphs remain untouched and are handled by the TensorFlow runtime. Unlike the other boards we used, the deployment of the models could be done directly in TensorFlow, with no further conversions required before loading the models on the Jetson. Optimization could be done directly on the device. Although this seems to be the best solution due to its simplicity and flexibility, in the end we could not pursue this path. In our case, the Jetson was not converting the models correctly, and we looked for an alternative. In the official documentation, it is recommended to use the ONNX Parser. This way we can get rid of TF altogether and run a pure TensorRT version. TensorFlow is indeed a heavy framework and running it on a device with only 4 GB of RAM like our Jetson can lead to problems. The conversion from TensorFlow to

ONNX was done using an open source library: tf2onnx [50]. However, the models were not yet fully compatible, and therefore it was necessary to further work on them. For this, we had to further "surgery" the model with a tool provided by NVIDIA, called Graphsurgeon [51].

6.6 FPGA Optimization

Working with FPGAs is complex and requires experienced developers with different backgrounds. In fact, to properly set the programmable logic, it is necessary to know low-level programming languages. The production of large scale models is consequently longer. In this sense, these solutions are similar to ASICs, although they are more flexible. However, Xilinx has made available a number of software and frameworks to simplify the development process. Even being able to run neural networks would be a much simpler process, thanks to Vitis SDK. This kit also includes Vitis AI, a utility for designing ML applications. Although this framework could accelerate inference with the MPSoC UltraScale+, being able to run it is not easy. The FZ3 card we are using is not officially supported by Vitis in fact. It is a custom board and for this reason it is necessary to create the image from scratch. This is not straightforward and requires in-depth knowledge of FPGAs. Both hardware and software designs are required. This is why we were not able to take full advantage of Vitis AI. This framework, would have made the whole development much easier and more flexible. Anyway, since we intend to analyze the main available solutions, and Vitis AI is part of them, we will still evaluate the optimization process with this framework. To run the models on the FPGAs we will rely on PaddlePaddle [52] instead, another AI framework provided by Baidu.

6.6.1 Vitis AI

The Vitis™ AI [53] development environment is Xilinx's development platform for AI inference on Xilinx hardware platforms, including edge devices. It consists of optimized IP (Intellectual Property), tools, libraries, models, and example designs. Similarly to the other libraries used, this one is compatible with the main deep learning frameworks and therefore also with TensorFlow 2. It offers both a model quantizer and a compiler. They should support models in the SavedModel format that we are using. In practice, optimization with this format was giving problems, so we used models in the older Keras H5 format. In this way, we are able to both quantize and compile the models correctly. We used the latest version of Vitis AI 1.4.1. This runs on a Docker container on an x86_64 machine, based on Ubuntu 21.04. Xilinx also offers an optional AI optimizer that can prune a model by up to 90%. This tool is only available under a paid license, and we will not use it. This library also provides APIs in Python, both for quantization and model preparation but also for inference on FPGAs (Vitis AI Runtime). The workflow for model optimization goes through quantization first and then is compiled. This maps the AI model to a high-efficient instruction set and data flow. It also performs sophisticated optimizations such as layer fusion, instruction scheduling, and reuses on-chip memory as much as possible.

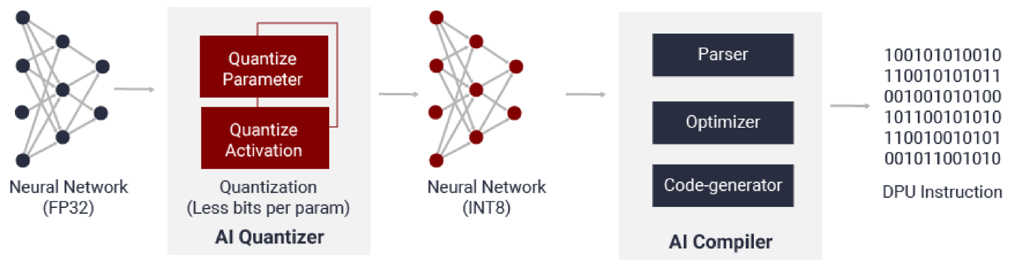


Figure 6.4: Vitis AI: Model Quantizer and Compiler workflow

As shown in Figure 6.4 the output of this processing step is a DPU serialized instruction. The Deep-Learning Processor Unit (DPU) is a programmable engine optimized for deep neural networks. It is a group of parameterizable IP cores pre-implemented on the hardware with no place and route required. It is designed to accelerate the computing workloads of deep learning inference algorithms widely adopted in various computer vision. An efficient tensor-level instruction set is designed to support and accelerate various popular convolutional neural networks, such as our SSD and MobileNet, among others. The DPU is scalable to fit various Xilinx devices, including the Zynq UltraScale+ MPSoCs we will deal with.

Optimizing the model for the Xilinx UltraScale+ gave no particular problems. The provided tools are compatible with all the models we used. What is undoubtedly more complex, however, is setting up the board for deployment. Unfortunately, Xilinx provides official support only for some boards, and ours MYiR FZ3 was not among them. For this reason we had to rely on another framework, much easier to set for our board, PaddleLite

6.6.2 Paddle

Paddle (PArallel Distributed Deep LEarning) is an easy-to-use, efficient, flexible and scalable deep learning platform, which was originally developed by Baidu. More in detail, we used the version for Edge, called PaddleLite. Thanks to the X2Paddle [54], we are able to convert our models. It does not support TensorFlow 2, so we have to use ONNX networks. The conversion is simple and happens smoothly for each model we tested. The board comes with preinstalled FPGA drivers that are immediately recognized by Paddle. The schematic of the software architecture of the FZ3 is described in Figure 6.5. One shortcoming of this framework is that examples are hard to find, and the documentation is mostly in Chinese.

