UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Comparative performance analysis between different development boards in image recognition tasks for biomedical and automotive applications.

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Gilabert Oltra, Ivan

Tutor: Manzoni, Pietro

External cotutor: Bliznuks, Dmitrijs

ACADEMIC YEAR: 2024/2025

Als meus.

Resum

Aquesta tesi presenta un estudi sobre l'avaluació comparativa del rendiment de dispositius en el “edge” populars, com ara la Raspberry Pi 5 i l’NVIDIA Jetson Nano, per a tasques de classificació d'imatges, aprofitant biblioteques avançades com ara TensorFlow Lite i PyTorch. L'estudi mesura sistemàticament la potència de processament, la velocitat i la precisió de la classificació a través d'una sèrie de proves de classificació d'imatges controlades i avaluades quantitativament. A partir d'aquests resultats, es seleccionarà el dispositiu òptim per desenvolupar una aplicació pràctica per al món real: detecció en temps real de sis tipus d'afeccions de la pell mitjançant una càmera web.

La investigació s'estructura en tres fases progressives. Primer, un estudi exploratori inicial utilitzant un petit subconjunt de dispositius en el “edge”, un únic model de classificació preentrenat i un conjunt de dades d'imatges conegut. En segon lloc, una avaluació comparativa que implica tots els dispositius en el “edge” disponibles, múltiples models ajustats i conjunts de dades diferents no familiars dels models preentrenats. Finalment, el desenvolupament d'una aplicació per al món real utilitzant el dispositiu i el model de classificació amb millor rendiment de l'avaluació prèvia, ajustats per detectar i diagnosticar problemes de la pell en temps real.

Aquesta metodologia de tres etapes garanteix que el projecte avance des de l'avaluació comparativa fonamental fins a la implementació pràctica i aplicada. Mitjançant una avaluació i selecció rigoroses, la tesi pretén oferir una eina de diagnòstic basada en la computació en el “edge”, rendible i d'alt rendiment, que equilibre la precisió, la latència i l'accessibilitat, fomentant contribucions significatives a la computació en el “edge” en aplicacions sanitàries.

Paraules clau: classificació d’imatges, enginyeria de computadors, aprenentatge automàtic, computació en el “edge”, ajust fi, ia en aplicacions sanitàries

Abstract

This thesis presents a study on benchmarking the performance of popular edge devices, such as the Raspberry Pi 5 and NVIDIA Jetson Nano, for image classification tasks, leveraging advanced libraries including TensorFlow Lite and PyTorch. The study systematically measures processing power, speed, and classification accuracy across a series of controlled, quantitatively evaluated image classification tests. Based on these results, the optimal device will be selected to develop a practical real-world application: real-time detection of six types of skin problems or conditions using a webcam.

The investigation is structured into three progressive phases. First, an initial exploratory study using a small subset of edge devices, a single pretrained classification model, and a well-known image dataset. Secondly, a comparative evaluation involving all available edge devices, multiple fine-tuned models, and distinct datasets unfamiliar from the pretrained models. Lastly, the development of a real-world application using the best-performing edge device and model from the prior evaluation, fine-tuned to detect and diagnose skin problems in real time.

This three-stage methodology ensures that the project advances from foundational benchmarking to applied, practical deployment. Through rigorous evaluation and selection, the thesis aims to deliver a cost-effective, high-performance edge-based diagnostic tool, balancing accuracy, latency, and accessibility, fostering meaningful contributions to edge computing in health applications.

Keywords: image classification, computer engineering, machine learning, edge computing, fine tuning, biomedical ai

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

Nowadays, edge devices have proliferated across diverse sectors, driven by their enhanced affordability, compact size, and increasingly powerful hardware capabilities. The global edge AI market, for instance, was estimated at USD 20.78 billion in 2024 and is projected to reach USD 66.47 billion by 2030, with a compound annual growth rate (CAGR) of 21.7 % [1]. These devices are ubiquitous, from industrial sensors to smart cameras, providing real-time data processing closer to data sources and reducing reliance on centralized cloud infrastructures [2].

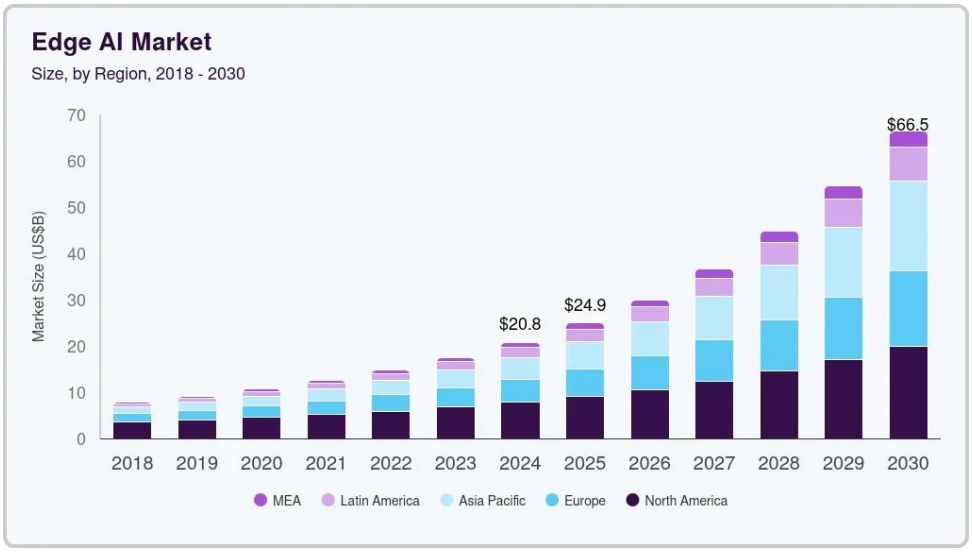

Figure 1 Edge AI Market [1]

This technological trend has opened new opportunities in image classification using AI, particularly for healthcare applications. Thanks to improvements in model design (e.g., pruning, quantization, knowledge distillation), it is now feasible to deploy fine-tuned deep learning models on resource-constrained hardware, achieving fast and efficient inference [3]. In medical diagnostics, such on-device AI systems can complement traditional imaging tools, like X-rays CT, MRI or ultrasound, to deliver rapid, cost-effective, and privacy-preserving analysis without external connectivity [4].

The convergence of affordable, compact edge hardware and optimized AI models enables a powerful paradigm: real-time image classification for biomedical applications. This combination promises diagnostic tools that are not only effective but also accessible, low-cost, and deployable in diverse settings, ranging from clinics to remote areas, thus significantly enhancing both reach and impact.

1.1 Motivation

From a very young age, I have been driven by a strong vocational calling: to leverage technology in support of those who need it the most. This personal aspiration has guided my academic journey and strengthened my determination to pursue impactful engineering projects. My Erasmus stay in Riga marked a turning point in that journey. There, I met Dmitrijs, a seasoned Computer Engineering professor whose deep commitment to crafting efficient systems with real-world relevance profoundly inspired me. Through our discussions and meetings, the seed for my Final Degree Project was planted: to design an IoT-based system capable of aiding healthcare professionals, making a diagnostic tool for skin problems more efficient, affordable, and accessible, especially in resource-constrained settings.

On a technical level, the capabilities of Deep Neural Networks (DNNs) captivate me. The notion that a pretrained model can be fine-tuned to perform highly accurate classification tasks, even on limited hardware, is nothing short of remarkable. This project provides me with the chance to explore these methods firsthand, experimenting with models, tuning them for specific tasks, and deploying them on edge platforms. Beyond academic interest, this hands-on exploration offers invaluable insights that will help clarify my professional direction, whether in industrial research, applied development, or advanced academic study.

From a professional standpoint, I see this project as a platform to distinguish myself in the rapidly expanding fields of informatics & AI Engineering, in particular. By operating at the intersection of artificial intelligence, embedded hardware, and bioinformatics, I am forging a multidisciplinary profile that speaks directly to future employers and collaborators. Developing edge-based diagnostic tools not only sharpens my technical skills but also demonstrates my readiness to tackle complex, socially meaningful challenges. The synthesis of hardware proficiency, model optimization, and biomedical understanding sets me apart in a competitive landscape.

In essence, my motivation stems from a personal desire to make a difference, enriched by academic mentorship and propelled by fascination with state-of-the-art AI techniques. This project embodies my commitment to harness technology for societal benefit, clarifies my career trajectory, and empowers me to contribute competently to areas where innovation and compassion converge. I approach this thesis with enthusiasm, purpose, and the conviction that my work can make a meaningful impact.

1.2 Objectives

Given the means available to us at the time of developing this thesis, the goals pursued are threefold:

- Compare edge devices under equal conditions. Evaluate various edge devices provided by Riga Technical University alongside my personal laptop and a powerful custom-built PC. Using the same model, dataset, and computational load, rank these platforms by cost-effectiveness in image classification.

- Develop a real-world application aimed to the top-ranked hardware. Based on the highest-performing edge device and its particularities, build an application that classifies webcam images and identifies the most probable one of six skin conditions, such as melanoma, chickenpox, or measles.

- Assess practical suitability and efficiency. Measure how edge devices balance accuracy, latency, and affordability compared to conventional systems, demonstrating scenarios where edge-based AI is most effective.

These objectives are specific, actionable, and aligned with our available resources. They guide the study from benchmarking to application, ensuring clarity, relevance, and practical value.

1.3 Expected impact

The comparative study of edge device efficiency in image classification developed is intended to benefit researchers, medical professionals, and students worldwide.

By providing clear performance rankings, this work helps decision-makers determine the most cost-effective combination of edge device, hardware configuration, and deep neural network (DNN) model for their specific needs. Whether they begin with an existing trained model or already own a particular edge device, they can refer to this study to select the ideal counterpart, either choosing a device that runs their model most efficiently and affordably or selecting the optimal DNN model to fine-tune for a given edge platform. This knowledge empowers stakeholders to make informed, context-specific decisions without having to conduct independent benchmarking.

Another significant impact derives from the practical application created during this project. The software developed and designed to detect and classify skin problems has the potential to facilitate early diagnosis in environments lacking advanced medical imaging facilities. Particularly in low-resource or underserved regions, where access to dermatological expertise may be limited, such a tool could accelerate recognition of serious conditions and trigger timely interventions. By lowering the barrier to diagnostic capability, this solution will directly contribute to saving lives if adopted by healthcare centers or field clinics.

At a broader level, this work aligns meaningfully with several United Nations Sustainable Development Goals (SDGs). SDG 9 (Industry, Innovation and Infrastructure) is supported by advancing AI-driven applications that enhance quality of life through accessible, innovative technologies. SDG 10 (Reduced Inequalities) is addressed by intentionally ensuring affordability in the final biomedical application, thereby enabling its use across diverse socioeconomic contexts. SDG 3 (Good Health and Well-Being) is inherently promoted through the application's contribution to health by enabling earlier detection and improving access to diagnostic tools.

This thesis is expected to have both scholarly and humanitarian impact: it equips the global research and healthcare community with a practical benchmarking resource and delivers a tangible diagnostic tool capable of extending access to early detection.

1.4 Structure of the report

The structure of this thesis is designed to guide the reader through its development in a progressive and logical manner. It comprises seven chapters, including the *Introduction* presented now. The document begins with it, with the objective of framing the thesis context, as well as including the motivation for it, specific objectives, expected impact, and this structural review. Following, in the *State of the art* chapter, the current landscape of edge devices on the market is reviewed, followed by the review of image-classification ML models. Finally, it synthesizes the combination of both by discussing the state of edge platforms running ML-based image classification models focused on biological applications. This consolidation is crucial for understanding the theoretical foundations.

Next, the *Experimental framework & setup* chapter discusses the theoretical basis for the study introducing Edge and Tiny machine learning and describes the available edge devices and the selected image classification models that form the core of the experimental setup. Then, *An introductory test* chapter takes the reader through the first steps of the author through the field of single label classification tasks, including the rationale, configurations, and insights gleaned from testing feasibility. Following that, the *Benchmarking fine-tuned models* chapter details the methodology: model training and fine-tuning, development environments, operating systems, and the outcome of side-by-side tests that inform the ranking of hardware–software combinations.

In the *A biomedical application* chapter, and aimed at the top-performing configuration, the design, development and deployment of the real-world biomedical application is described in.

Lastly, the *Conclusions* chapter synthesizes the achievements, discusses the lasting contributions of this work, and proposes potential directions for future work. The thesis concludes with the bibliography and appendices, which support reproducibility and transparency.

CHAPTER 2
State of the art

Advances in embedded hardware, microcontrollers, and connectivity have kicked off in a paradigm shift in how machine learning can be deployed. Tiny Machine Learning (TinyML) enables ML models to run locally on devices with minimal power consumption, limited memory, and constrained processing capacity. Its increasing prominence lies in unlocking real-time analytics, reducing latency, improving data privacy, and eliminating dependence on cloud infrastructure [5] [6]. By moving intelligence to the edge, TinyML brings ML within reach of ubiquitous, low-cost embedded platforms.

The impact of TinyML on real-world applications is growing rapidly. In healthcare, for instance, TinyML-enabled digital stethoscopes now process auscultation data in real time for respiratory disease detection, delivering 96% accuracy using compact convolutional neural networks on low-power microcontrollers [7]. In another domain, ultra-low-power TinyML systems have achieved object detection at 30 frames per second while consuming just 160 mW by integrating custom neural coprocessors and instruction sets optimized for edge environments [8].

These real-world implementations described in the fields of healthcare monitoring and visual processing powerfully illustrate how edge devices running ML models are already delivering tangible benefits in latency-critical and resource-constrained contexts. The embedded execution of ML models, enabled by frameworks such as TensorFlow Lite Micro (TFLM), makes such applications feasible across a host of disciplines, from medical diagnostics to environmental sensing [9].

This chapter presents a comprehensive review of the current ecosystem, organized into three sections. First, the *Edge devices* subchapter will examine available platforms, their capabilities, and benchmarking power. Next, the *Image classification models* one will assess the evolution of algorithms, from mobile-optimized CNNs to lightweight classification architectures suitable for edge deployment. Finally, the *Edge-based ML for biology* subchapter will explore recent academic projects that show how image classification models and hardware converge, enabling TinyML in real-world use cases, especially those that focus on biological applications.

By grounding the reader in both technological maturity and practical deployment, this chapter lays the conceptual foundation necessary for the benchmarking and application development that follows. The insights drawn here will inform the design of experiments and evaluation in later chapters, ensuring they are anchored in the current capabilities of edge and ML technologies.

2.1 Edge devices

Edge computing emerged from the necessity to process data locally in the 1990s, initially through content delivery networks (CDNs) that distributed cache servers closer to end users to reduce latency [10] [11]. Over time, the concept broadened beyond content delivery to encompass distributed computing architectures that perform storage and computation near data sources. This evolution laid the foundation for modern edge devices capable of enabling real-time, low-latency, and privacy-conscious applications in various domains.

Modern edge devices, such as single-board computers and embedded AI platforms, differ widely in architecture, performance, power consumption, and price. The *NVIDIA Jetson Nano*, introduced in 2019 as an affordable AI module (~99 EUR), features a quad-core ARM Cortex-A57 CPU, a 128-core NVIDIA Maxwell GPU, and delivers around 472 GFLOPS of computational throughput. It operates within a power envelope of 5 to 10 W, making it suitable for continuous inference tasks [12]. Its successors, such as the *Jetson Orin Nano* and *Orin NX*, offer significantly higher performance, up to 34–157 TOPS (Tera Operations Per Second) in sparse INT8 workloads while maintaining relatively moderate power consumption [13].

In contrast, the *Raspberry Pi 5*, released in 2023, is optimized for versatility rather than raw AI performance. It centers around a quad-core ARM Cortex-A76 CPU and consumes around 3 to 5 W under general load. With an optional AI Kit accessory, it can reach performance up to 26 INT8 TOPS for AI inference but remains inherently general-purpose in its design [12].

Comparatively, edge device selection involves trade-offs between computing power, energy efficiency, cost, and ecosystem support [14]. While Jetson platforms provide substantial parallel processing power through integrated GPUs, their higher costs and power needs make them better suited for fixed or powered deployments.

Conversely, Raspberry Pi devices offer low cost, low power consumption, and broad community support, which may be advantageous for educational or portable use cases.

Among the latest edge computing platforms, two hardware families stand out for their exceptional inference strength. *NVIDIA Jetson Orin AGX* delivers up to 275 INT8 TOPS within a power envelope of 15 to 60 W, making it one of the most powerful modules for embedded AI at present [15].

In real-world usage, *Jetson AGX Orin* achieved 290 FPS on 1080p face detection tasks through optimized co-use of its CPU, GPU, and other accelerators, and realized a power saving of ≈800 mW compared to non-optimized configurations [16].