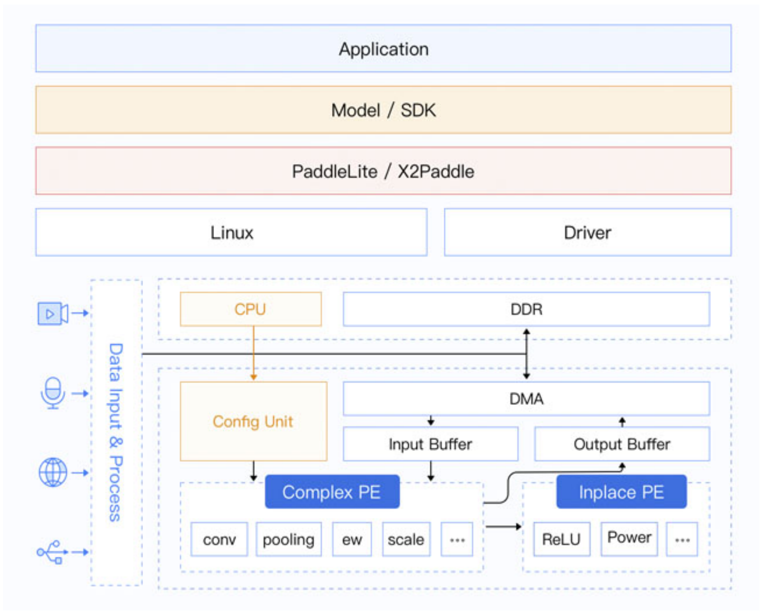


Figure 6.5: Diagram block of FZ3 Board inference process

Chapter 7

Performance comparison

In this chapter, we will look in detail at the performance of each model for any employed devices. The aspects that mainly interest us are the accuracy of the model and therefore a possible loss in the case of optimized and quantized models, the size of the network but above all the time required by inference. It is necessary to be able to speed up as much as possible the prediction of the output in order to enable real-time applications. Regarding the shown results, they were generated at two different times. We have tiny test datasets, so using only those images would not have provided very statistically significant results. In the case of segmentation, for example, we only have 16 images. We therefore decided to use an unlabeled dataset of 200 images to evaluate the timing for more statistically significant test results. Since they were not labelled, however, we could not use them to evaluate their accuracy. We therefore initially ran the algorithms on this dummy dataset. We then evaluated the accuracy on the few labeled images we had available. This will imply that the results in terms of accuracy will not reliable, in an absolute sense. We will mainly use them to evaluate the drop in accuracy and to demonstrate that in the various quantization and optimization processes, changes to the network structure and weights do not significantly affect the final outputs. Since each device and architecture behaves differently, we will review the performance of each and finally provide a quick comparison.

7.1 Raspberry Pi - Baseline performance

As a first analysis, we assess the performance of the Raspberry Pi 3B+. This device is undoubtedly the least performing among those we tested and also the cheapest among the various alternatives. However, being one of the most popular SBC in the market, we will take it as an example for comparison with other boards and in the next sections we will evaluate the performance advantages that we can achieve. We use the TensorFlow Lite runtime for the execution of neural networks. Although we used models that are not resource intensive and tailored specifically for mobile devices, inference times are too high. The results for each model are shown in Table 7.1

Precision	Object Detection	Segmentation	Keypoint detection
FLOAT32	4216.3 ms	2168.1 ms	447.3 ms
FLOAT16	4198.8 ms	2127.4 ms	436.0 ms
INT8	2710.1 ms	1606.3 ms	303.2 ms

Table 7.1: Inference time for each model (Raspberry Pi 3B+)

Using FLOAT32 models, we have average inference times of over than 4 seconds. This would make any application based on this system non-responsive. Using such a device is therefore not suitable in all those applications, such as intelligent vehicles or automated space missions. What we want to emphasize from these data is how the quantization of the model immediately brings benefits in terms of speed. The operations to be performed in fact are the same but are carried out in integers. A SoC like ours immediately obtains an advantage of approximately 1.5x. But since speed is nothing without precision, we must analyze the loss in accuracy. Results are shown in Table 7.2.

Precision	Object Detection	Segmentation	Keypoint detection
FLOAT32	71.26%	84.95%	0.079
FLOAT16	70.31%	84.93%	0.079
INT8	49.09%	84.49%	0.080

Table 7.2: Accuracy for each model (Raspberry Pi 3B+)

With the few images, we then manage to get working models. Accuracy is measured in terms of IoU for object detection and semantic segmentation (Jaccard Index), while for keypoint detection the MSE is evaluated. Test datasets are those described in Chapter 3. The semantic segmentation model performs better than others, we can even bring the accuracy to 88.43% by first cropping the images with object detection and then applying the segmentation. In that case, the inference pipeline is divided into two stages, thus delaying the inference time. We can do the same for keypoint detection. This is what is normally done for algorithms that aim to achieve maximum accuracy. In our case, the speed of inference has priority, so we decided to perform inference directly without first cropping the images in the proximity of the spacecraft. For object detection, IoU values above a certain threshold (in our case around 60%) should be interpreted as correctly classified. From our tests, outputs with an IoU higher than about 85% are perfectly detected. This is because we have to consider also human error in the labelling phase, especially for images where the subject is very distant. We decided not to use metrics such as precision and recall because of the limited dataset. Image 7.1 shows examples of correctly identified images, but with IoU values below 60%.

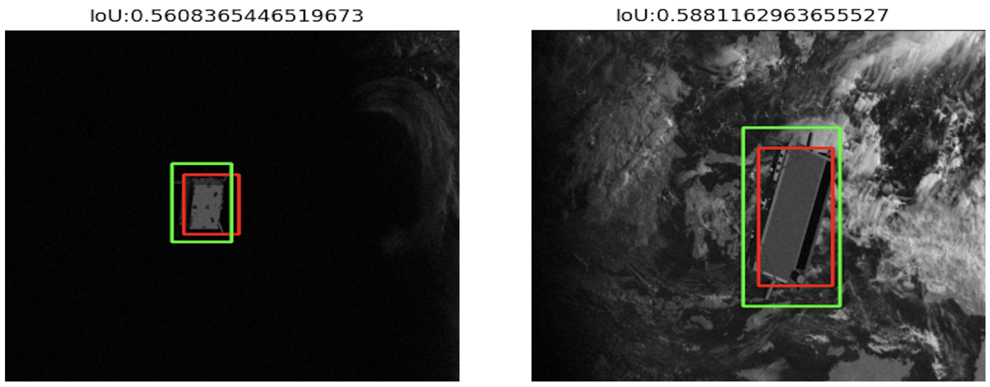


Figure 7.1: Correctly detected bounding boxes with low IoU values

From our tests, the loss in accuracy after quantization is not negligible. We lose 31% and 0.5% for object detection and semantic segmentation, respectively.

The segmentation network therefore reacts very well to quantization, even better than we expected. On the other hand, object detection instead decreases significantly the performances. Probably this could be avoided with more complex backbones or with more sophisticated training techniques. However, we are training the models with reduced datasets and extra layers to fit the network for our purpose. Furthermore, we are using a dataset quite complex to recognize in order to evaluate its accuracy. In fact, we wanted to test the network on a wide set of scenarios. There are images where the spacecraft is very distant and with the Earth as a background, which makes identification more complex. Regarding loss in accuracy, it is interesting to see that not all images are affected in the same way from the optimization process. Some even work better after quantization, others instead are no longer correctly identified. For this reason, evaluating the drop in accuracy on such a small dataset has no statistical value but should be considered only as an estimation. The predictions on some images are even more accurate than the original model. This is due to the non-linearity of the neural networks and the intrinsic model's noise.

Figure 7.2 reports the histogram of distribution of calculated IoU for the original and quantized models. As can be seen, the values are not simply shifted by 20%. The same statement can be made for all other models.

As far as keypoint detection is concerned, the results in terms of accuracy are not significant. The network overfits and for this reason the accuracy is very low.

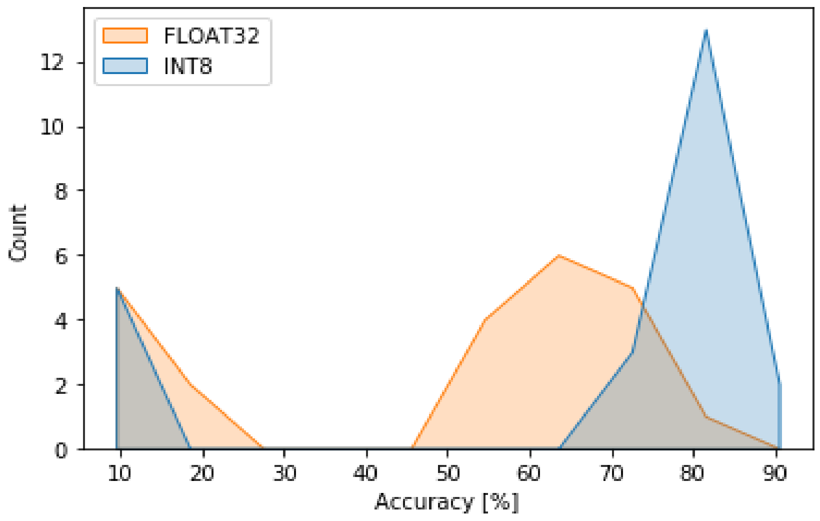


Figure 7.2: IoU Distribution

The quantized network should however recognize the points of the images used for the training. The dataset for the test in this case consists of both images used for training and new images. Although, from a theoretical point of view, these data do not make much sense, we can use them to evaluate if the inference result changes after optimization.

7.1.1 Raspberry Pi 3B+ vs Raspberry Pi 4

At this point, we can also compare the results just shown with those of the Raspberry Pi 4. The two devices then perform the exact same operations, since we use on both the TensorFlow Lite runtime. The outputs of the models are identical, so we will only compare the inference speed. The results are shown in Table 7.3. On both, we see better performance for integer-stored models, confirming that such devices fail to take full advantage with floating-point models. With a newer, more powerful SoC, we are therefore able to speed up inference by an average of more than 3 times.

7.2 Arm CPU - Neon accelerator

Unlike the other accelerators we have employed, Arm Neon is integrated into the board's CPU. One of the reasons why we are interested in Arm architecture is that it is equipped on most SBCs, and mobile devices in general, on the market. Being able to speed up the execution on these architectures would therefore allow

FLOAT32			
Board	Object Detection	Segmentation	Keypoint detection
RPi 3B+	4216.3 ms	2168.1 ms	447.3 ms
RPi 4	1200.9 ms	1352.9 ms	123.2 ms
INT8			
Board	Object Detection	Segmentation	Keypoint detection
RPi 3B+	2710.1 ms	1606.3 ms	303.2 ms
RPi 4	815.6 ms	587.3 ms	82.5 ms

Table 7.3: Raspberry Pi 3B+ vs Raspberry Pi 4: inference time

speeding up the inference without the need to rely on external hardware. Moreover, although our analysis is limited only to Cortex-A CPUs, the same procedure can be applied to other Arm devices. In fact, the code, always written in Python, can be easily adapted to Mali GPUs, integrated in several mobile devices, but also ETHOS NPUs, FPGAs, DSPs and accelerators specifically targeted for neural networks. We use for inference the same TF Lite models, described in the previous section. Although these optimizations could be ported to many devices, Arm NN has poorer support than the other frameworks we used. Unfortunately, we could not parse all models properly.

		Object Detection	Segmentation	Keypoint det.
FP32	Inference time	DNR**	264.2 ms	50.5 ms
	Accuracy	DNR	84.95%	0.079
	Speed gain*	DNR	8.2x	8.9
FP16	Inference time	DNR	265.0 ms	50.2 ms
	Accuracy	DNR	84.93%	0.079
	Speed gain*	DNR	8.1	8.7
INT8	Inference time	DNR	227.3 ms	38.4 ms
	Accuracy	DNR	84.49%	0.080
	Speed gain*	DNR	7x	7.9

*inference speed gained compared to baseline scenario (RPi3B+)
**DNR: Did Not Run

Table 7.4: Inference time and accuracy - Arm Neon (RPi4)

The version of Arm NN (21.08) we are using, has official support for TF 2.3. In practice, however, our models were not working properly. Therefore, we were forced to downgrade to version 2.1 of TensorFlow. Despite this, the SSD we use for Object Detection seems not to be compatible anyway. As far as the other two models are concerned, we get very good results. With the Armv8 based CPU of the Raspberry Pi 4 we obtain performances comparable to those of the INCS2, which we will show later. It is therefore possible to compare the performance achieved with Arm NN also with the performance achieved with the TFLite runtime on the same board. Results are reported in Table 7.5 Even with the quantized model, we can have faster performance, but losing accuracy. Results are reported in Table 7.4.