Figure 2 Jetson AGX Orin delivers 8X the AI performance of Jetson AGX Xavier [17]

On the other end, Renesas RZ/V2H's DRP-AI3 delivers 80 INT8 TOPS with an energy efficiency of ~10 TOPS/W, enabling inference for image classification up to 14 times faster than lower-end variants [18]. Together, these platforms define the current benchmarks in edge AI compute performance, each excelling in different dimensions of throughput and efficiency.

2.2 Image classification models

The evolution of image classification models reflects a persistent drive toward achieving high accuracy while satisfying the constraints of embedded and edge hardware. Early neural networks such as *AlexNet* and *VGG* prioritized accuracy at the expense of computational demand, making them impractical for low-resource deployments. In response, research pivoted toward mobile-optimized and lightweight architectures that balance performance with efficiency.

Lightweight models such as *MobileNet* introduced depthwise separable convolutions to dramatically reduce parameters and FLOPs, enabling efficient real-time inference on mobile and embedded devices [19]. Its successors, *MobileNetV2* and *V3*, further improved accuracy through inverted residual blocks, linear bottlenecks, and neural architecture search guided by latency constraints [20].

The *MobileNetV4* family, released in 2024, employs compound scaling and attention mechanisms to optimize speed and accuracy in edge environments [21].

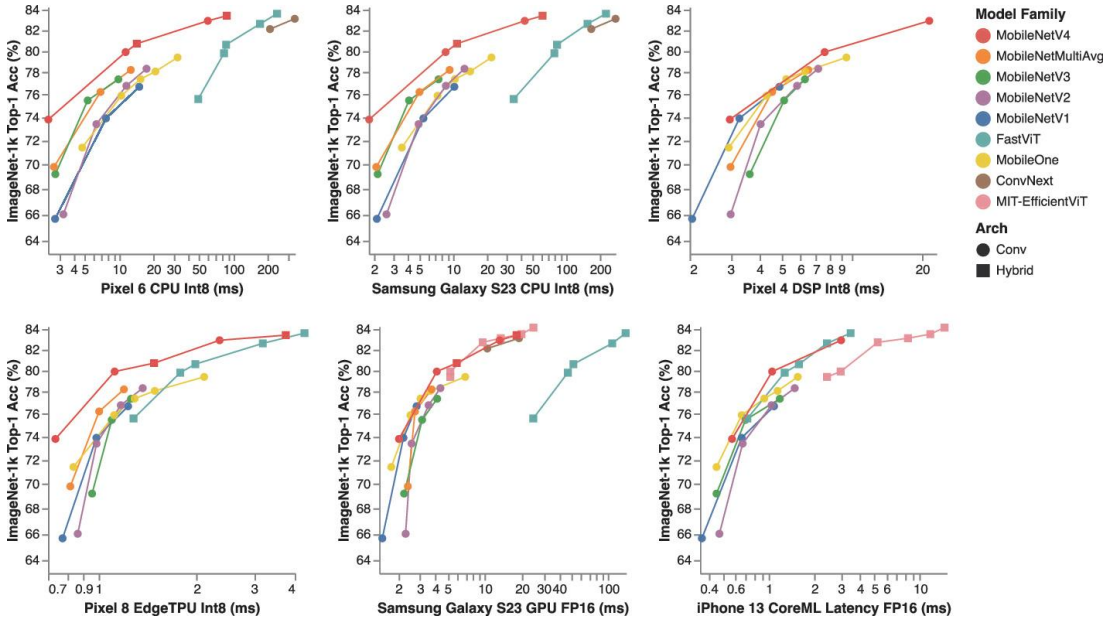

Figure 3 Evolution on the MobileNet family of models [21]

Alternative approaches include *ShuffleNet*, designed for mobile platforms with extreme resource constraints. By utilizing pointwise group convolutions and channel shuffling, *ShuffleNet* delivers significant computation reduction, achieving approximately 13x speedup over *AlexNet* on ARM-based devices while maintaining similar accuracy [22].

Beyond CNNs, *MCUNet* exemplifies the emerging frontier of Tiny Deep Learning on microcontrollers. *MCUNet* co-optimizes architecture (TinyNAS) and inference engine (TinyEngine), achieving over 70% ImageNet top-1 accuracy on MCU platforms using 3.5x less memory and 5.7x less flash than quantized *MobileNetV2* and *ResNet-18* variants [23].

More recently, *MobileViT* bridges compact CNN and lightweight Vision Transformer paradigms. Utilizing self-attention modules and convolutional layers, *MobileViT* achieves 78.4% top-1 accuracy on ImageNet-1k using only about 6 million parameters, outperforming *MobileNetV3* and *DeIT* under similar complexity constraints [24].

Model	# Params.	Top-1
MobileNetv1	2.6M	68.4
MobileNetv2	2.6M	69.8
MobileNetv3	2.5M	67.4
ShuffleNetv2	2.3M	69.4
ESPNetv2	2.3M	69.2
Mobile ViT-XS	2.3M	**74.8**

Table 1 Comparison of MobileViT with other ligh-weight CNNs with similar parameters [24]

These models illustrate the trajectory from conventional CNNs to highly optimized architectures suitable for edge deployment. They highlight how architectural innovations and

co-design frameworks advance the feasibility of image classification under tight resource budgets.

2.3 Edge-based ML for biology

The integration of TinyML into biological image classification began in the early 2020s, when researchers saw the potential of combining lightweight neural networks with low-power hardware to perform diagnostics in remote settings. A notable 2024 experiment conducted at Edinburgh Napier University fine-tuned *MobileNetV2* using 10,000 skin lesion images and deployed it on a *Raspberry Pi 3* connected to a webcam. The system achieved a test accuracy of 78% and a test loss of 1.08, although the developers noted that its accuracy was still below clinical-grade standards, limiting its immediate adoption [25].

Prior to that, in 2022, a pilot study made in collaboration between different university researchers fine-tuned a model and deployed it into a *Raspberry Pi 4* and a small camera to detect skin cancer patches. Their results were promising, but the accuracy and consistency required further validation and improvement of the dataset images definition before practical use [26].

Together, these early experiments illustrated that deploying image-classification models on edge devices for health applications was technically feasible. However, both studies also highlighted limitations in accuracy, model generalization, and clinical applicability.

An experiment now considered state of the art for its significant advances in performance and efficiency is *MedMambaLite*, introduced in August 2025. This hardware-aware model was optimized through knowledge distillation, resulting in a compact student network tailored for edge inference [27]. The model achieves 94.5% accuracy across ten MedMNIST medical imaging datasets, while reducing parameter count by a factor of 22.8x compared to its larger counterpart, *MedMamba*. Crucially, MedMambaLite was deployed on an *NVIDIA Jetson Orin Nano*, delivering 35.6 GOPS/J (a measure of energy efficiency per inference), that outperforms the original model architecture by 63%. This demonstrates that advanced biomedical image classification can be both accurate and power-efficient on high-performance edge hardware.

The deployment pipeline employs modern TinyDL toolchains, offering hardware-aware model compression, efficient inference kernels, and energy-optimized deployment, highlighting how model design and hardware capabilities co-design drives real-world feasibility in medical edge applications.

This state-of-the-art study exemplifies the maturity of TinyML in addressing both accuracy and deployment constraints in biomedical imaging, and firmly establishes the feasibility of clinically relevant image classification on contemporary edge platforms.

CHAPTER 3
Experimental framework & setup

This chapter offers a structured overview of the foundational elements employed throughout this work. It begins with an introduction to Edge and Tiny machine learning, and the specific image classification models selected for our experiments. Following that, each hardware platform is presented in detail, highlighting its characteristics and relevance. Then, outline the evaluation methodology; focusing on performance metrics and benchmarking protocols, and conclude with an overview of the software environment and deployment tools that support the research.

In this section, the resources and configurations used for conducting the experiments will be described, as well as the frameworks upon which they rely. The objective is to provide a clear and comparable foundation that helps the reader understand how the following experimental setups were configured. Within this hardware-software landscape, the objective is to pursue two primary goals: first, to allow future research teams to reproduce the results reported in the following chapters by describing all the necessary hardware-software stack, and second, to acknowledge the reader the importance of the devices differences in architecture, operative systems, commonly used tools and restrictions in order to fit their specific needs, which differ greatly from device to device. This approach allows different architectures to be optimized for accessible and efficient edge computing scenarios.

The specific configuration details for each experiment will be covered in the corresponding chapters. In this chapter, the focus is on presenting the devices, models, and methodology utilized during the later phase of the work, ensuring transparency and reproducibility.

3.1 Understanding Edge & Tiny Machine Learning

In recent years, the proliferation of data-generating devices such as cameras, sensors, and smart instruments has created an urgent demand for near-real-time processing, high responsiveness, and enhanced data privacy. This has driven the evolution of a paradigm in machine learning known as Edge Machine Learning (EdgeML), defined as the process of performing inference, and in some cases training directly on devices situated at the network's periphery, avoiding round-trip delays to remote servers and minimizing data transmission. EdgeML spans deployments across a broad spectrum of platforms, ranging from smartphones and embedded systems to network-edge servers. It has become especially relevant in domains where latency, autonomy, and security are paramount.

Tiny Machine Learning, or TinyML, refers to the deployment of machine learning inference, and increasingly, lightweight training, on extremely resource-constrained devices, typically microcontrollers with as little as 32–512 kB of SRAM, flash storage in the megabyte range, and power budgets in the milliwatt scale [28] [29].

Modern computing paradigms often rely on centralized computation: data flows from sensors, travels over networks, and is processed on remote high-performance servers. TinyML fundamentally challenges that model by moving intelligence to the most lightweight edge devices, in a similar way to EdgeML, thus enabling real-time, offline inference but with a focus on minimal latency and enhanced privacy [30].

EdgeML shares TinyML's fundamental principle of on-device inference but applies it to devices with considerably greater processing and memory capacity, such as embedded Linux platforms, single-board computers, or localized AI accelerators. These systems support richer operating environments and can handle more complex models and workloads, making them particularly suitable for practical deployments that require a balance between efficiency and capability.

Technologies	Latency	Privacy considera-tion	Accuracy	Power Consumption	Reliability	Memory Consumption
Cloud Computing	10-500 ms	Very Low	86-94%	50-1000 W	Depends on the uninterrupted internet connectivity	GBs to TBs
Fog Computing	5-400 ms	Low	85-93%	10-100 W	Depends on active wireless connectivity	MBs to GBs
Edge computing	0.70-350 ms	Low	80-90%	1-10 W	Depends on active wireless connectivity	Few KB
TinyML	0.18-300 ms	Very High	80-90%	25-300 mW	Does not relay on network connectivity	Few KB

Table 2 Comparison of ML performances of existing technologies [31]

Importantly, the present thesis adopts the EdgeML perspective for the main reason of the selected hardware platforms, that range from the Raspberry Pi Zero 2 W to the NVIDIA Jetson Nano. These devices have fewer constraints than microcontrollers but still operate at the network edge. Therefore, the subsequent discussion and model choices are grounded in EdgeML, emphasizing frameworks like TensorFlow Lite rather than its microcontroller-specific counterpart.

3.1.1 Advantages and strategic role of EdgeML

EdgeML offers a compelling balance of operational efficiency and flexibility that fits well with biomedical image classification tasks. Unlike TinyML where devices are severely constrained, requiring specialized frameworks like TensorFlow Lite Micro and extensive model optimization, EdgeML leverages platforms that support standard operating systems and

complete AI libraries like PyTorch or TensorFlow. This reduces the learning curve and fosters faster development, dataset handling, and debugging.

Operational costs are not necessarily higher with EdgeML. Even though edge devices have greater power and computational requirements than microcontrollers, the ease of deployment and toolchain interoperability reduce development time and maintenance overhead [32]. Furthermore, the ability to run full ML toolsets simplifies model tuning, visualization, and data handling, tasks that otherwise demand custom engineering in TinyML environments.

In addition, EdgeML enables rapid iteration and deployment in domains such as medical diagnostics. Platforms with more compute headroom allow for more complex architectures and higher accuracy, while still delivering low latency inference at the point of care, without cloud dependency. As a result, EdgeML supports a productive and scalable development experience for image classification in healthcare scenarios, striking a practical balance between performance, usability, and accessibility.

3.1.2 Hardware-Software co-design: Essential for efficient EdgeML

Hardware-software co-design is a critical strategy for achieving optimal performance in EdgeML applications, especially those involving biomedical image classification, which demand low latency and high accuracy. Co-design aligns neural network architecture, model optimization techniques, and hardware capabilities to identify the most efficient balance between computational demand and inference quality. For example, SECDA (Efficient Hardware-Software Co-Design of FPGA-based DNN accelerators) demonstrated a 3.5x speedup and a 2.9x reduction in energy consumption over CPU-only inference by jointly optimizing model and FPGA design [33]. Such integrated approaches ensure that both model complexity and hardware constraints are addressed in tandem, rather than treated as independent factors.

3.1.3 Frameworks and deployment toolchains

EdgeML deployment is powered by robust frameworks and toolchains that simplify the path from model development to efficient on-device inference. *PyTorch Mobile* and its emerging extension *PyTorch Edge* enable direct deployment of PyTorch models on platforms like Raspberry Pi and Jetson Nano with low overhead and strong hardware portability [34]. For higher-level workflows, *FastAI* leverages pre-trained models and epochs, integrated with PyTorch, making fine-tuning and rapid experimentation particularly accessible on edge platforms.

Intel's *OpenVINO* toolkit excels in optimizing models across Intel CPUs, integrated GPUs, and Neural Compute Sticks (NCS), providing conversion tools and inference engines that yield substantial acceleration, up to 3x faster inference on standard topologies [35].

Meanwhile, *Apache TVM* is an open-source compiler stack that offers deep optimization through graph-level and operator-level transformations. It supports models from *TensorFlow*,

PyTorch, *ONNX*, and others, and automatically generates tuned code for diverse hardware targets, ensuring both performance portability and minimal development effort [36].

These frameworks collectively enable seamless and high-performance deployment of ML models on edge devices, using familiar development tools, while delivering efficiency and maintaining flexibility for experimentation.

3.2 Chosen image classification models

The following subchapter introduces and justifies the selection of the image classification models employed in the thesis. Models were chosen based on their suitability for edge deployment, balancing accuracy, computational efficiency, and proven real-world applicability. Also, crucially, ensure that each architecture exhibits substantial internal differences, avoiding comparison between overly similar models.

3.2.1 MobileNetV2

The MobileNet family, introduced in 2017 by Google, set a new standard for deploying efficient neural networks on mobile and embedded platforms by using depthwise separable convolutions, dramatically reducing both computational load and parameter count [37].

In 2018, *MobileNetV2* refined this architecture by incorporating inverted residual blocks and linear bottlenecks [38]. In each block, the input is first expanded via a 1x1 convolution, then processed with a lightweight 3x3 depthwise convolution, and finally projected back to a low-dimensional space through a 1x1 linear convolution. The unconventional inverted residual structure (thin-thick-thin) and the removal of nonlinearity in narrow layers preserve information flow, improve gradient propagation, and limit computational overhead.

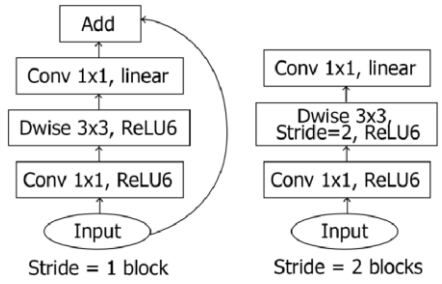

Figure 4 MobileNetV2 architecture [38]

This design enables *MobileNetV2* to deliver strong representational power, achieving around 300 million Multiply-Add operations (MAdds) and approximately 3.4 million parameters for typical configurations, while remaining suitable for edge platforms such as the Raspberry Pi Zero 2 W or Jetson Nano.

In comparison to the original MobileNet, *MobileNetV2* achieves significantly better accuracy and efficiency, owing to its bottleneck structure and improved gradient dynamics. MobileNetV3, while offering even faster inference and modest accuracy gains, relies on hardware-aware NAS techniques and squeeze-and-excitation modules, which add complexity and increase training overhead. For a biomedical image classification task, MobileNetV2 provides an optimal balance: it maintains high accuracy and compact size without the architectural and optimization complexity of MobileNetV3, making it both performant and more reproducible across diverse edge setups, thus, representing the first chosen model for the thesis.

3.2.2 ResNet-18

ResNet-18 is a compact member of the Residual Network family, introduced in 2015 to address the degradation and vanishing gradient problems associated with deep networks through the use of residual (skip) connections [39]. The model comprises 18 layers, of which 17 are convolutional layers grouped into residual blocks sharing identical feature-map dimensions, followed by a fully connected layer and softmax classification layer.

Each residual block contains two 3x3 convolution layers, each followed by batch normalization and a ReLU activation. A skip connection adds the input directly to the output of the second convolution, facilitating gradient flow and enabling training of deeper architectures with improved convergence. The network requires approximately 11.7 million parameters and around 1.8 GFLOPs per forward pass on a 224x224 input image [40].

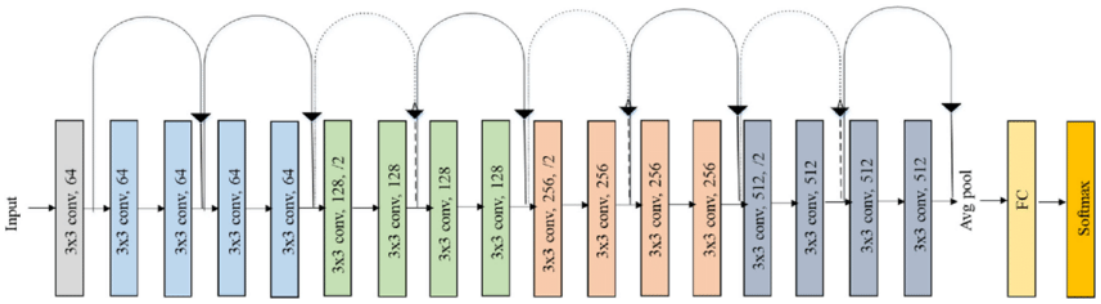

Figure 5 ResNet-18 architecture [41]

Compared to ResNet-50, which contains significantly more layers and employs 1x1, 3x3, and 1x1 bottleneck convolutions, ResNet-18 is far lighter, delivering reduced memory footprint and lower inference latency, critical for deployment on edge devices. While ResNet-50 generally achieves higher accuracy due to its deeper representation capacity, the overhead in computation and power consumption makes it less practical for real-time edge inference. ResNet-18 provides robust feature extraction and convergence behavior, yet remains feasible for rapid fine-tuning and deployment on constrained hardware. This balance between representational power and resource efficiency positions ResNet-18 as the preferred choice over more complex variants for this thesis.

3.2.3 YOLOv5s Classification

YOLOv5s Classification stems from YOLOv5's evolution into a versatile model, integrating image classification capabilities in addition to its original object detection functionality. While YOLOv5 was first released in June 2020, classification support was added in August 2022, enabling deployment in both single-label and multi-label classification tasks [42]. This extension leveraged the same lightweight and production-oriented design that made YOLOv5 popular in real-world deployment.

Structurally, this model adjusts the detection pipeline into a streamlined classification network. It maintains the core backbone named CSP (Cross Stage Partial) network and neck (FPN or PAN) but replaces detection heads with a global pooling layer and fully connected classifier tailored to assign category labels for entire images. The model remains compact: YOLOv5s features approximately 7.3 million parameters (~16.5 GFLOPs at 640-pixel resolution), compared to YOLOv5m (~21.2 M / 49.0 GFLOPs) and YOLOv5l (~46.5 M / 109 GFLOPs), keeping both minimal memory and compute demands.

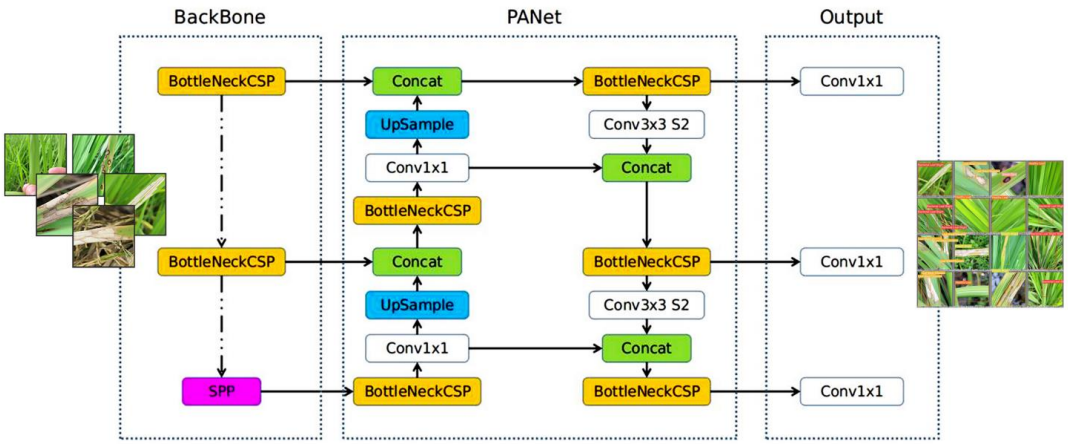

Figure 6 YOLOv5 architecture [43]

For the thesis application, this model seems an optimal choice. Its minimal footprint allows inference under strict resource constraints while retaining adequate representational capacity. Although YOLOv5m and YOLOv5l offer higher accuracy, their larger architectures demand significantly greater memory and compute power, reducing their feasibility for compact embedded systems. Thus, YOLOv5s strikes an appropriate balance of efficiency, deployability, and accuracy for target hardware in this work.

3.3 Chosen hardware & edge devices

The following section presents the hardware platforms selected to execute inference on this thesis experiments, organized according to their availability to the student. The chosen devices cover a broad spectrum of processing capabilities and form factors, deliberately varying in their technical specifications. This diversity enhances the rigor of the study by enabling a comprehensive evaluation across different regular and edge computing configurations.

3.3.1 MSI Modern 14

Represents the personal computer of the student in charge of the thesis. Firstly aimed at developing the algorithms for the computing tests, is later added to the comparison pool with the objective of giving an extra point of view in front of the capabilities of edge devices.

Specific data is summarized in the following chart:

Model	MSI Modern 14 B4MW-056XES
Launch date	01/09/2020
Processor	AMD Ryzen 5 4500U (6 x 2.3 - 4 GHz, Renoir-U (Zen 2))
Graphics card	AMD Radeon Vega 6 (Integrated)
RAM	16GB (2x8GB 3200MHz)
ROM	1TB NVME PCIe SSD
OS	Windows 10 Pro
Python version	3.11 (64-bit), 3.9 (64-bit)

Table 3 MSI B4MW specifications

Figure 7 MSI B4MW

This computer's modern architecture, compute power and versatile development environment offer a valuable reference point for benchmarking. It enables rapid prototyping, detailed preprocessing, and efficient inference testing of image classification models, especially

when comparing deployment performance on edge devices. The Ryzen 5 4500U's six cores and Vega 6 GPU provide a balance of raw processing capability and energy efficiency that exceeds most edge platforms, allowing calibrations and model tuning under less constrained conditions while serving as a high-performance baseline for comparative analysis. Dual Python installations (3.11 for inference tasks and 3.9 for model conversion utilities) ensure compatibility with libraries that are only available to lower or higher builds of the programming language.

3.3.2 Raspberry Pi Zero 2 W

Represents the ultra-low-power single-board computer used to characterize the lower bound of on-device inference in the comparison pool. Selected for its compact form factor, integrated wireless interfaces, and adequate support for lightweight image-classification pipelines on Linux at the edge.

Specific data is summarized in the following chart:

Model	Raspberry Pi Zero 2 W
Launch date	28/10/2021
Processor	Broadcom BCM2710A1 (4x ARM Cortex-A53 - 1GHz)
Graphics card	-
RAM	512 MB LPDDR2
ROM	128GB (microSD)
OS	Raspberry Pi OS Lite (Debian 12 Bookworm) (64-bit)
Python version	3.11.2 (64-bit)

Table 4 Raspberry Pi Zero 2 W specifications

Figure 8 Raspberry Pi Zero 2 W

The Zero 2 W serves as a realistic, low-cost reference point for evaluating algorithmic portability, throughput under tight resource budgets, and the impact of preprocessing strategies when contrasted against higher-performance edge accelerators and the laptop baseline.

3.3.3 Raspberry Pi Zero 2 W & Intel Neural Compute Stick 2

This pairing is conceived to illustrate a cost-efficient means of integrating a dedicated AI accelerator, the Intel Neural Compute Stick 2 (NCS2), while using the Raspberry Pi Zero 2 W as an affordable yet capable host. The Zero 2 W fulfills the essential roles of power delivery, I/O management, and orchestration, enabling hardware-accelerated inference without incurring the higher expense of more powerful computing platforms.

Specific data is summarized in the following chart:

Model	Intel Neural Compute Stick 2
Launch date	14/11/2018
Processor	Intel Movidius Myriad X VPU
Graphics card	-
RAM	-
ROM	-
OS	OpenVINO™ 2022.3.2
Python version	3.9.10 (64-bit)

Table 5 Intel NCS2 specifications

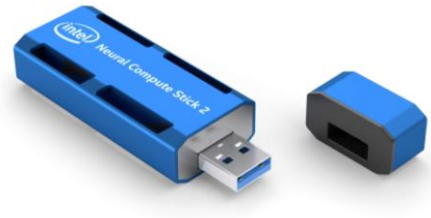

Figure 9 Intel NCS2

It provides a highly cost-effective hardware-accelerated inference solution in a compact USB form factor, powered by the Movidius Myriad X VPU. It is explicitly designed to deliver AI inference acceleration efficiently and affordably at the edge, making it ideal for embedded systems.

When paired with a Raspberry Pi, the NCS2 can offload computationally intensive deep neural network inference tasks from the host CPU, markedly improving throughput and responsiveness.

3.3.4 Raspberry Pi 5

Represents a higher-performance yet still affordable option within the Raspberry Pi family, introduced to evaluate whether increased cost yields proportional gains in edge-computing inference. It shares and also improves some aspects with the Raspberry Pi Zero 2 W, by offering enhanced throughput and I/O performance while remaining cost-efficient.

Specific data is summarized in the following chart:

Model	Raspberry Pi 5 8GB
Launch date	23/10/2023
Processor	Broadcom BCM2712 (4x ARM Cortex-A76 - 2.4GHz)
Graphics card	VideoCore VII with support for OpenGL ES 3.1 & Vulkan 1.2.
RAM	8GB LPDDR4X-4267
ROM	32 GB (microSD)
OS	Raspberry Pi OS Lite (Debian 12 Bookworm) (64-bit)
Python version	3.11.2 (64-bit)

Table 6 Raspberry Pi 5 specifications

Figure 10 Raspberry Pi 5

Introduced in late 2023, features a quad-core ARM processor at 2.4 GHz, delivering approximately 2 to 3x higher CPU performance than previous models. It also integrates a VideoCore GPU and the first-generation RP1 southbridge chip, significantly improving I/O bandwidth and system responsiveness.

Also, improvements in RAM configurations, connectivity, pricing, faster storage and thermal and power management make the Pi 5 a robust standalone platform for the objective thesis tasks and its workloads.

3.3.5 NVIDIA Jetson Nano

Represents a departure in architectural design, selected to evaluate the performance impact of a dedicated AI-centric hardware platform. Unlike general-purpose single-board devices, the Jetson Nano integrates a CUDA-capable GPU within a compact form, enabling efficient inference workloads and serving as a benchmark for gauging the value of specialized AI hardware in comparison to more generic edge devices.

Specific data is summarized in the following chart:

Model	Jetson Nano B01 P3450 Developer Kit (945-13450-0000-100)
Launch date	12/03/2020
Processor	4 x ARM A57 @ 1.43 GHz Processor rev 1 (v8l)
Graphics card	NVIDIA Tegra X1 - 128 core Maxwell
RAM	4 GB 64-bit LPDDR4 25.6 GB/s
ROM	32GB (microSD)
OS	ubuntu 18.04 LTS (64-bit)
Python version	3.6 (64-bit)

Table 7 NVIDIA Jetson Nano specifications

Figure 11 NVIDIA Jetson Nano

Its design supports real-time execution of multiple neural networks in parallel, enabling high-resolution image processing within a small and energy-efficient package. The platform is supported by the JetPack SDK, which includes Ubuntu-based OS, CUDA Toolkit, cuDNN, and TensorRT, ensuring native compatibility with major frameworks such as TensorFlow or PyTorch.

Benchmark studies targeting edge devices confirm the Jetson Nano's substantial advantage in GPU-accelerated tasks, thus the interest to include it in the comparison pool.

3.3.6 Custom build PC

Represents a high-cost, high-performance workstation configured to provide an upper bound reference for computation capability. Selected to evaluate whether substantial investment in CPU and GPU yields proportionally greater value for biomedical image-classification tasks relative to edge and mid-range platforms.

Specific data is summarized in the following chart:

Model	Custom Build RTU
Launch date	Latest, GPU: 29/10/2020
Processor	Intel Core i9 - 10900X CPU @ 3.70 GHz, 3696 Mhz, 10 Cores, 20 Logical Processors
Graphics card	Nvidia GeForce RTX 3070
RAM	64GB
ROM	HDD Toshiba DT01ACA200 2 TB
OS	Windows 11 Pro
Python version	3.11.9 (64-bit)

Table 8 Custom build PC specifications

Figure 12 Custom build PC CPU & GPU

The custom PC is anchored by an Intel Core i9 processor built on the X-series (LGA2066 chipset), delivering robust multithreaded performance and support for advanced instruction sets such as AVX-512 and Deep Learning Boost. This CPU is designed for workstation-grade workloads, including intensive data processing and parallel model execution. The GPU, based on the Ampere architecture, offers excellent compute resources for both model inference, via CUDA or TensorRT.

This configuration serves as a performance ceiling within the experimental framework. It aids in quantifying cost-effectiveness by contrasting throughput, latency, and energy usage against edge solutions such as the ones presented, ensuring an informed evaluation of when and if such high-end investments bring meaningful return in biomedical image classification contexts.

3.4 Evaluation methodology

This section outlines the uniform protocol employed to assess both performance (inference speed) and classification accuracy of the image classification models across the selected hardware platforms. With the exception of the specialized biomedical test, which will be detailed later in that chapter, all evaluations follow an identical methodology to ensure comparability.

The procedure is as follows:

- *Selection of test images*

  3,000 images not used during training or fine-tuning are selected evenly across all available classes (e.g., CIFAR-10: 300 images per class). Selection may involve random sampling within a larger pool to maintain class balance.

- *Model inference execution*

  A Python script iterates over each image sequentially. Prior to processing each image, a timer from Python's 'time' library is initiated. After obtaining the top-1 prediction, the timer stops, and the elapsed interval is accumulated into a global variable (e.g., total_inference_time). The prediction is compared to the true class; correct predictions are counted and accumulated into another global variable (e.g., correct_predictions).

- *Result aggregation and reporting*

  Upon completion of all 3,000 inferences, the Python script prints:
  - Total inference time, in seconds, for the full image batch.
  - Number of correctly classified images.
  - Accuracy percentage, calculated as (correct / 3,000) x 100, meaning, for instance, 2,817 correct predictions correspond to approximately 93.90%.

- *Interpretation*

  Inference speed provides a quantitative baseline of throughput performance per hardware device. Also, Top-1 accuracy reflects effectiveness of model generalization on unseen data.

The methodology aligns with standard practices in model evaluation and benchmarking, measuring latency and throughput via per-image inference timing is common in deep learning performance studies [44].

Furthermore, the combined evaluation of accuracy and inference latency reflects multi-metric benchmarking approaches: recent edge-AI research emphasizes the importance of evaluating both speed and correctness for deployment feasibility [45].

3.5 Software and toolchain overview

This subchapter is aimed at introducing the principal software and toolchain utilized during the development of thesis and inside its many image inference tests, serving as the necessary informatic support for the correct and most effective deployment, version control, fine tuning for the models and dataset preparation.

3.5.1 Inference frameworks

This section details the primary Python packages employed for model inference across the chosen hardware platforms. All libraries are installed via 'pip', ensuring reproducibility and ease of deployment.

- *TensorFlow* (v2.18.0)

  Used on high-performance hardware (student's personal laptop, custom PC), TensorFlow 2.18 integrates the 'keras' subpackage for seamless import of pretrained models such as MobileNetV2 and supports accelerated GPU execution and enhanced numerical operations via improved CUDA support and compatibility with NumPy 2.0.

- *TensorFlow Lite*

  A lightweight version of TensorFlow optimized for resource-constrained platforms. Employed on devices like the Raspberry Pi Zero 2 W, it facilitates efficient model inference in constrained environments by using compact, quantized models tailored for edge execution.

- *OpenVINO™ Runtime*

  Utilized exclusively with the Intel Neural Compute Stick 2, this toolkit enables highly optimized inference on the Movidius Myriad X VPU. It also supports Python API deployment across CPU, GPU, and NPU targets. Installation is handled through 'pip' as well, in this thesis case within virtual environments, and immediate device detection and inference are validated using simple Python instructions.

- *ONNX Runtime*

  Selected in cases where other runtimes failed to execute, the quick and easy conversion to ONNX format makes it a fundamental framework in this thesis. ONNX Runtime allows execution of exported ONNX models with efficient utilization of GPU cores, aligning well, for example, with the Nano's architectural strengths.

- *PyTorch*

  A dynamic, imperative deep-learning framework valued for its intuitive interface and ease of debugging. Selected across multiple hardware platforms and inference tests, PyTorch supports rapid development, flexible model architectures, and hardware acceleration via CUDA on GPU-equipped systems.