		Object Detection	Segmentation	Keypoint detection
FP32	TF Lite	DNR*	1352.9ms	123.5 ms
	Armv8 Neon	DNR	264.2 ms	50.5 ms
	Speed gain	DNR	5.1x	2.4x
FP16	TF Lite	DNR	1338.5 ms	123.0 ms
	Armv8 Neon	DNR	265.0 ms	50.2 ms
	Speed gain	DNR	5.1x	2.4x
INT8	TF Lite	DNR	587.29 ms	82.5 ms
	Armv8 Neon	DNR	227.3 ms	38.4 ms
	Speed gain	DNR	2.6x	2.1x

DNR*: Did Not Run

Table 7.5: Inference time on TF Lite runtime vs Arm Neon accelerator on the Raspberry Pi 4

Since also the Raspberry Pi 3B+ features the Neon architecture, we tested the models also in this device, and we compared the results with the SoC of the Pi 4. Similarly to the Pi 4 we were able to get 50% faster inference, but still almost 4 times slower.

7.3 Intel Myriad VPU

Inference on Intel Myriad achieves good performance, although lower than the other employed accelerators. The results obtained are summarized in table 7.6. The values in terms of inference speed are good, but we could have expected better. However, certain applications manage to provide good real-time performance, such as keypoint detection, which processes 19 FPS (Frames per Second). With our implementation of Object Detection, which is not a heavy model anyway, we cannot process even 2 images per second. We have no loss in accuracy. The outputs are almost identical to those provided by the original network. The Intel Neural

Compute Stick 2 only supports floating-point operations, so we cannot compare it with 8-bit quantized models. The algorithms are running on the same Raspberry Pi on which we evaluated the base cases. With this plug and play device we are able to get faster models of 7x, 19x and 9x for object detection, segmentation and keypoint detection respectively. For this device, reducing the accuracy of the data does not bring any benefit in terms of speed. The network behaves in exactly the same way. We only have a memory footprint reduction, and the model is lightened by almost half. This is a good compromise considering that the loss in accuracy is almost zero. This is only true for our models. Probably, evaluating the loss on more carefully fine-tuned models would have induced a loss in accuracy, however very low.

		Object Detection	Segmentation	Keypoint detection
FP32	Inference time	611.6 ms	113.7 ms	51.9 ms
	Accuracy	68.23 %	84.81%	0.079
	Accuracy loss*	4.2%	0.2%	0
FP16	Inference time	611.8 ms	113.5 ms	51.9 ms
	Accuracy	68.18%	84.78%	0.079
	Accuracy loss*	3%	0.2%	0

*Accuracy loss compared to the baseline model

Table 7.6: Inference time and accuracy - INCS2

The speedup that this device gives is not as extreme as that of the accelerators that we will see later on. An important aspect, however, concerns the support of floating point operations, something not guaranteed by devices such as the Coral Dev Board. This is a pro and a con, and strictly depends on the results to be achieved, the neural network to be executed and the employed device. In fact, thanks to this data precision we have no loss in accuracy, but on the other hand, neither reduction in memory footprint. In devices with no memory limitations, such as those we are analyzing actually, this can be a good compromise, since we accelerate the execution without losing accuracy. The conversion and optimization process is also well-supported. All operations our networks are based on are fully recognized by the Model Optimizer and model compilation runs smoothly. For completeness, we also tested model quantization with OpenVINO and everything went well.

7.4 Edge TPU

For inference, we used the TensorFlow Lite Python API and Edge TPU Runtime. It is sufficient to modify a couple of lines of code for inference with respect to the script for TFLite. The Edge TPU runtime library has to be loaded as delegate during the interpreter definition. The results achieved on Coral Dev Board are the best among the devices we tested. Even the compilation of the models is the easiest one. It is in fact sufficient to provide the model to the compiler, and it will return as output a new compatible one. This will be optimized for Edge TPU and therefore can no longer run on other runtimes. Moreover, if some operation is not supported by Edge TPU, it is simply executed by the CPU without compromising the correct execution. In other accelerators, if some model is not convertible, it cannot be used at all. In these situations, it will be necessary to act directly on the models, modifying the inference graph or using a different set of operations. Our models are not all fully compatible. In fact, the Object Detection is not entirely delegated to the TPU. Its execution will be significantly slower than the other models. This probably depends on the fact that we have used other libraries on top of TensorFlow that use operations that are not properly converted. Regarding the keypoint detection and semantic segmentation networks instead, we have impressive gains. All operators are mapped to the Edge TPU correctly.

The final inference times are shown in the Table 7.7.

	Object Detection	Segmentation	Keypoint detection
Inference time	391.4 ms	31.17 ms	7.35 ms
Accuracy	49.09%	84.49%	0.079%

Table 7.7: Accuracy and inference time with Edge TPU

We are able to increase the execution of MobileNet v2 for keypoint detection to 136 FPS. Refresh rates like this are really high for many applications. Many cameras cannot deliver so many images per second, and therefore the input devices become the real bottleneck. More complex models, such as the U-Net for segmentation, clearly require more time, but we are still able to reach 32 FPS. These are still sufficient for most scenarios. Object detection instead, as we expected, reaches good results but not comparable with the other models. Compared to the floating-point baseline scenario on Raspberry 3B+ we are still able to speed up the model by 10x, while for segmentation and keypoint we have an increase of 79x and 61x respectively. We have no accuracy loss compared to the integer model. However, we had to quantize the model since the TPU edge only works with INT8. The model was quantized and converted using TensorFlow Lite. The cost of achieving such performance therefore lies in the quantization-induced loss. We have non-negligible losses for the object detection, since we had to deal with several compromises in the training phase. More complex models, perhaps exploiting the Quantization Aware Training, or with more images available are able to obtain the same advantages in terms of speed, but with a reduced drop in accuracy.

7.5 NVIDIA GPU

Even with NVIDIA GPUs, the performance is significantly improved. The process of model conversion was strangely more tricky than for other boards. Unlike the other devices, we do not have to use the TensorFlow Lite runtime, which is very lightweight, but we must rely on TF. The memory of the GPU got saturated quickly. The RAM in fact is shared by CPU and GPU, and loading a heavy library like TensorFlow caused problems. Probably this solution is designed for more powerful products, with more memory available. Our Jetson has only 4 GB available. For this model, it is recommended to run the pure TensorRT version. So we were able to run our models, but one more step is needed. Since this library is not compatible with TensorFlow models, it is necessary to convert them to ONNX models. This is done with an open source tool, called tf2onnx [50]. The conversion is very simple and intuitive, but at the end, the model is subjected to several steps. Not all conversions go smoothly, and not all operation sets used by ONNX are correctly read by the parser in TensorRT. Without knowing the model or how each framework implements a specific operation, it is hard to say if it will be correctly converted. For this reason, we tried different versions of TensorFlow and different operation sets for ONNX. Despite these precautions, in some cases ONNX models were still not loaded. Further action was needed. This was done with a tool provided by Nvidia called onnx-graphsurgeon. It allows to manually modify ONNX networks to make them TensorRT compliant.

Going through ONNX is definitely worth the effort. The results we are able to obtain are shown in the Table 7.8. Although the inferences are slower than on the Coral Dev Board, we must specify that the models here are floating-point. This affects the final accuracy of the model. Moreover, the Edge TPU used on Coral is an ASIC and therefore cannot be dedicated to other purposes while the Jetson GPU is a general purpose device.

We have discarded a warm-up period of about 500 ms. This is because the first inferences are slower. This happens for all boards, but more in particular for the Jetson. In fact, in the first iterations, it is necessary to activate the GPU and load the model in memory. After the very first, inference times are stable on all boards, with deviations in the order of a few ms. Figure 7.3 shows the trend of inference times for the segmentation network on the test dataset. After the first two slower inference times, the times settle down and remain stable.

Precision	Object Detection	Segmentation	Keypoint detection
FP32 (TF)	237.6 ms	109.0 ms	25.8 ms
FP16 (TF)	237.0 ms	108.8 ms	25.5 ms
FP32 (TRT)	182.1 ms	78.2 ms	13.1 ms
FP16 (TRT)	159.3 ms	64.6 ms	11.2 ms

Table 7.8: Inference time on Jetson Nano

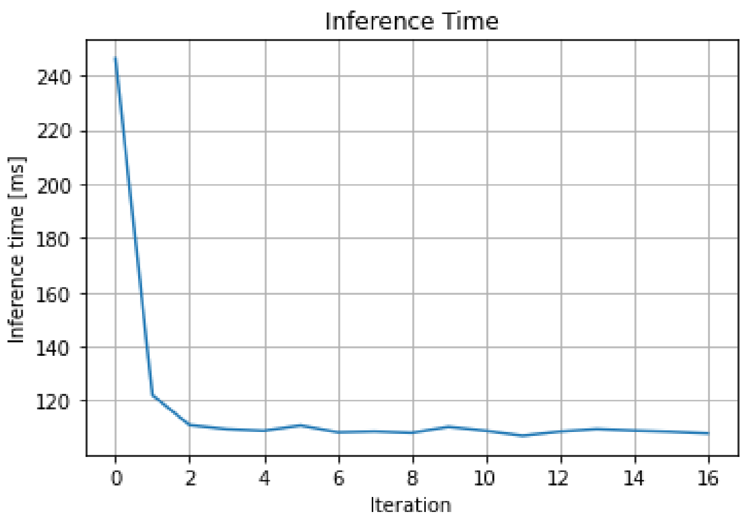


Figure 7.3: Warm-up period during inference time

Unlike the other devices we used, this board executes the algorithms very fast even without any kind of optimization. In fact, TensorFlow takes advantage of the GPU by parallelizing the operational flow. The boost in performance is given by the TensorRT with the optimizations described in the paragraph 6.5.

7.6 Xilinx UltraScale+ MPSoC

Although our models were correctly converted, they did not run on FPGAs. Therefore, we had to use state-of-the-art models trained directly in Paddle. For this reason, we cannot analyze the drop in accuracy. Although PaddleLite also features Python APIs, the installation is complex due to the lack of English documentation. Therefore, the inference scripts, in this case, are written in C++. The models are slightly different from those used so far. We compared them with tools such as Netron [55] and we evaluated the number of parameters and the number of expected operations. They are practically identical to our models and the data precision is FLOAT32. Ideally, the board can work with any data type, but the installed FPGA Kernel works in floating-point. The results are shown in Table 7.9.

SSD 640x640	U-Net 224x224	MobileNet V2 224x224
91.6 ms	52.2 ms	10.17 ms

Table 7.9: Inference time on Xilinx board

From the results, we can say that the performance of the MPSoC Ultrascale+ ranks between that of Jetson's GPU and Coral's Edge TPU. These are values that we could expect, since we work with small batch sizes which gives an advantage over the GPU and since in terms of processing capabilities, the FPGA is capable of delivering more operations per second. The ASIC by Google is specifically designed for this purpose and better optimized. As stated also in the technical specifications, it is able to guarantee 4 TOPS, while the FPGA is limited at 1.2. A summary of the number of FPS we can achieve with each device is shown in Figures 7.4 and 7.5.