- *FastAI*

  Built atop PyTorch, FastAI introduces higher-level abstractions and polished APIs for rapid prototyping in domains such as vision, tabular data, and more. Its layered design, catering both beginner-friendly workflows and low-level customization, enables efficient training, data handling (via DataBlock API), and fine-tuning of models in fewer lines of code, thus streamlining development without sacrificing performance.

3.5.2 Programming IDE and version control system

This subsection presents the primary development environment and the version control practices adopted in the project. The objective is to document how code was authored, organized, and synchronized across devices in a reproducible and reliable manner.

On the student's personal computer, the main IDE was *Visual Studio Code* (VS Code). VS Code natively integrates Git through its Source Control view, enabling commit, branch, pull, push, and merge operations without leaving the editor. Projects are opened as folders in a main workspace, and typically one folder represents one experiment, which simplifies structured layouts and multi-folder scenarios when needed. This integration allows a smooth workflow with a GitHub-hosted repository, including status, diffs, staging, and commit history from within the IDE.

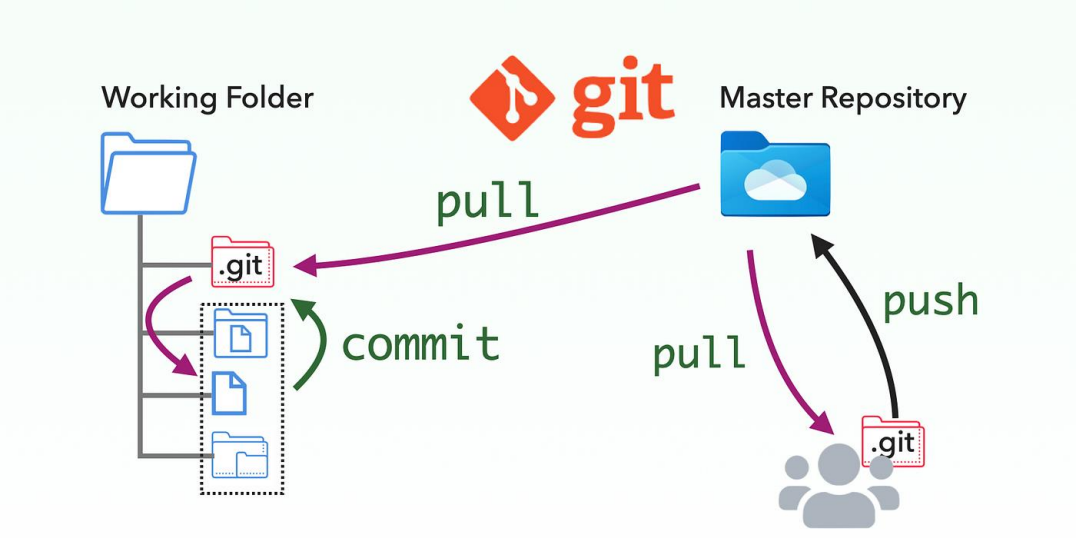

Figure 13 Git version control scheme

The repository is hosted under the student's GitHub account and accessed from VS Code's Source Control interface. Standard Git operations remain available in the terminal or the GUI; cloning or pulling updates the local copy, while staging and committing track changes before pushing to the remote. This approach follows the canonical Git workflow and ensures recoverability and collaboration if needed. Folder structure mirrors the experimental design, with a top-level directory per main image-classification test to keep datasets, models, and scripts scoped and discoverable.

On the remaining edge devices and hosts without a graphical interface, VS Code is not used. These systems are reached over SSH using *PuTTY*, a widely used Windows SSH client and terminal emulator. Editing is performed directly on the target using *Vim*, a modal text editor available by default in most Unix-like systems. Version control is handled with the command-line *git* tool, which provides the same functionality as the VS Code integration, albeit with a steeper learning curve for newcomers. This headless workflow is standard in remote Linux administration and software deployment.

In practice, the command-line workflow on these devices follows a simple cycle: update or obtain the repository, edit files, stage changes, commit, and push back to the remote. This sequence maps to *git pull* or *git clone*, followed by *git add*, *git commit*, and *git push*, which are the essential operations used to synchronize code across machines and maintain a coherent history.

Finally, the choice of editors and tooling reflects the deployment realities of edge devices. A GUI-centric workflow on the development laptop accelerates authoring and review, while SSH-plus-Vim ensures low-overhead editing and immediate access on devices that run without a desktop environment, aligning with best practices for remote work on embedded Linux systems.

CHAPTER 4
An introductory test

This chapter reports a preliminary experiment designed to validate the full inference pipeline across the selected hardware prior to more ambitious evaluations. The test performs image classification over a set of 3,000 images drawn from the Tiny ImageNet dataset. This is a down-scaled derivative of ImageNet with 200 classes and 64x64 images, widely used for rapid prototyping and method comparison.

The selected model is MobileNetV2, employed in its pretrained configuration for ImageNet without additional fine-tuning.

The objective of this introductory test is not to maximize accuracy or throughput per device but to establish a reliable baseline for code structure, portability, and model conversion across runtimes. The same Python driver executes single-image, sequential inference and records top-1 predictions, while inference latencies are captured per image using a timer and aggregated for reporting, as defined in subchapter 3.4. Conversion and deployment paths are exercised where appropriate and will be detailed later in this chapter.

4.1 Dataset & model acquisition & understanding

The test employs the *Tiny ImageNet* dataset, obtained directly from the official ImageNet website after registering a research account. The complete Tiny ImageNet distribution comprises 100,000 JPEG training images at 64x64 resolution across 200 classes, together with validation and test partitions. It is a down-scaled, class-reduced subset of ImageNet, sampling object categories from the WordNet-based hierarchy and covering broad themes such as animals, vehicles, tools and household objects. This choice reduces storage, memory and compute requirements relative to 224x224, 1,000-class ImageNet, which makes it suitable for constrained edge devices while retaining semantic diversity for meaningful benchmarking. Access instructions and downloads are provided on the ImageNet site, and the dataset specification is documented in the CS231n Tiny ImageNet challenge materials [46].

Recent work continues to use Tiny ImageNet as a relevant benchmark for evaluating efficiency and accuracy trade-offs in modern architectures, which supports its use in the present study [47].

Figure 14 Sample of the Tiny Imagenet dataset

The model is *MobileNetV2* in its pretrained ImageNet configuration, selected for fast, reproducible deployment of inference pipelines without fine-tuning. On the student's main device, MobileNetV2 is instantiated through Keras library within TensorFlow, which provides ready-to-use weights trained on ImageNet and downloads them automatically at first use. This enables almost immediate top-1 classification on the 3,000 held-out Tiny ImageNet images, with the driver code handling sequential inference and timing. The same pretrained model is then converted to the rest of devices and discussed in later chapter sections.

4.2 Code & performance analysis

The subchapter details the device-specific inference process, covering code-structure highlights, model-conversion paths, and per-device results. Subchapters for the different hardware and edge devices describe the Python driver's particularities, then report accuracy and timing metrics. The procedure, following subchapter 3.4, aims at a standard inference-benchmarking practice, prioritizing consistent measurement of per-image latency and throughput for fair cross-device comparison. Following this subchapter, a closing synthesis compares aggregate outcomes across platforms to contextualize performance-cost trade-offs.

4.2.1 MSI Modern 14

The inference carried on this hardware begins with three unoptimized Python code files. In this subchapter, the fourth and most optimized version of the code will be covered, understanding its particularities.

The script implements a clear, CPU-only inference pipeline. It first disables GPU discovery by setting `CUDA_VISIBLE_DEVICES='-1'`, ensuring TensorFlow performs inference on the CPU, which aligns with the hardware profile of the laptop.

The MobileNetV2 network with ImageNet weights is then instantiated via Keras Applications, which automatically retrieves and caches pretrained weights, enabling immediate use without auxiliary setup.

The validation set consists of 3,000 Tiny ImageNet images stored in a flat folder; a label file is loaded so that predictions can be checked against class identifiers. To ensure deterministic traversal, filenames are iterated in natural-sorted order.

MobileNetV2 > PC_validation_imgs_correct_preds.txt

1	val_0.JPEG	n03444034	0	32	44	62
2	val_1.JPEG	n04067472	52	55	57	59
3	val_2.JPEG	n04070727	4	0	60	55
4	val_3.JPEG	n02808440	3	3	63	63
5	val_4.JPEG	n02808440	9	27	63	48
6	val_5.JPEG	n04399382	7	0	59	63
7	val_6.JPEG	n04179913	0	0	63	56
8	val_7.JPEG	n02823428	5	0	57	63
9	val_8.JPEG	n04146614	0	31	60	60
10	val_9.JPEG	n02226429	0	3	63	57
11	val_10.JPEG	n04371430	37	38	44	45
12	val_11.JPEG	n07753592	0	1	63	47
13	val_12.JPEG	n02226429	5	12	57	53
14	val_13.JPEG	n03770439	21	33	36	43

Figure 15 Sample of the label file containing file names and synsets

Each image is read with OpenCV, resized to 224x224, cast to 32-bit floating point, normalized to the [0, 1] range, and expanded with a batch dimension to produce a tensor of shape (1, 224, 224, 3).

For each sample, the code starts a timer with `time.perf_counter()`, calls `model.predict` on the prepared tensor, and immediately stops the timer, accumulating the elapsed duration into `total_inference_time`. This timer is suited to short-interval benchmarking in Python and avoids epoch-based clock issues.

The raw logits are converted to human-readable top-1 outputs using `tf.keras.applications.mobilenet_v2.decode_predictions`, which returns the predicted ImageNet synset and description. The script compares the synset identifier with the ground truth and increments a counter when they match.

Resource usage, added to this introductory test as an extra metric, is sampled via `psutil.cpu_percent` near the beginning and end of execution to provide a coarse estimate of average CPU load that is independent from the per-image timing loop. After processing all images, the script prints the total accumulated inference time in seconds, the number of correct predictions, and the corresponding accuracy percentage over 3,000 samples, offering a concise summary of throughput and correctness for this device.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy	CPU usage
245.35 s.	279/3,000	9.30 %	10.80 %

Table 9 MSI Modern 14 average results

Results were modest. The very low top-1 accuracy is possibly explained by an input preprocessing mismatch: images were read in OpenCV's BGR order and scaled to [0,1], rather than MobileNetV2's expected RGB with `preprocess_input` scaling to [−1,1], which typically degrades predictions [48]. In addition, Tiny ImageNet's 64x64 imagery upscaled to 224x224 introduces a pronounced resolution and domain shift relative to ImageNet, further reducing transfer accuracy.

4.2.2 Raspberry Pi Zero 2 W

The script aimed to this edge device mirrors the one commented before but adapted to the TensorFlow Lite framework.

This adaptation is done in a separate Python script. The conversion occurs with the execution of the following instructions:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

converter.target_spec.supported_types = [tf.float16]
```

Here, MobileNetV2 (`model`) is converted, optimized and reduced in order to work in an optimized way in the edge device. Then, it is saved to the file system as a '.tflite' file.

Back to the inference on this device, and as on the last hardware, it iterates deterministically over the 3,000 validation images, applies per-image preprocessing, measures latency with `time.perf_counter`, accumulates `total_inference_time`, and tracks correctness against a ground-truth mapping loaded from the Tiny ImageNet validation labels file. The validation split of Tiny ImageNet is flat and requires an external mapping, which the script reads into a dictionary before inference, in a slightly different way than before.

The key difference is, as commented in the beginning, the inference backend. Instead of `tf.keras` and `model.predict`, the Raspberry Pi uses the lightweight TFLite Python interpreter: the model is loaded once, `allocate_tensors()` prepares memory, and each sample is processed by copying the input into the model's input tensor, invoking `interpreter.invoke()`, and reading the output tensor. Output handling also differs. On the MSI path, predictions are decoded to ImageNet synsets with `decode_predictions`, then compared to Tiny ImageNet labels. On the Raspberry Pi, the script computes `argmax` directly on the output vector and compares the resulting class index with the integer label from a modified label file, which avoids the synset decoding step and reduces overhead on the host.

MobileNetV2 > IntelNCS2_RPZ2W_validation_imgs_correct_preds.txt