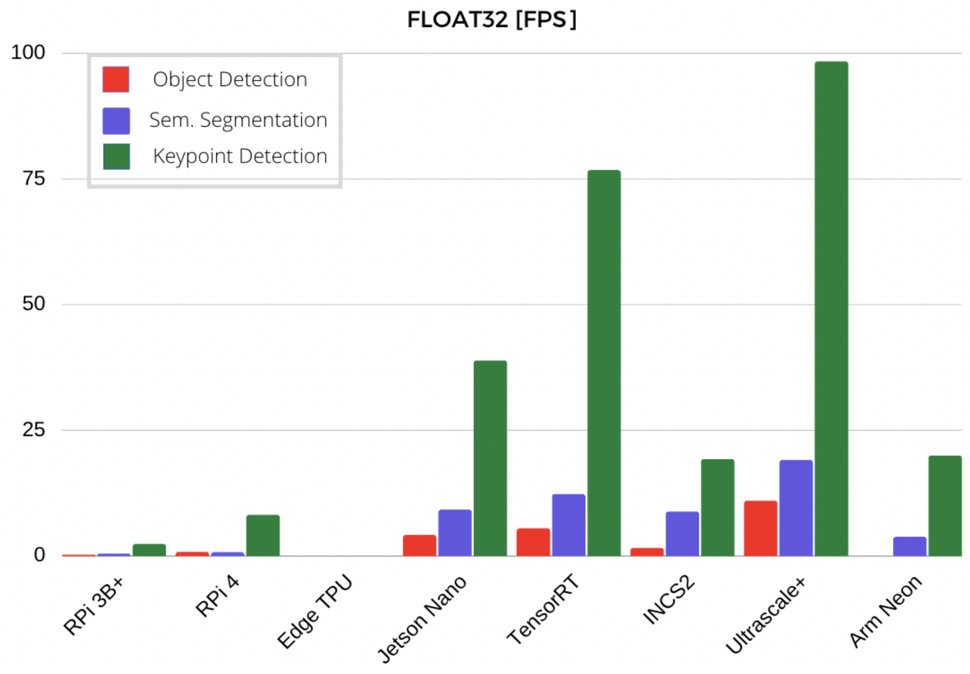


Figure 7.4: FLOAT32 models performance [FPS]

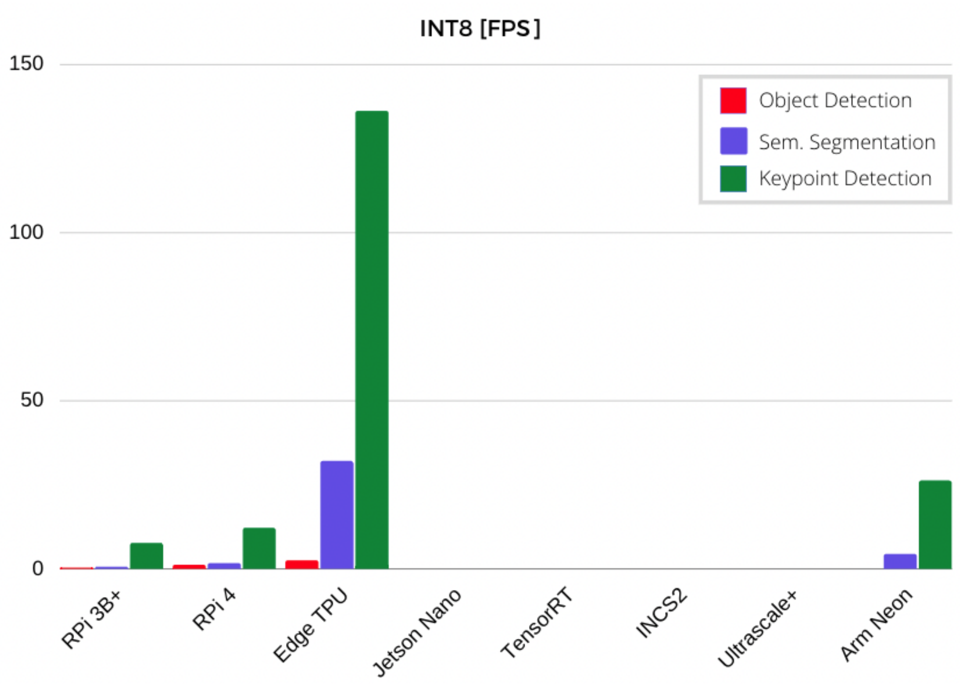


Figure 7.5: INT8 models performance [FPS]

Chapter 8

Conclusion

8.1 Future implementations

Although we have tried to employ the widest possible range of devices and scenarios, covering many cases, the work done can be extended in several directions. We limited the work to convolutional neural networks as we were interested in solving computer vision problems, widely used in every field. To test more thoroughly the devices we have mentioned, however, we could work on structurally different models, such as Recurrent Neural Networks. Although the architectures of the accelerators we tested are particularly suited to image processing and CNN execution, it would be interesting to test the performance in other contexts as well. Another aspect that we have not considered is the consumption of the accelerators. One of the key points of these devices in fact concerns energy efficiency. They are designed to be battery powered and being able to extend their life is essential. Another possible future implementation is to extend the analysis to other kinds of devices as well. Particular attention should be directed to microcontrollers. Being able to perform artificial intelligence algorithms on so cheap and widely used devices, would open the way to endless other applications. These devices are even more limited in memory and computing power, therefore the development is even more complex. The compromise in terms of accuracy becomes not negligible, and inference times are longer. The models would be easily convertible thanks to TensorFlow Micro, a further extension of TF Lite for TinyML [21]. Moreover, for the space domain we are investigating, using ad-hoc devices like the ones we have presented is the best choice. However, the same techniques can also be applied to SoCs embedded in modern smartphones. They feature AI accelerators that can be used in many possible commercial applications. Evaluating their performance, highlighting pros and cons, and ranking the main SoCs available would be interesting also for mobile developers.