```
1   val_0.JPEG  573 0   32  44  62
2   val_1.JPEG  758 52  55  57  59
3   val_2.JPEG  760 4   0   60  55
4   val_3.JPEG  435 3   3   63  63
5   val_4.JPEG  435 9   27  63  48
6   val_5.JPEG  850 7   0   59  63
7   val_6.JPEG  786 0   0   63  56
8   val_7.JPEG  440 5   0   57  63
9   val_8.JPEG  779 0   31  60  60
10  val_9.JPEG  311 0   3   63  57
11  val_10.JPEG 842 37  38  44  45
12  val_11.JPEG 954 0   1   63  47
13  val_12.JPEG 311 5   12  57  53
14  val_13.JPEG 655 21  33  36  43
```

Figure 16 Sample of the modified label file for the edge devices

The rest of the script, including image processing, is kept identical.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy	CPU usage
549.90 s.	282/3,000	9.40 %	0.25 %

Table 10 Raspberry Pi Zero 2 W average results

An even slower total inference time is obtained, as expected due to the big decrease in computing power. Accuracy remains almost the same, increasing a bit possibly due to the optimization and reduction of the model during the conversion of the model step detailed earlier.

4.2.3 Raspberry Pi Zero 2 W & Intel Neural Compute Stick 2

The pipeline preserves the evaluation logic of the first two scripts presented earlier. Deterministic listing of the 3,000 validation images, per-image preprocessing to 224x224 float32 in [0,1], high-resolution timing with `time.perf_counter`, accumulation in `total_inference_time`, and accuracy computed against a filename-to-label mapping.

The key divergence is the inference backend: the RPZ2W acts only as host while the Intel NCS2 performs inference on its Movidius Myriad X VPU. The script initializes the OpenVINO Runtime (`ov.Core()`), loads an IR model with `core.read_model()`, then compiles it for the MYRIAD target using `core.compile_model(model, device_name="MYRIAD")`. Device naming and selection follow OpenVINO conventions, where MYRIAD denotes the NCS2 VPU; the call returns a `CompiledModel` executed by the

stick. Outputs are read by indexing the compiled model with the prepared input tensor and selecting the top-1 class via `argmax`, avoiding any synset decoding overhead.

This version replaces `model.predict()` or `interpreter.invoke()` with OpenVINO's compiled-model invocation, which offloads compute to the VPU while the Pi handles I/O, preprocessing, and orchestration. The structure of the loop, timing, and accuracy accounting remains identical, enabling fair cross-device comparison with minimal code divergence.

Obtaining a valid IR format model is not as straightforward as it can seem. Detailed process is as follows:

- *Exporting Keras model to HDF5 (.h5)*.
  On a TensorFlow-capable host, MobileNetV2 with ImageNet weights is instantiated and saved with `model.save("mobilenetv2.h5")`.

- *Producing a TensorFlow SavedModel.*
  The HDF5 model is loaded in a new Python script and exported with `tf.saved_model.save(model, "mobilenetv2_savedmodel")`. SavedModel is a directory with saved_model.pb and variables, suitable for downstream conversion.

- *Converting to OpenVINO IR*.
  The Model Optimizer command is used to generate the IR model from the SavedModel directory, specifying the input shape in batch, height, width, channels (NHWC) in order to fit MobileNet-style classifiers. The command is:
  ```
  mo --saved_model_dir C:\...\mobilenetv2_savedmodel --output_dir C:\...\model_ir --input_shape [1,224,224,3] --framework tf --use_legacy_frontend
  ```

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy	CPU usage
162.74 s.	277/3,000	9.23 %	0.30 %

Table 11 Intel Neural Compute Stick 2 average results

Even though the accuracy of the inference remains mostly the same, an incredible 3,4x time reduction is achieved in comparison to giving inference with the standalone Raspberry Pi. This demonstrates the capabilities of the Intel's VPU device in comparison to CPU-only execution.

4.3 Results

This introductory test accomplished its core objective: validating an end-to-end, portable inference pipeline and exercising model conversion across runtimes (TensorFlow/Keras, TFLite, OpenVINO IR). Tiny ImageNet's 200 classes at 64x64 enabled a fast, semantically diverse shakedown before larger experiments, confirming correct deployment on each device.

Measured latencies align with architectural expectations. The Intel configuration was the fastest overall, with a total of 162.74s for 3,000 images, about 54ms per image, roughly 18 frames-per-second (fps). The MSI CPU baseline required 245.35s, about 82ms per image, close to 12 fps. The Raspberry Pi's CPU-only path was slowest at 549.90s, about 183ms per image, near 5.5 fps. The acceleration observed with the NCS2 reflects offloading convolutional workloads to the Myriad X VPU via OpenVINO; a device specifically designed for low-power DNN inference.

Top-1 accuracy clustered around 9 % on all three paths. Two factors most plausibly explain this: a preprocessing mismatch against MobileNetV2's expected RGB input scaled to [−1, 1], and a pronounced train–test resolution gap when upscaling 64x64 Tiny ImageNet images to 224x224 for an ImageNet-pretrained model. Both tend to depress transfer accuracy unless addressed with proper normalization and task-specific adaptation.

	Total inference time	Correctly classified images	Accuracy	CPU usage	FPS
MSI Modern 14	245.35 s.	279/3,000	9.30 %	10.80 %	12
RPZ2W	549.90 s.	282/3,000	9.40 %	0.25 %	5.5
Intel NCS2	162.74 s.	277/3,000	9.23 %	0.30 %	18

Table 12 Complete results for the introductory test

In subsequent chapters, the evaluation will explicitly target higher precision and speed by (i) enforcing framework-consistent preprocessing, (ii) refining conversion pipelines, and (iii) applying fine-tuning to mitigate the resolution/domain shift, thereby aiming for materially improved accuracy at lower per-image latency on each hardware class.

CHAPTER 5
Benchmarking fine-tuned models

This chapter advances the study beyond the introductory pipeline check by introducing fine-tuning on a new dataset and expanding the set of evaluated platforms. The objective is to compare heterogeneous devices under identical experimental conditions, using standardized preprocessing and common metrics for accuracy, latency, and cost, enabling a principled assessment of cost-effectiveness and practical suitability. The evaluation logic aligns with established benchmarking practice that treats latency and accuracy as co-equal constraints for deployment [49].

Methodologically, the chapter operationalizes *transfer learning*: pretrained vision backbones are fine-tuned to the new task to reduce domain and resolution gaps observed in Chapter 4. Prior work shows that models with stronger ImageNet performance tend to transfer better across downstream datasets, and that careful adaptation improves effectiveness over fixed-feature baselines, motivating the fine-tuning strategy adopted here [50].

Finally, a dedicated subchapter will synthesize all device-level outcomes into a ranking. The scheme aggregates, per device, the average market price (Q3 2025) and, for each of the three fine-tuned models, the top-1 accuracy and the total inference time. This multi-metric view follows established benchmarking practices that balance accuracy, latency and monetary cost to assess deployment suitability on edge platforms.

5.1 Dataset acquisition & models fine-tuning

To progress beyond the introductory pipeline check, the study pivots from Tiny ImageNet, local based deployment to a smaller, domain-agnostic and online workflow that enables even quicker fine-tuning cycles and methodical validation, before moving to biomedical data.

The rationale is pragmatic: establish robust training and deployment routines on an accessible corpus whose size and resolution reduce computational burden across heterogeneous devices, while preserving enough semantic diversity to stress the end-to-end workflow.

In this subchapter, all the details about the dataset acquirement, processing and deployment to the fine-tuning environment, as well as the detailed fine-tuning workflows, will be covered.

5.1.1 CIFAR-10

CIFAR-10 meets the requirements. It's a dataset that contains 60,000 color images at 32x32 resolution partitioned into 50,000 training and 10,000 test samples, evenly distributed across 10 classes. The categories span common objects and animals, providing a balanced set of images that is well suited for controlled transfer-learning experiments and repeatable cross-device comparisons. Typical class labels include *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*.

Figure 17 Sample of the CIFAR-10 dataset

CIFAR-10 is distributed by the University of Toronto and originates from the Tiny Images effort. It remains a canonical baseline in vision research and pedagogy, precisely because its low resolution enables fast experimentation while remaining challenging enough to differentiate models and optimization choices. Using CIFAR-10 here supports efficient ablations on preprocessing, augmentation, and fine-tuning schedules that later transfer to the biomedical setting.

To streamline acquisition, storage, and preparation, the dataset was sourced and managed through *Roboflow Universe*. *Universe* is a large, public repository where institutions and users publish datasets and pretrained models; projects can be inherited into a personal workspace, enabling reuse with full control over subsequent processing and deployment. In this study, the student inherited a public project from Universe and created a private working copy in their own workspace, establishing a clean starting point for experimentation and versioning.

Universe datasets can be downloaded or exported for local training, but here the emphasis was on keeping the corpus online to simplify iteration and provenance tracking. Roboflow's platform supports export to common ML formats and provides tooling to integrate with external training stacks if desired, preserving flexibility while keeping dataset management centralized.

But, in this study, the focus of Roboflow remains only on the scope of dataset management and version deployment to other fine-tuning platforms, reducing the overhead of local data wrangling and allowing effort to focus on fine-tuning and deployment in the subsequent subchapters.

5.1.2 MobileNetV2

MobileNetV2 is fine-tuned following the procedure outlined in the [51] reference, an entry in the Roboflow blog aimed at providing a complete guide to fine-tune the specific model targeted in this experiment. This source provides a clear, end-to-end recipe for adapting an ImageNet-pretrained MobileNetV2 to a new classification task, and is adopted here to ensure a reproducible, standards-aligned workflow from dataset export to model adaptation. The blog entry also gives general tips for applying transfer learning concepts to other models training procedures, so in the subsequent subchapters, this blog entry will continue to be the reference point.

Fine-tuning code execution is carried out in *Google Colab*, which supplies a ready-to-use notebook environment with on-demand GPU acceleration. Colab's managed runtimes and straightforward GPU enablement make it suitable for rapid experimentation and iterative refinement without local setup overhead, while preserving the ability to share and archive notebooks for documentation and reproduction of the thesis's experiments. Since the original guide notebook utilizes and is aimed to use a dataset stored in Roboflow as the source for the training images, the setup described in the prior subchapter perfectly accomplishes the needs of this fine-tuning procedure.

Inside the Google Colab environment, the GPU instance should be initiated before proceeding to the code execution, that way ensuring top-speed training and fine-tuning of the model.

The pipeline begins by loading the Roboflow-exported dataset in an ImageFolder layout, constructing an input pipeline, and applying MobileNetV2's required preprocessing (RGB scaled to [−1, 1], resize to 224x224). A MobileNetV2 backbone with ImageNet weights is instantiated and a compact classification head for ten classes is added.

The initial phase freezes the convolutional base (`base_model.trainable = False`) and trains only the new head using Adam with the base learning rate set in the notebook (`model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001)…)`) for the defined number of epochs (`initial_epochs = 20`).

Training is initiated with the ~2.2M parameters of the model frozen, but ~12,000 trainable parameters in the Dense layer. These are divided between two `tf.Variable` objects, the weights and biases.

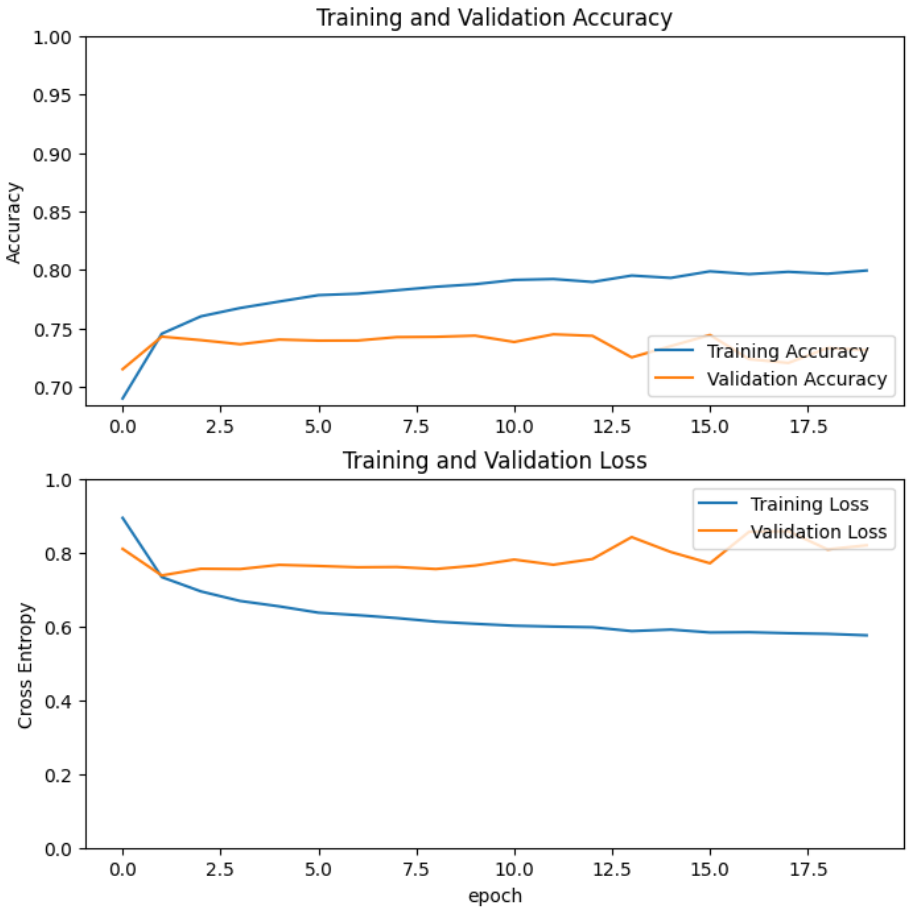

Figure 18 MobileNetV2 Training & Validation accuracy & loss

Training and validation curves show accuracy climbing rapidly from chance and loss decreasing, which indicates that the new head is learning discriminative features while the base remains stable, exactly the pattern expected in Keras transfer learning [48].

Now, the fine-tuning part of the code will update the weights of the top layers of the pre-trained MobileNetV2 in ImageNet. This will improve the generic feature maps of the model to be associated with the CIFAR-10 dataset. By doing this only in the top layers of the model, thus the most specialized ones, the model will conserve the generic learning of the bottom layers while learning only the most specific features of the new dataset.

The method consists, as mentioned, of unfreezing upper layers of the backbone (`fine_tune_at = 100; base_model.trainable = True; for layer in base_model.layers[:fine_tune_at]: layer.trainable = False`) and re-compiling with a reduced learning rate (commonly an order of magnitude lower, `tf.keras.optimizers.Adam(learning_rate=base_lr/10)`), then continuing for additional fine-tune epochs (`fine_tune_epochs = 10; model.fit(...)`).

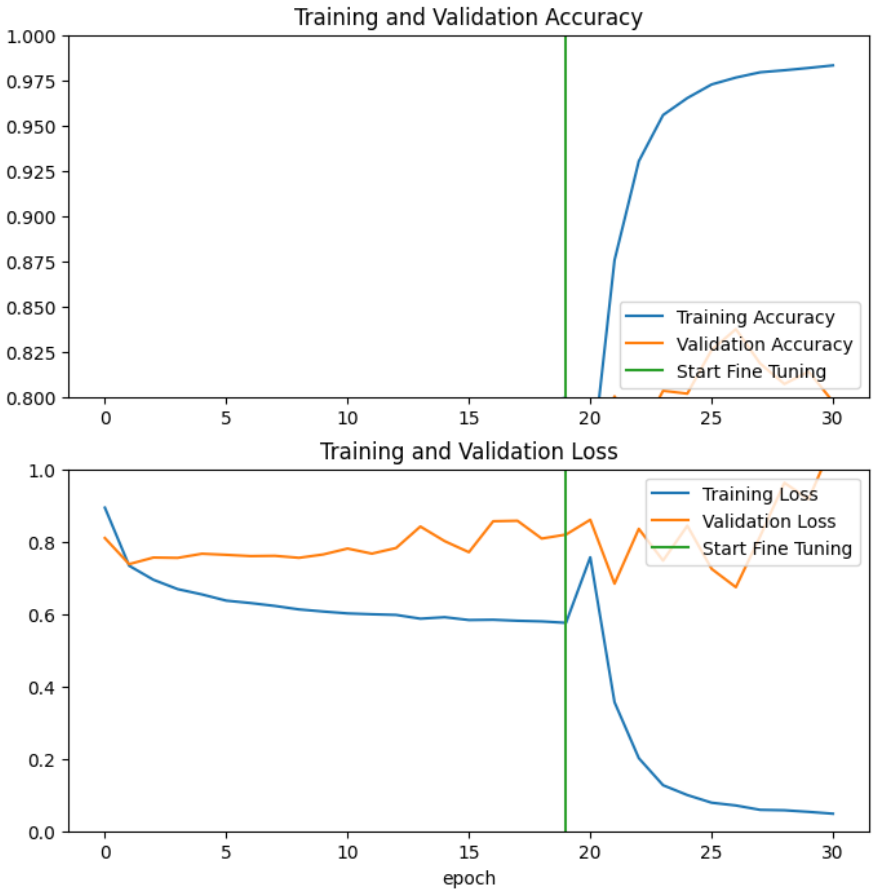

Figure 19 MobileNetV2 Training & Validation accuracy & loss after fine-tuning

As expected, a significant improvement in training accuracy is achieved, up to practical perfection. Validation accuracy also improves and seems to stabilize at around 80 %. Training loss decreases also greatly, and validation loss stabilizes at around 0.8.

Crucially, the trained model is exported to portable formats (`tf.saved_model.save(...)` and `model.save("... .h5")`) to enable downstream conversion and deployment on the heterogeneous devices evaluated later. These steps, code patterns, and curve interpretations align with established Keras guidance for transfer learning and fine-tuning [48].

5.1.3 ResNet-18

For ResNet-18, the approach mirrors the MobileNetV2 setup but relies on Roboflow's ResNet-32 model page and its companion Colab notebook as a practical starting point [52]. Although the page targets ResNet-32, the workflow is readily adapted to ResNet-18 by selecting the corresponding backbone in the notebook. This choice maintains a consistent transfer-learning recipe while switching to a residual architecture whose skip connections ease optimization and are well established on benchmarks like CIFAR-10. The Roboflow materials provide the

scaffolding for dataset intake and training loops, while the academic foundation comes from the original residual networks work.

The notebook is implemented with *FastAI*, which streamlines computer-vision transfer learning through high-level learners and sensible defaults, reducing boilerplate and accelerating iteration. Continuing with the training methodology presented earlier, executing in Google Colab supplies a managed environment with on-demand GPUs, so every process can proceed without local setup. Together, these resources allow the thesis to obtain a comparable, fine-tuned classifier for CIFAR-10 under the same experimental controls used elsewhere in the chapter. This occasion, keeping CIFAR-10 at its native 32x32 resolution, avoiding any resizing stage. Because the Roboflow export grouped all 60,000 images under "train" for convenience, the split is created in code with