8.2 Final considerations

The results show that optimization techniques and especially ad-hoc hardware architectures bring incredible advantages to speed up inference. Starting with the most impressive results we get, provided by the Edge TPU, we are able to execute each task at several FPS. We can speed up the same tasks by 80 times with respect to the Raspberry Pi. This enables real-time applications on mobile devices, which is our primary goal. Moreover, in our case, we have voluntarily decided to adopt networks devoted more to speed than to accuracy. From the obtained inference times, we realize that these devices would provide predictions at several FPS even on more complex networks and with higher resolution input images. Surprisingly, the INCS 2, based on Intel Myriad VPU, provides results slightly below expectations when compared to other accelerators. Coral, on the other hand, is apparently the one that performs best, except for Object Detection, which is partially delegated to the CPU. We are only evaluating the inference times. A real application needs also pre-processing and post-processing steps that cannot be easily speeded up with ASIC devices like Edge TPU. The GPUs and FPGAs we employed can speed up these processes as well. Deciding which device is best is therefore not easy. In our case the INCS 2 should be excluded, but with other models it is not necessarily the worst choice. These considerations apply only to the time for inference, since the devices work at different data precision that can affect, even significantly, the accuracy as shown in Chapter 7. Furthermore, another aspect to consider during the development of these systems, is the compatibility of the models with the optimization frameworks. Deep learning development is proceeding at a frantic pace. Very often new approaches are proposed, and new methods are invented. For manufacturers, keeping up with the times is difficult and not all operations are supported. Before starting the development, it is important to check if custom layers are supported, otherwise, implementation is not so straightforward. What remains an obstacle for the full adoption of such systems on space applications is the lack of reliability. Matching both speed of execution with maximum accuracy is difficult. Nearly infallible systems are needed to fill important roles on expensive spacecraft. The optimization algorithms we have tried sometimes significantly affect the quality of the output. For example, we expected better results regarding the quantization for the object detection. A last consideration to point out concerns the general trend of the processor market. With newer devices of the same family, we can speed up of about 4 times the inference, as in the case of our Arm-based Raspberry boards. This means that, continuing this way, year after year we will be able to run more and more accurate networks in a faster and faster way. At the moment we are already able to run quickly models for many tasks, but in the near future the possible applications could be many more.

Bibliography

[1] BCC Research LLC. *Deep Learning: Global Markets*. en. Tech. rep. IFT214A. Dec. 2020. URL: https://www.bccresearch.com/market-research/information-technology/deep-learning-market.html (visited on 11/23/2020) (cit. on p. 1).

[2] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536 (cit. on pp. 1, 5).

[3] AIKO S.R.L. *AIKO: Autonomous Space Missions*. URL: https://www.aikospace.com/ (visited on 11/24/2021) (cit. on p. 1).

[4] Kyle Millar, Adriel Cheng, Hong Gunn Chew, and Cheng-Chew Lim. «Using Convolutional Neural Networks for Classifying Malicious Network Traffic». In: *Deep Learning Applications for Cyber Security*. Ed. by Mamoun Alazab and MingJian Tang. Cham: Springer International Publishing, 2019, pp. 103–126. ISBN: 978-3-030-13057-2. DOI: 10.1007/978-3-030-13057-2_5. URL: https://doi.org/10.1007/978-3-030-13057-2_5 (cit. on p. 7).

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. eprint: arXiv:1506.02640 (cit. on p. 7).

[6] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. «Ssd: Single shot multibox detector». In: *European conference on computer vision*. Springer. 2016, pp. 21–37 (cit. on pp. 7, 8).

[7] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. «Region-Based Convolutional Networks for Accurate Object Detection and Segmentation». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1 (2016), pp. 142–158. DOI: 10.1109/TPAMI.2015.2437384 (cit. on p. 7).

[8] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. «U-net: Convolutional networks for biomedical image segmentation». In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241 (cit. on p. 9).

[9] An-Min Zou and Krishna Dev Kumar. «Neural Network-Based Distributed Attitude Coordination Control for Spacecraft Formation Flying With Input Saturation». In: *IEEE Transactions on Neural Networks and Learning Systems* 23.7 (2012), pp. 1155–1162. DOI: `10.1109/TNNLS.2012.2196710` (cit. on p. 10).

[10] European Space Agency. *Artificial Ingelligence in sapce*. URL: `https://www.esa.int/Enabling_Support/Preparing_for_the_Future/Discovery_and_Preparation/Artificial_intelligence_in_space` (visited on 11/20/2021) (cit. on p. 10).

[11] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. *A Survey of Model Compression and Acceleration for Deep Neural Networks*. 2017. eprint: `arXiv:1710.09282` (cit. on p. 12).

[12] Sepp Hochreiter. «The vanishing gradient problem during learning recurrent neural nets and problem solutions». In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116 (cit. on p. 13).

[13] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. «Pruning filters for efficient convnets». In: *arXiv preprint arXiv:1608.08710* (2016) (cit. on p. 14).

[14] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. «What is the state of neural network pruning?» In: *arXiv preprint arXiv:2003.03033* (2020) (cit. on p. 14).

[15] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. *Pruning Convolutional Neural Networks for Resource Efficient Inference*. 2016. arXiv: `1611.06440` (cit. on p. 15).

[16] R. Reed. «Pruning algorithms-a survey». In: *IEEE Transactions on Neural Networks* 4.5 (1993), pp. 740–747. DOI: `10.1109/72.248452` (cit. on p. 15).

[17] Song Han, Jeff Pool, John Tran, and William J Dally. «Learning both weights and connections for efficient neural networks». In: *arXiv preprint arXiv:1506.02626* (2015) (cit. on p. 15).

[18] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. eprint: `arXiv:1510.00149` (cit. on p. 15).

[19] Steven J. Nowlan and Geoffrey E. Hinton. «Simplifying Neural Networks by Soft Weight-Sharing». In: *Neural Computation* 4.4 (1992), pp. 473–493. DOI: 10.1162/neco.1992.4.4.473 (cit. on p. 15).

[20] Colby R. Banbury et al. *Benchmarking TinyML Systems: Challenges and Direction.* 2020. eprint: arXiv:2003.04821 (cit. on p. 15).

[21] Robert David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems.* 2020. eprint: arXiv:2010.08678 (cit. on pp. 15, 70).

[22] Ping Services. *Blade Monitoring Hardware.* URL: https://pingmonitor.co/media.html (visited on 11/20/2021) (cit. on p. 16).

[23] PlantVillage - Offered By Google Commerce Ltd. *PlantVillage Nuru.* URL: https://play.google.com/store/apps/details?id=plantvillage.nuru&hl=en_GB (visited on 11/20/2021) (cit. on p. 16).

[24] Mate Kisantal, Sumant Sharma, Tae Park, Dario Izzo, Marcus Märtens, and Simone D'Amico. «Satellite Pose Estimation Challenge: Dataset, Competition Design and Results». In: *IEEE Transactions on Aerospace and Electronic Systems* PP (Apr. 2020), pp. 1–1. DOI: 10.1109/TAES.2020.2989063 (cit. on p. 18).

[25] Jean-Sébastien Ardaens, Simone D'Amico, and Oliver Montenbruck. «Final commissioning of the PRISMA GPS navigation system». In: *22nd International Symposium on Spaceflight Dynamics.* Vol. 28. 2011 (cit. on p. 18).

[26] Creative Commons. *Attribution-NonCommercial-ShareAlike 3.0 Italy (CC BY-NC-SA 3.0 IT).* URL: https://creativecommons.org/licenses/by-nc-sa/3.0/it/deed.en (visited on 11/24/2021) (cit. on p. 18).

[27] Inc Labelbox. *LabelBox.* URL: https://labelbox.com (visited on 11/04/2021) (cit. on p. 20).

[28] Bo Chen, Jiewei Cao, Alvaro Parra, and Tat-Jun Chin. «Satellite Pose Estimation with Deep Landmark Regression and Nonlinear Pose Refinement». In: (2019). arXiv: 1908.11542 (cit. on p. 22).

[29] Connor Shorten and Taghi M Khoshgoftaar. «A survey on image data augmentation for deep learning». In: *Journal of Big Data* 6.1 (2019), pp. 1–48 (cit. on p. 22).

[30] Albumentations. *Albumentations: fast and flexible image augmentation.* URL: https://albumentations.ai/ (visited on 11/24/2021) (cit. on p. 22).

[31] Google. *TensorFlow.* URL: https://www.tensorflow.org/ (visited on 11/24/2021) (cit. on p. 24).

[32] *Keras: the python deep learning library.* URL: https://keras.io/ (visited on 11/24/2021) (cit. on p. 25).

[33] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. «SpotTune: Transfer Learning Through Adaptive Fine-Tuning». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019 (cit. on p. 26).

[34] Princeton University © 2020 Stanford Vision Lab Stanford University. *ImageNet*. URL: `https://image-net.org/` (visited on 11/24/2021) (cit. on p. 26).

[35] Minyoung Huh, Pulkit Agrawal, and Alexei A. Efros. *What makes ImageNet good for transfer learning?* 2016. eprint: `arXiv:1608.08614` (cit. on p. 26).

[36] Pavel Yakubovskiy. *Segmentation Models*. `https://github.com/qubvel/segmentation_models`. 2019 (cit. on p. 28).

[37] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. *Benchmarking TPU, GPU, and CPU Platforms for Deep Learning*. 2019. eprint: `arXiv:1907.10701` (cit. on p. 31).

[38] Intel. *Introducing Neon for Arm*. URL: `https://developer.arm.com/documentation/102474/0100/Fundamentals-of-Armv8-Neon-technology` (visited on 11/11/2021) (cit. on p. 34).

[39] Google. *Coral supported operations*. URL: `https://coral.ai/docs/edgetpu/models-intro/#supported-operations` (visited on 11/24/2021) (cit. on p. 36).

[40] nVidia. *CUDA C++ Programming Guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 11/18/2021) (cit. on p. 39).

[41] MYIR Tech Limited. *Myir FZ3 Card*. URL: `http://www.myirtech.com/list.asp?id=630` (visited on 11/07/2021) (cit. on p. 40).

[42] 2021 Xilinx. *Xilinx Zynq UltraScale+ MPSoC*. URL: `https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html#productTable` (visited on 11/07/2021) (cit. on p. 40).

[43] Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. «Large-scale FPGA-based convolutional networks». In: *Scaling up Machine Learning: Parallel and Distributed Approaches* 13.3 (2011), pp. 399–419 (cit. on p. 41).

[44] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. *Deep Learning on FPGAs: Past, Present, and Future*. 2016. eprint: `arXiv:1602.04283` (cit. on p. 41).

[45] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. «A GPU-outperforming FPGA accelerator architecture for binary convolutional neural networks». In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 14.2 (2018), pp. 1–16 (cit. on p. 41).

[46] Babak Falsafi, Bill Dally, Desh Singh, Derek Chiou, J Yi Joshua, and Resit Sendag. «FPGAs versus GPUs in data centers». In: *IEEE Micro* 37.1 (2017), pp. 60–72 (cit. on p. 41).

[47] BN Singh, AJE Foreman, and H Trinkaus. «Radiation hardening revisited: role of intracascade clustering». In: *Journal of nuclear materials* 249.2-3 (1997), pp. 103–115 (cit. on p. 43).

[48] Google. *TensorFlow Lite and TensorFlow operator compatibility.* URL: `https://www.tensorflow.org/lite/guide/ops_compatibility` (visited on 11/04/2021) (cit. on p. 46).

[49] Copyright 2020 Google LLC. All rights reserved. *Run Inference on the Edge TPU.* URL: `https://coral.ai/docs/edgetpu/tflite-python/#overview` (visited on 11/08/2021) (cit. on p. 48).

[50] onnx - Github. *tensorflow-onnx.* URL: `https://github.com/onnx/tensorflow-onnx` (visited on 11/18/2021) (cit. on pp. 53, 65).

[51] NVIDIA - Github. $onnx_graphsurgeon$. URL: `https://github.com/NVIDIA/TensorRT/tree/master/tools/onnx-graphsurgeon` (visited on 11/18/2021) (cit. on p. 53).

[52] Baidu. *PaddlePaddle.* URL: `https://www.paddlepaddle.org/` (visited on 11/24/2021) (cit. on p. 53).

[53] Xilinx. *Vitis AI.* URL: `https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html` (visited on 11/24/2021) (cit. on p. 53).

[54] Baidu. *X2Paddle.* URL: `https://github.com/PaddlePaddle/X2Paddle` (visited on 11/24/2021) (cit. on p. 54).

[55] OpenJS Foundation. *Netron | Apps | Electron.* URL: `https://www.electronjs.org/apps/netron` (visited on 11/30/2021) (cit. on p. 66).