```
ImageDataLoaders.from_folder(path, valid_pct=0.2, bs=bs,
batch_tfms=[Normalize.from_stats(*imagenet_stats)]),
```

which performs a random 80/20 partition and applies ImageNet-style normalization at batch time. This preserves the dataset's original scale and mirrors common residual-network practice on CIFAR-10, while ensuring a reproducible validation protocol within fastai's dataloader API.

The learning-rate finder is run (`learner.lr_find()`), its curve inspected (`learner.recorder.plot()`), and a suitable base rate selected before training, at a range between 0.001 and 0.01. This method reduces guesswork and speeds convergence.

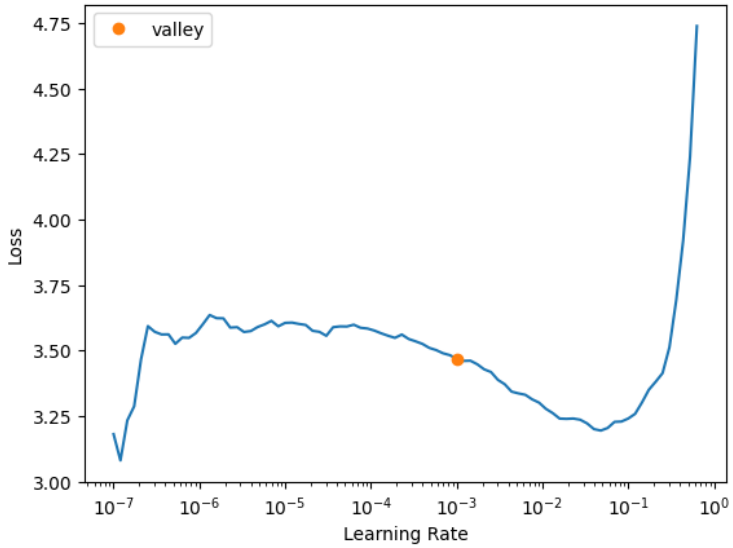

Figure 20 Learning rate curve and selected valley point for ResNet-18 training

A first training stage freezes the pretrained body and fits a newly attached head. Over 8 epochs, training proceeds from an initial `accuracy` ≈ 0.526 with `valid_loss` ≈ 1.34 to `accuracy` ≈ 0.752 with `valid_loss` ≈ 0.713, indicating steady, well-behaved optimization of the classifier head.

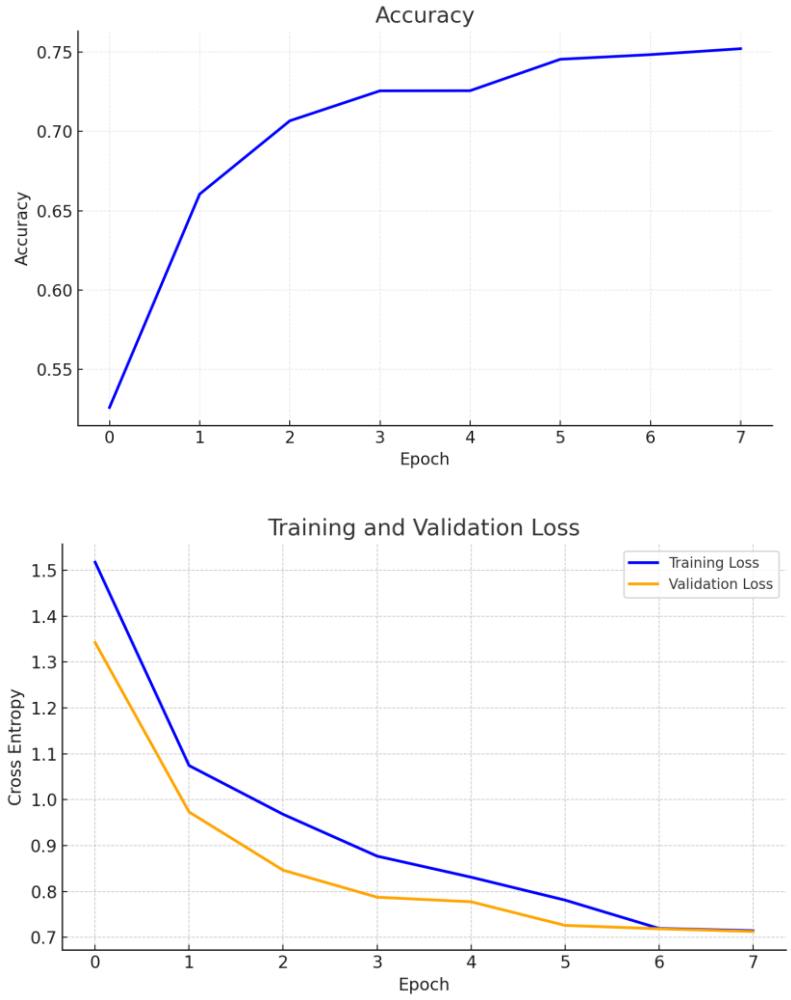

Figure 21 ResNet-18 accuracy & training & validation loss

The interim confusion analysis shows the most frequent confusions among visually similar categories: `('ship','truck', 36)`, `('airplane','truck', 30)`, `('automobile','airplane', 30)`, and `('dog','frog', 30)`, which is consistent with early-stage features still coarsely separating vehicles and certain animal classes.

Fine-tuning then unfreezes the upper portion of ResNet-18 and continues with a reduced learning rate (from 0.00001 to 0.0001) and a one-cycle schedule, refining higher-level representations while limiting destabilization of earlier layers.

Across the recorded run, accuracy improves further from ≈ 0.756 to ≈ 0.797, while validation loss decreases initially and later plateaus, signaling a point of diminishing returns and the onset of mild overfitting. This pattern is typical of transfer learning on compact datasets and supports the conclusion that the fine-tuned model is well adapted for CIFAR-10 within the constraints of the chosen schedule and regularization.

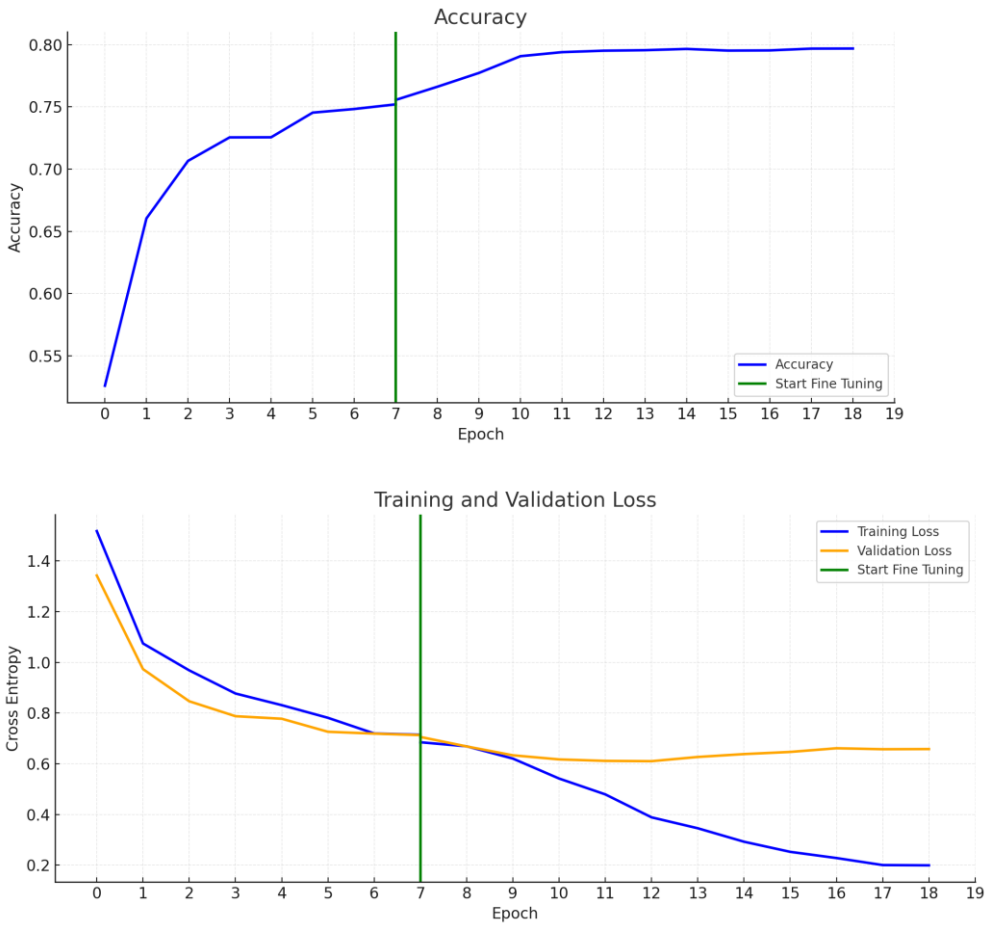

Figure 22 ResNet-18 accuracy & training & validation loss after fine-tuning

For deployment, the notebook saves artifacts in two complementary forms: a fastai Learner export (`learner.export('stage-1-bo')`) that encapsulates the preprocessing pipeline for straightforward loading, and a PyTorch model checkpoint (`torch.save(learner.model, 'stage-2-pytorch')`) suitable for downstream conversion or integration. These mechanisms align with recommended practices for packaging models for inference and further experimentation.

5.1.4 Yolov5s Classification

For YOLOv5s, the approach mirrors the previous two models: a Roboflow resource is used as the source for the training notebook, and the training itself is executed in Google Colab, quicker thanks to GPU acceleration. The training notebook for this model is developed and provided by the creator's company, Ultralytics [53].

The notebook follows Ultralytics' classification pipeline and differs from the MobileNetV2 and ResNet-18 notebooks in two main ways. First, training is orchestrated by the Ultralytics CLI and code in `classify/train.py`, which handles data loading, mixed-precision,

logging, checkpointing, and evaluation out of the box. A typical invocation specifies a pretrained checkpoint, the dataset, number of epochs, and image size, for example:

```
python classify/train.py --model yolov5s-cls.pt --data cifar10 --epochs 30 --img 128 --pretrained weights/yolov5s-cls.pt
```

This, apart from setting a resize to 128x128 for all images, leverages the built-in classification data loaders and evaluation hooks that report Top-1 and Top-5 accuracy each epoch. Second, the notebook logs GPU memory, AMP status, and saves metrics to the standard Ultralytics outputs. The deprecation warnings seen for `torch.cuda.amp.autocast` simply reflect PyTorch's migration to `torch.amp.autocast("cuda", ...)`, not an error in training.

The training is configured for 30 epochs, and overall it can be observed that Top-1 accuracy improves from 0.318 to 0.817, while Top-5 accuracy rises from 0.866 to 0.990. Loss steadily declines on both train and test splits, from roughly 2.05 down to 1.12 (train) and 1.92 drops to 0.916 (test). This trajectory indicates well-behaved convergence under the default schedule at 128x128 input resolution, and the final metrics align with the classifier's expected behavior in the Ultralytics framework.

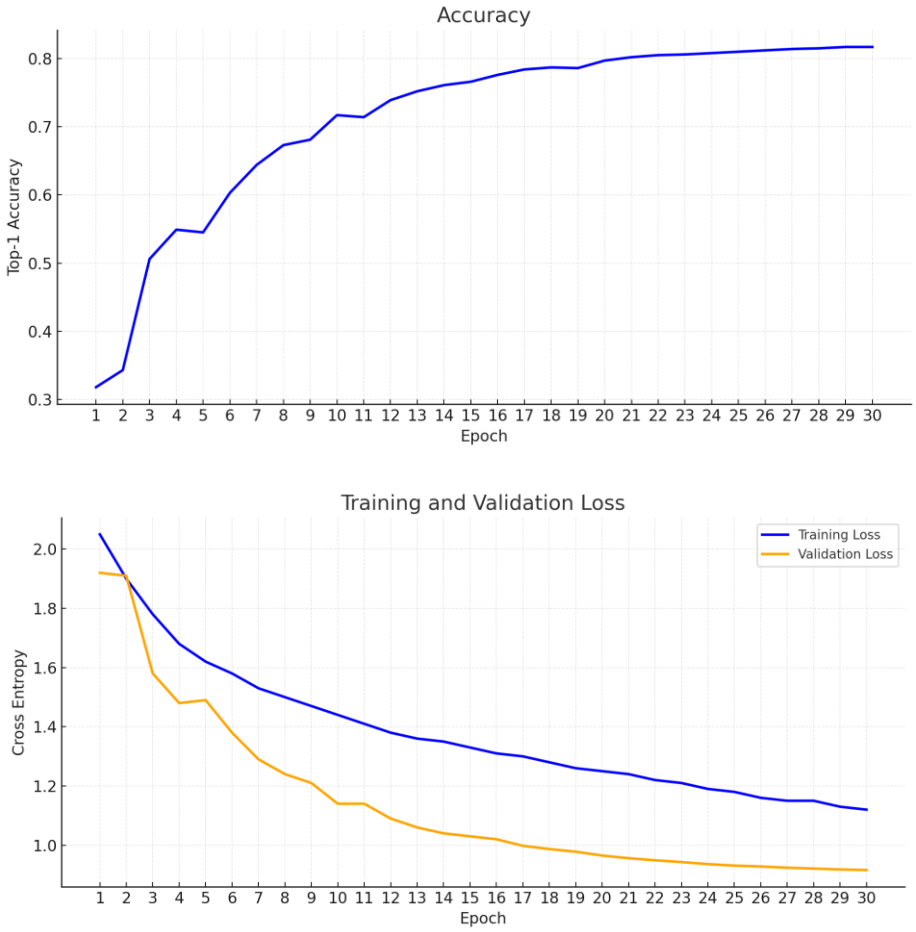

Figure 23 Yolov5s accuracy & training & validation loss

Unfortunately, and due to time constraints in the thesis development, the active learning part of the notebook, aimed at further improving in the training results, is left behind unused. These cells remain commented by default because the active learning methodology for the training of Yolov5s requires explicit project setup and safeguards: configuring the Roboflow Upload API with workspace credentials, selecting the target project/split, and defining a sampling rule (e.g., uploading low-confidence predictions) so that hard examples are programmatically added back to the dataset for later labeling and re-versioning.

Since only the weights of the trained are needed for further edge device deployment, a simple execution of the following code is enough to download everything needed:

```
model_path = 'runs/train-cls/exp2/weights/best.pt'

torch.save(model.state_dict(), model_path)
```

5.2 Inference with fine-tuned MobileNetV2

This subchapter details the device-specific inference process for the fine-tuned MobileNetV2, executed uniformly on 3,000 CIFAR-10 test images. For each device, the narrative highlights code-structure particularities and any conversion steps (e.g., TensorFlow Lite, OpenVINO IR, ONNX where applicable), then reports inference results and performance metrics. Timing follows a consistent per-image measurement to enable fair cross-device comparison and reflect MobileNetV2's design intent for efficient on-device inference.

Devices are, under the same model, assessed with identical preprocessing and single-image sequential inference, enabling like-for-like comparisons of throughput and correctness across heterogeneous runtimes. Each subsection includes a brief commentary interpreting the observed performance in light of architectural constraints. The resulting figures for MobileNetV2 will later feed into the cross-model ranking presented in subchapter 5.5.

5.2.1 MSI Modern 14

No model conversion is required: the Keras .h5 model format from Colab is loaded directly via `tf.keras.models.load_model(...)`, ensuring parity with the training graph.

The script builds a TFDS ImageFolder test set, applies MobileNetV2-consistent preprocessing (RGB scaled to [−1, 1]) with resizing to 224x224, and runs single-image inference while accumulating per-image latency. XLA JIT is enabled to stabilize throughput.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
210.00 s.	2415/3,000	80.50 %

Table 13 Average results for MSI Modern 14 & fine-tuned MobileNetV2

Average outcomes improved markedly versus Chapter 4: lower total time (245.35 s down to 210.0 s) and ~8.6x higher accuracy (9.30 % up to 80.50 %). Gains stem from aligned preprocessing for MobileNetV2 and transfer learning on CIFAR-10, which reduces domain and resolution mismatch observed earlier.

5.2.2 Raspberry Pi Zero 2 W

The Keras .h5 model is converted to TFLite with `TFLiteConverter`, enabling `Optimize.DEFAULT` and float16 weights to shrink the model for edge deployment, and on a host device like the Raspberry Pi Zero, this chiefly reduces size rather than latency.

Inference uses the lightweight `tflite_runtime.Interpreter`, with inputs resized to 224x224 and normalized to MobileNetV2's expected [−1,1] range.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
554.50 s.	2241/3,000	74.70 %

Table 14 Average results for Raspberry Pi Zero 2 W & fine-tuned MobileNetV2

Average outcomes over 3,000 images: 554.50 s total, 74.70 % top-1. Accuracy improves dramatically versus Chapter 4 (averaged at ≈9 %), reflecting correct preprocessing and fine-tuned weights; throughput remains CPU-bound on the Cortex-A53, hence similar runtime to the earlier RPZ2W test.

5.2.3 Raspberry Pi Zero 2 W & Intel Neural Compute Stick 2

The model is exported from Keras to SavedModel and converted to OpenVINO IR, then compiled for the MYRIAD target to run in FP16 on the NCS2's Myriad X VPU. A notable code particularity is the direct call of the compiled model (`result = compiled_model(image)[output_key]`), which leverages OpenVINO's callable CompiledModel to execute inference without managing an explicit InferRequest.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
160.30 s.	2,205/3,000	73.50 %

Table 15 Average results for Intel NCS2 & fine-tuned MobileNetV2

Results, as expected, have improved greatly from the Pi Zero standalone inference, achieving 160.30 s total, 73.50 % top-1, this time. The latency improvement over CPU hosts reflects VPU offload in FP16; the small accuracy gap versus MSI is consistent with backend and precision differences.

5.2.4 Raspberry Pi 5

Conversion mirrors RPZ2W: the Keras .h5 is exported to TFLite with `Optimize.DEFAULT` and float16 weights. Inference uses the lightweight `tflite_runtime.Interpreter`, with per-image normalization. images are randomly sampled via `Path.rglob` and labeled from folder names, keeping I/O simple and repeatable.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
103.50 s.	2,196/3,000	73.20 %

Table 16 Average results for Raspberry Pi 5 & fine-tuned MobileNetV2

The results show an improvement over the previous device, 103.50 s in total, 73.20 % in top-accuracy. The sharp latency gain over RPZ2W reflects the Pi 5's Cortex-A76 CPU uplift, while the accuracy gap to MSI's Keras baseline is consistent with TFLite/float16 differences, which primarily cut model size and may not accelerate CPU paths.

5.2.5 NVIDIA Jetson Nano

The Keras .h5 is frozen to a concrete graph and converted with tf2onnx directly to ONNX format using an additional python script. Then, inference process is executed after the usual preprocessing documented earlier. A particularly interesting detail in this version of the inference test is the use of provider fallback logic: by explicitly listing providers, the code attempts to use GPU acceleration, but gracefully degrades to CPU if unsupported operations are encountered. This makes the inference robust across different JetPack environments. Furthermore, session initialization occurs once outside the inference loop, reducing runtime overhead from tensor binding or engine creation.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
285.50 s.	2,184/3,000	72.80 %

Table 17 Average results for NVIDIA Jetson Nano & fine-tuned MobileNetV2

Average outcome shows 285.50 s total and 72.80 % top-1. The relatively higher latency versus Raspberry Pi 5 suggests that TensorRT acceleration may not have been fully utilized (graph partitioning or EP fallback can occur when ops/shapes are unsupported on TRT), leading to execution on CUDA or CPU.

5.2.6 Custom build PC

No model conversion is required due to the support of full Tensorflow Python libraries; the .h5 model is loaded directly using the same inference driver as the MSI. The script runs sequential, single-image inference using TFDS's ImageFolder loader, applies MobileNetV2-consistent preprocessing and captures per-image latency with `time.time()`.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
220.30 s.	2,424/3,000	80.80 %

Table 18 Average results for the custom build PC & fine-tuned MobileNetV2

Average outcome shows 220.30 s in total, 80.80 % for top-1 accuracy on 3,000 CIFAR-10 images.

Accuracy slightly improves over the MSI baseline (80.50 %), likely due to faster memory subsystems, elevated CPU clock speed, and additional available RAM supporting more efficient batching and caching. The inference time is modestly slower than MSI's 210 s, despite superior hardware, suggesting diminishing returns for general-purpose desktop CPUs with this specific setup and model.

This underscores that MobileNetV2's lightweight design achieves near-peak CPU performance even on modest systems, and that moving to infer-accelerator or edge GPUs offers minimal latency advantage over well-optimized CPU inference in the desktop tier.

5.3 Inference with fine-tuned ResNet-18

This subchapter mirrors the structure of subchapter 5.2 but replaces MobileNetV2 with the fine-tuned ResNet-18. As in subchapter 5.2, each device-specific subchapter comments briefly on the inference pipeline, any conversion steps required by the target runtime, and the per-device results on 3,000 CIFAR-10 test images sampled uniformly across classes.

5.3.1 MSI Modern 14

On the MSI Modern 14, the ResNet-18 checkpoint is used directly: the model saved with `torch.save(...)` during training is restored via `torch.load(..., map_location='cpu')` on the Python script, avoiding any format conversion.

FastAI's `ImageDataLoaders.from_folder` with `Normalize.from_stats(*imagenet_stats)` reconstructs the validation loader and enforces preprocessing consistent with the pretrained backbone.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
28.10 s.	2,814/3,000	93.80 %

Table 19 Average results for MSI Modern 14 & fine-tuned ResNet-18

Average outcome shows 28.10 s in total, 93.80 % for top-1 accuracy on 3,000 CIFAR-10 images. Inference running under `torch.no_grad()`, measuring per-image latency and accumulating total time for a fair, sequential comparison. So far, achieves both the quickest and most accurate inference output variables so far.

5.3.2 Raspberry Pi Zero 2 W

On the Raspberry Pi Zero 2 W, the fine-tuned ResNet-18 checkpoint is loaded directly with `torch.load(..., map_location='cpu')`, so no conversion is required. Preprocessing uses `torchvision.transforms.Normalize` with ImageNet statistics, and inference runs under `torch.no_grad()` to avoid autograd overhead during evaluation.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
289.15 s.	2,814/3,000	93.80 %

Table 20 Average results for Raspberry Pi Zero 2 W & fine-tuned ResNet-18

Despite matching the MSI's accuracy (2,814/3,000; 93.80%), total latency is 289.15 s, consistent with the RPZ2W's quad-core 1 GHz Cortex-A53 CPU constraints. Relative to MobileNetV2 on the same device (554.5 s; 74.7%), ResNet-18 is both faster and more accurate here, plausibly due to operating at CIFAR-10's native 32x32 whereas MobileNetV2 targets 224x224 inputs in this thesis' experiments.

5.3.3 Raspberry Pi Zero 2 W & Intel Neural Compute Stick 2

On the RPZ2W with Intel NCS2, the PyTorch ResNet-18 checkpoint is exported to ONNX via `torch.onnx.export` with a 1x3x32x32 dummy input. The ONNX graph is then loaded with OpenVINO Runtime (`ov.Core.read_model`) and compiled for the MYRIAD target, i.e., the Movidius Myriad-X VPU inside the NCS2. This avoids a prior IR conversion and enables USB-attached VPU offload from the ARM host. Preprocessing applies ImageNet mean and standard deviation for compatibility with the pretrained backbone.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
26.70 s.	2,796/3,000	93.20 %

Table 21 Average results for RPZ2W, Intel NCS2 & fine-tuned ResNet-18

The run yields 2,796 correct of 3,000 (93.20%) in 26.70 s total, faster than MSI inference with the same model (28.10 s) and far faster than RPZ2W standalone test (289.15 s) at comparable accuracy, underscoring VPU efficiency at 32x32 inputs.

5.3.4 Raspberry Pi 5

On Raspberry Pi 5, the fine-tuned ResNet-18 checkpoint is loaded directly in PyTorch; no conversion is applied. Preprocessing uses TorchVision's ImageNet normalization and retains CIFAR-10's native 32x32 resolution. Inference executes as single-image, sequential evaluation under `no_grad`.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
110.50 s.	2,814/3,000	93.80 %

Table 22 Average results for Raspberry Pi 5 & fine-tuned ResNet-18

The run yields 2,814/3,000 correct image classifications (93.80%) in 110.50 s total. Relative to the RPZ2W CPU, latency improves markedly, consistent with Raspberry Pi 5's quad-core Arm Cortex-A76 at 2.4 GHz, yet it remains slower than the NCS2 VPU and MSI CPU while matching their accuracy. These observations support the expectation that CPU microarchitecture uplift helps, but dedicated accelerators still dominate throughput at this input size.

5.3.5 NVIDIA Jetson Nano

On Jetson Nano, the fine-tuned ResNet-18 is exported to ONNX with opset 15 and executed via ONNX Runtime using a provider cascade, from TensorRT to CUDA and finally CPU, enabling TensorRT-accelerated subgraphs with automatic fallback. Preprocessing is kept the same, retaining 32x32 resolution and ImageNet normalization for all input images.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
72.50 s.	2,816/3,000	93.87 %

Table 23 Average results for NVIDIA Jetson Nano & fine-tuned ResNet-18

The run delivers 2,816/3,000 correct (93.87%) in 72.50 s total, a faster inference execution than the average of the Raspberry Pi 5 tests (110.50 s) and far ahead of Pi Zero 2 (289.15 s), yet slower than NCS2/MYRIAD (26.70 s) and MSI (28.10 s). This behavior aligns with the Nano's 128-core Maxwell GPU characteristics and software stack.

5.3.6 Custom build PC

On the custom PC, the fine-tuned ResNet-18 is used without conversion: the PyTorch checkpoint is restored via `torch.load(..., map_location='cpu')`, and inference runs under `torch.no_grad()` with batch size 1. Data loading mirrors the MSI path, applying ImageNet-style normalization to CIFAR-10 tensors to remain compatible with the transferred backbone.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
15.45 s.	2,823/3,000	94.10 %

Table 24 Average results for the custom build PC & fine-tuned ResNet-18

The run attains 2,823/3,000 correct (94.10%) in 15.45 s total, the best latency among all devices for this model, outpacing MSI (28.10 s) and NCS2/MYRIAD (26.70 s) at comparable accuracy. The result is consistent with the desktop-class Intel Core i9-10900X's higher clocks and wider headroom for single-image sequential inference.

5.4 Inference with fine-tuned Yolov5s Classification

This subchapter mirrors and aligns with the structure presented in both subchapters 5.2 & 5.3 but replaces the fine-tuned model with the Yolov5s Classification trained in Google Colab. As in the last subchapters, each device-specific section comments briefly on the inference pipeline, any conversion steps required by the target runtime, and the per-device results on 3,000 CIFAR-10 test images sampled uniformly across classes.

5.4.1 MSI Modern 14

On the MSI, the fine-tuned YOLOv5s-cls checkpoint is used directly in .pt format and loaded via PyTorch Hub (`torch.hub.load('ultralytics/yolov5', 'custom', path=...)`), requiring no conversion. The inference script applies a 128x128 resize and ImageNet-style normalization and evaluates sequentially with `torch.no_grad()` to suppress autograd overhead.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
25.30 s.	2,442/3,000	81.40 %

Table 25 Average results for MSI Modern 14 & fine-tuned Yolov5s Classification

On average, the execution yields 2,442/3,000 correct inferences (81.40%) in 25.30 s total. Relative to the same device running ResNet-18 (28.10 s; 93.80%), YOLOv5s-cls is a bit faster but notably less accurate, illustrating a speed–accuracy trade-off that will be revisited in subchapter 5.5 when aggregating cross-model, cross-device outcomes.

5.4.2 Raspberry Pi Zero 2 W

On Raspberry Pi Zero 2 W, the fine-tuned YOLOv5s-cls checkpoint (.pt) is used directly and loaded via PyTorch Hub, avoiding any conversion pipeline. Preprocessing also follows the 128x128 resize for all input images and ImageNet-style normalization. Inference proceeds sequentially under `torch.no_grad()`.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
637.30 s.	2,430/3,000	81.00 %

Table 26 Average results for Raspberry Pi Zero 2 W & fine-tuned Yolov5s Classification

Usually, the run attains 2,430/3,000 correct inference on top-1 (81.00%) in 637.30 s total. Compared with the MSI result (25.30 s; 81.40%), the large latency gap reflects the Zero 2 W's quad-core Cortex-A53 at 1 GHz and 512 MB RAM constraints. Relative to ResNet-18 on the same device (289.15 s; 93.80%), YOLOv5s-cls is less accurate and over 2x slower, while it remains far slower than the RPZ2W+NCS2 VPU path (26.70 s; 93.20%).

5.4.3 Raspberry Pi Zero 2 W & Intel Neural Compute Stick 2

On RPZ2W with Intel NCS2, the YOLOv5s-cls checkpoint is exported to ONNX using `torch.onnx.export` with a 1x3x128x128 dummy input to match the trained resolution, ensuring shape-consistent execution. The ONNX graph is loaded with OpenVINO Runtime and compiled for the MYRIAD target, offloading inference to the Movidius Myriad-X VPU. Preprocessing applies a 128x128 resize and ImageNet statistics to remain compatible with pretrained backbones.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
59.00 s.	2,466/3,000	82.21 %

Table 27 Average results for RPZ2W, Intel NCS2 & fine-tuned Yolov5s Classification

The run produces 2,466/3,000 correct (82.21%) in 59.00 s, substantially faster than RPZ2W CPU (637.30 s; 81.00%) and approaching desktop CPU efficiency, though still slower than MSI (25.30 s; 81.40%). Compared to ResNet-18 on the same VPU (26.70 s; 93.20%), YOLOv5s-cls is less accurate and slower, reflecting architectural and input-size differences.

5.4.4 Raspberry Pi 5

On Raspberry Pi 5, the fine-tuned YOLOv5s checkpoint (.pt) is loaded directly via PyTorch Hub, requiring no conversion; inference runs sequentially under `no_grad`. Preprocessing also aligns with the requirements of the model and Ultralytics classification pipeline.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
121.00 s.	2,475/3,000	82.50 %

Table 28 Average results for Raspberry Pi 5 & fine-tuned Yolov5s Classification

The quad-core Cortex-A76 at 2.4 GHz explains the substantial latency drop versus the RPZ2W's older Cortex-A53. The run attains 2,475/3,000 correct inferences (82.50%) in 121.00 s, a much faster result than the same test on RPZ2W (637.30 s; 81.00%), yet slower than NCS2 offload (59.00 s; 82.21%) and MSI (25.30 s; 81.40%). Relative to ResNet-18 on Pi 5 (110.50 s; 93.80%), YOLOv5s is slightly slower and notably less accurate, underscoring a speed-accuracy trade-off for subchapter 5.5.

5.4.5 NVIDIA Jetson Nano

On Jetson Nano, the fine-tuned YOLOv5s-cls checkpoint is exported to ONNX (with opset 15) using `torch.onnx.export` in a specific conversion Python script and executed via ONNX Runtime using a provider cascade `[TensorRT, CUDA, CPU]`, enabling TensorRT-accelerated subgraphs with automatic fallback. Preprocessing continues to be aligned with the requirements of the model and Ultralytics classification pipeline.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
130.69 s.	2,441/3,000	81.37 %

Table 29 Average results for NVIDIA Jetson Nano & fine-tuned Yolov5s Classification

Given the Nano's 128-core Maxwell GPU, the pipeline achieves moderate acceleration: 2,441/3,000 correct classification inferences (81.37%) in 130.69 s, therefore slower than MSI (25.30 s) and RPZ2W+NCS2 (59.00 s), faster than RPZ2W standalone (637.30 s), and slightly behind Raspberry Pi 5 test (121.00 s) at similar accuracy.

5.4.6 Custom build PC

On the custom PC, the fine-tuned YOLOv5s checkpoint is used directly in .pt format via PyTorch Hub (`torch.hub.load('ultralytics/yolov5','custom', path=…)`), so no conversion is applied. Preprocessing continues to be aligned with the requirements of the model and Ultralytics classification pipeline.

A typical execution of the Python script obtains the following results:

Total inference time	Correctly classified images	Accuracy
22.40 s.	2,385/3,000	79.50 %

Table 30 Average results for the custom build PC & fine-tuned Yolov5s Classification

The sequential inference under `no_grad` yields 2,385/3,000 correct inferences (79.50%) in 22.40 s, therefore being the fastest YOLOv5s run, outpacing MSI (25.30 s) while trailing it in accuracy (81.40%). The result is consistent with the desktop-class Intel Core i9-10900X's higher core count and clocks for single-image CPU inference.

5.5 Global results & ranking

This subchapter presents the global results of the whole CIFAR-10 inference experiment. Then, a ranking based on the presented factors and another one for extra fairness in the process will be explained, in order to select the best combination for the last experiment of the present thesis.

The table below collates, for each model-device pair, the total inference time, number of correct predictions, accuracy, and the derived FPS, providing a like-for-like view of throughput and correctness across architectures and runtimes.

MobileNetV2	Total inference time	Correctly classified images	Accuracy	FPS
MSI Modern 14	210.00 s.	2,415/3,000	80.50 %	~ 15
RPZ2W	554.50 s.	2,241/3,000	74.70 %	~ 6
Intel NCS2	160.30 s.	2,205/3,000	73.50 %	~ 19
Raspberry Pi 5	103.50 s.	2,196/3,000	73.20 %	~ 29
Jetson Nano	285.50 s.	2,184/3,000	72.80 %	~ 11
Custom build PC	220.30 s.	2,424/3,000	80.80 %	~ 14

ResNet-18	Total inference time	Correctly classified images	Accuracy	FPS
MSI Modern 14	28.10 s.	2,814/3,000	93.80 %	~ 107
RPZ2W	289.15 s.	2,814/3,000	93.80 %	~ 10
Intel NCS2	26.70 s.	2,796/3,000	93.20 %	~ 112
Raspberry Pi 5	110.50 s.	2,814/3,000	93.80 %	~ 27
Jetson Nano	72.50 s.	2,816/3,000	93.87 %	~ 42
Custom build PC	15.45 s.	2,424/3,000	94.10 %	~ 194

Yolov5s	Total inference time	Correctly classified images	Accuracy	FPS
MSI Modern 14	25.30 s.	2,442/3,000	81.40 %	~ 119
RPZ2W	637.30 s.	2,430/3,000	81.00 %	~ 5
Intel NCS2	59.00 s.	2,466/3,000	82.21 %	~ 51
Raspberry Pi 5	121.00 s.	2,475/3,000	82.50 %	~ 25
Jetson Nano	130.69 s.	2,441/3,000	81.37 %	~ 23
Custom build PC	22.40 s.	2,385/3,000	79.50 %	~ 134

Table 31 Global results for the CIFAR-10 inference experiment

Presented the global results, finally the chapter presents the ranking of the different individual tests.

The objective is to collapse heterogeneous evidence containing accuracy, latency, and hardware price, into a single, comparable score for each hardware–software combination. To do so, a simple additive weighting (SAW), also known as a weighted-sum model (WSM), is used over min–max–normalized metrics. SAW/WSM is widely applied in multi-criteria decision analysis because it is transparent, tunable, and amenable to sensitivity analysis. The "ranking tool", a simple Python script, implements this scheme and outputs a sorted list from highest to lowest score [54].

Two out of the three inputs come directly from the experimental protocol, and the last one, from the market context. The first one, accuracy, is the top-1 percentage over 3,000 sequential, single-image inferences per device and model. Then, latency is the total elapsed time for those 3,000 inferences. Price (PVP) is the last one, and it reflects typical Q3-2025 pricing in Spain for each device: current retail if still in production, or representative second-hand values when discontinued. Including price alongside technical metrics follows standard practice when decisions must balance technical performance with budget constraints.

MSI Modern 14	Raspberry Pi Zero 2 W	Raspberry Pi Zero 2 W + Intel NCS2	Raspberry Pi 5	NVIDIA Jetson Nano	Custom build PC
400€	20€	20€ + 40€ = 60€	80€	220€	~2.100€

Table 32 Q3 2025 prices for each hardware

Because these metrics are in different units and value directions, they are normalized before aggregation. Min–max normalization is applied to map each metric to [0,1]. For accuracy, "higher is better," so values are scaled directly. For time and price, "lower is better," so the normalized values are inverted (i.e., shorter time and lower cost receive larger normalized scores). This elementary normalization is recommended in MCDA to put heterogeneous criteria on a common, dimensionless footing and to keep the arithmetic and interpretation simple [55].

Weights express the study's priorities: here, 0.35 for latency, 0.25 for accuracy, and 0.25 for price. The higher weight on latency reflects the much larger cross-device spread observed in the measurements and the thesis goal of efficient edge inference, which aligns with established literature that treats accuracy and real-world latency as joint objectives for on-device ML. While other combination rules exist (e.g., multiplicative), a tunable weighted sum remains a defensible choice and supports post-hoc weight-sensitivity checks if needed [56].

An extra metric, called 'resize bonus' is then added for models that require up-scaling inputs at inference time: +0.15 for MobileNetV2 (32x32 to 224x224) and +0.10 for YOLOv5s (32x32 to 128x128). The intent is to partially compensate for devices that expend extra computing power and memory bandwidth on local resizing, which is not needed by ResNet-18 running natively at 32x32. This adjustment is justified by the well-documented quadratic growth of convolutional computing with input resolution and by classifier baselines trained at higher input sizes [57]. Alternative fairness corrections could subtract measured preprocessing time; here, an additive bonus is preferred for simplicity and transparency.

Finally, the tool aggregates the normalized terms using the specified weights, adds the resize bonus where applicable, and sorts combinations by the resulting score. Conceptually, this produces a single scalar that increases with accuracy, decreases with latency and price, and modestly rewards pipelines burdened by larger input resolutions. The method is intentionally lightweight, reproducible, and consistent with broader benchmarking guidance that treats preprocessing as part of the end-to-end path while allowing design-specific adjustments to reflect study aims.

Device	Model	Norm. Accuracy	Norm. Time	Norm. Price	Bonus	Score
RPZ2W + NCS2	ResNet18	0.957746	0.981909	0.980769	0	0.828297
MSI	ResNet18	0.985915	0.979657	0.817308	0	0.793686
Jetson Nano	ResNet18	0.989202	0.908258	0.903846	0	0.791152
RPi 5	ResNet18	0.985915	0.847150	0.971154	0	0.785770
RPZ2W + NCS2	YOLOv5s	0.441784	0.929967	0.980769	0.10	0.781127
MSI	YOLOv5s	0.403756	0.984160	0.817308	0.10	0.749722
RPi 5	YOLOv5s	0.455399	0.830265	0.971154	0.10	0.747231
Jetson Nano	YOLOv5s	0.402347	0.814682	0.903846	0.10	0.711687
RPi 5	MobileNetV2	0.018779	0.858406	0.971154	0.15	0.697926
RPi Zero 2 W	ResNet18	0.985915	0.559862	1.000000	0	0.692430
MSI	MobileNetV2	0.361502	0.687143	0.817308	0.15	0.685203
RPZ2W + NCS2	MobileNetV2	0.032864	0.767066	0.980769	0.15	0.671881
Custom PC	ResNet18	1.000000	1.000000	0.000000	0	0.600000
Jetson Nano	MobileNetV2	0.000000	0.565731	0.903846	0.15	0.573967
Custom PC	YOLOv5s	0.314554	0.988824	0.000000	0.10	0.524727
Custom PC	MobileNetV2	0.375587	0.670580	0.000000	0.15	0.478600

RPi Zero 2 W	MobileNetV2	0.089202	0.133151	1.000000	0.15	0.468903
RPi Zero 2 W	YOLOv5s	0.384977	0.000000	1.000000	0.10	0.446244

Table 33 Ranking for the experiment results

Reading the ranking in Table 5.19, a clear pattern emerges. Combinations built on ResNet-18 occupy the top of the table across heterogeneous hardware, reflecting the model's consistently high accuracy at the native 32×32 resolution and the absence of any resizing overhead. The first four entries are ResNet-18 on RPZ2W+NCS2, MSI, Jetson Nano, and Raspberry Pi 5; all pair strong correctness with competitive end-to-end time. Mid-table positions are dominated by YOLOv5s. The resize bonus helps these pipelines, yet it does not offset their lower accuracy on this task and, on several devices, their longer total inference time. MobileNetV2 trails for the same reasons, despite receiving the larger bonus. This outcome is coherent with the known growth of convolutional compute with input resolution: once accuracy and latency are jointly valued, models that avoid up-scaling tend to fare better under a weighted-sum synthesis.

Device characteristics further explain the ordering. The Raspberry Pi 5's Cortex-A76 CPU reduces latency substantially relative to the Zero 2 W's Cortex-A53, but dedicated accelerators are still decisive: offloading to the Intel Neural Compute Stick 2's Myriad X VPU yields near-laptop throughput from a very low-cost host. Conversely, the Jetson Nano's 128-core Maxwell GPU lifts performance via TensorRT/ONNX Runtime but its higher device price tempers the composite score once cost is included. Finally, the custom desktop remains the raw-latency leader, yet its acquisition price drives the normalized price term toward zero, pulling down the overall score under a transparent weighted-sum (SAW/WSM) aggregation.

Focusing on the winner, RPZ2W + Intel NCS2 + ResNet-18 achieves the best composite score by aligning strength on all three criteria. Accuracy remains high for CIFAR-10 (93.2%). End-to-end latency compresses to 26.7 s for 3,000 images through VPU offload on the Myriad X. Price is minimal in the Q3-2025 Spanish context because the Zero 2 W is an inexpensive host and the NCS2 is a low-cost USB accelerator. Under a weighted-sum scheme, this combination dominates cost and time while retaining excellent accuracy, making it a pragmatic choice for edge-constrained biomedical prototypes that demand portable, low-power, and reproducible inference, such as the one presented in the next and final experiment of the thesis.

CHAPTER 6
A biomedical application

This chapter operationalizes the benchmark findings into a practical, end-to-end biomedical application. Building on the selection made in Chapter 5, the student and the external co-tutor converged, after considering adjacent use cases, on a tool that, in real time, proposes the most probable condition among six common skin diseases. The choice reflects both feasibility on the selected edge platform and clinical relevance. Skin conditions constitute a substantial global burden and are increasingly addressed through remote triage and decision support, where fast, portable image classification can augment point-of-care workflows. In this light, translating the benchmarked model-device pair into a lean, field-oriented pipeline directly serves health institutions priorities.

To support deployment in live settings, the chapter introduces a compact, interchangeable hardware component. This component helps to capture inputs under realistic conditions, while keeping the remainder of the system unchanged, thereby preserving portability across similar peripherals. The dataset choice is presented and justified, as well as the fine-tuning process for ResNet-18.

In continuity with earlier chapters, the practical implementation is accompanied by a brief analysis and explanation of the results obtained at execution time.

6.1 Hardware verification

The interchangeable imaging device selected for the live biomedical test and introduced before is the NGS SwiftCam 300. It is first attached to the student's PC via USB and later connected to the RPZ2W through a USB hub. This sequence preserves the same capture pipeline across hosts and allows verifying behavior before moving to the edge target.

The SwiftCam 300 is a USB-2.0 webcam with a 300 Kpx sensor, built-in microphone, and vendor software support for up-scaled capture. In practice, the device exposes standard video modes, including VGA (640x480) and QVGA (320x240), which are the operating points exercised in this subchapter. These modes are consistent with typical 300 Kpx webcams. Considering this device releases in the mid 2000's, the camera is modest but adequate: it is inexpensive, drivers are plug-and-play, and it delivers stable frames at VGA/QVGA, sufficient for prototyping the proposed edge pipeline.

Figure 24 NCS SwiftCam 300

To validate capture and inference end-to-end on the host MSI PC, a short Python script is developed. The script is based on the one used for the inference test on MSI using fine-tuned ResNet-18. At first, it opens the webcam with OpenCV, normalizes each frame with the same transforms used during training, and feeds the fine-tuned ResNet-18 for CIFAR-10 in evaluation mode.

The loop runs at the camera's default 640x480 setting and prints the top-1 class with confidence on each frame. Above 70% of confidence, the detected object is considered as detected. Otherwise, the verdict is not detected. At first the webcam is tested pointing to on-screen sample images and then to a real object (e.g., Figure 6.2), confirming correct operation under live input.

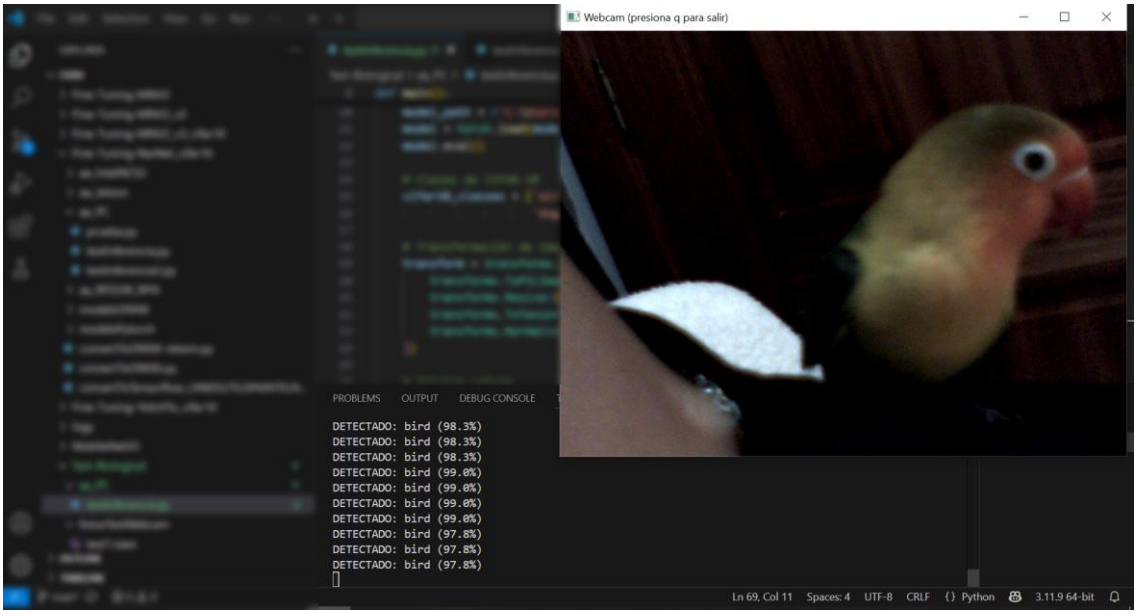

Figure 25 SwiftCam 300 inference verification

The same program is then adapted to run at 320x240 by switching the camera to a lower mode. This can be easily done thanks to `cv2` library and the instructions:

```
cap.set(3, 320)

cap.set(4, 240)
```

Inference remains stable at this resolution, so the application is finalized with 320x240 in mind to reduce bandwidth and preprocessing load when deployed on the RPZ2W + Intel NCS2 platform.

6.2 Dataset acquisition & model fine-tuning

The dataset for this application is sourced from Kaggle, a large, community-curated platform that hosts public and private datasets, along with tooling for storage, versioning, and distribution to support machine-learning research and reproducibility. Its "Datasets" section enables topical browsing and filtering (e.g., "Classification"), which facilitates discovering task-appropriate, labeled image corpora for supervised learning.

Among competing options, the selected corpus is "Skin Lesions Classification Dataset" by *ahmedxc4*, which merges HAM10000 (2019) with MSLD v2.0, providing images labeled into 14 diagnostic categories (Actinic keratoses; Basal cell carcinoma; Benign keratosis-like lesions; Chickenpox; Cowpox; Dermatofibroma; Healthy; HFMD; Measles; Melanocytic nevi; Melanoma; Monkeypox; Squamous cell carcinoma; Vascular lesions). The merge yields a broad mix of dermatoscopic and clinical photographs gathered from established sources, offering label breadth and visual diversity that are advantageous for transfer learning. The constituent HAM10000 dataset is a widely used benchmark of multi-source dermatoscopic images of common pigmented lesions, documented in Scientific Data. MSLD v2.0 is a curated image set introduced to support multiclass lesion recognition [58] [59].

This dataset is preferred over alternatives for two practical reasons. First, it offers a substantial number of unique images per class rather than primarily relying on heavy data augmentation to inflate counts, which can overstate performance. Second, for this prototype, a six-class subset is retained to minimize label overlap and maximize separability: Actinic keratoses, Chickenpox, Cowpox, Dermatofibroma, Measles, and Melanoma. Several excluded categories are either too broad for triage (e.g., Healthy, Vascular lesions) or visually confusable in routine (e.g., Cowpox vs. Monkeypox), which can compromise end-user trust in a real-time assistant.

It is important to understand that image sizes vary per class, and even inside each class.

The chosen subset balances prevalence, clinical salience, and inter-class distinctiveness, setting a tractable target for the edge model selected in Chapter 5.

Figure 26 Sample of the skin lesions dataset

Then, the dataset is downloaded locally and uploaded manually to a new Roboflow project named "*skin_lesions*". Classes are then defined and set with the correct amount of images, coming from the Kaggle dataset commented before.

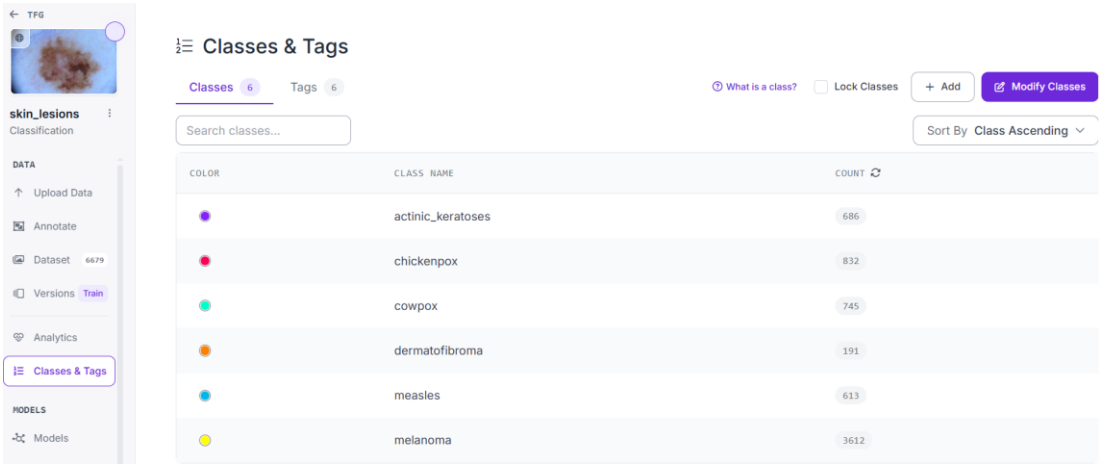

Figure 27 Classes for "skin_lesions" project in Roboflow

Inside the Roboflow workspace, and with the objective of maintaining image size coherence with the webcam resolution, all images are given a resize to 320x240. This way, the creation of perfect dataset version for the fine-tuning is achieved.

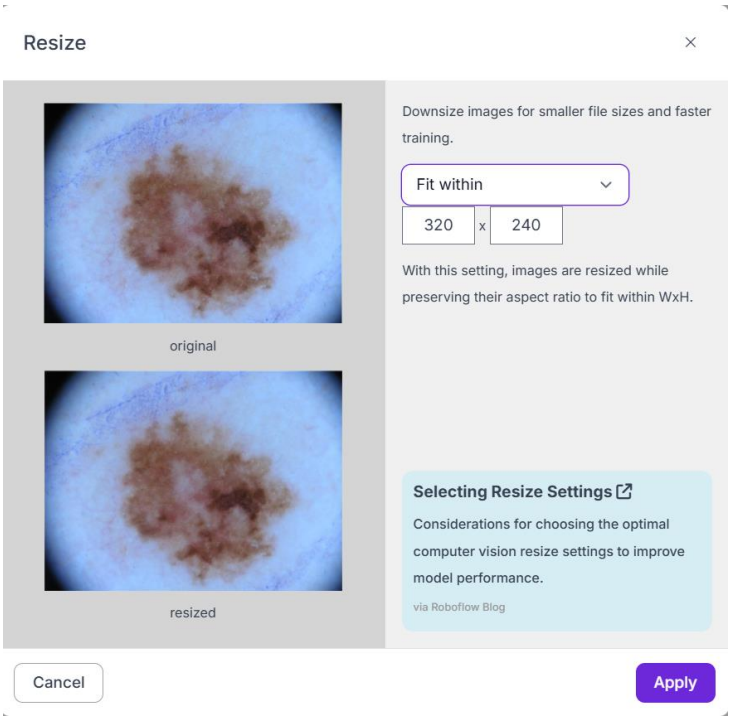

Figure 28 Resize confirmation for "skin_lesions" project

Fine-tuning follows the same ResNet-18 transfer-learning workflow introduced in subchapter 5.1.3, reused the notebook provided in [52] on Google Colab with minimal changes for the six-class skin-lesion task. After preparing the "*skin_lesions*" dataset in Roboflow and exporting at 320x240, an 80/20 train–validation split is applied. The learning-rate finder again exhibits a clear valley profile, and the selected start LR is close to the CIFAR-10 setting, reflecting similar loss–LR geometry under frozen-then-unfrozen training.

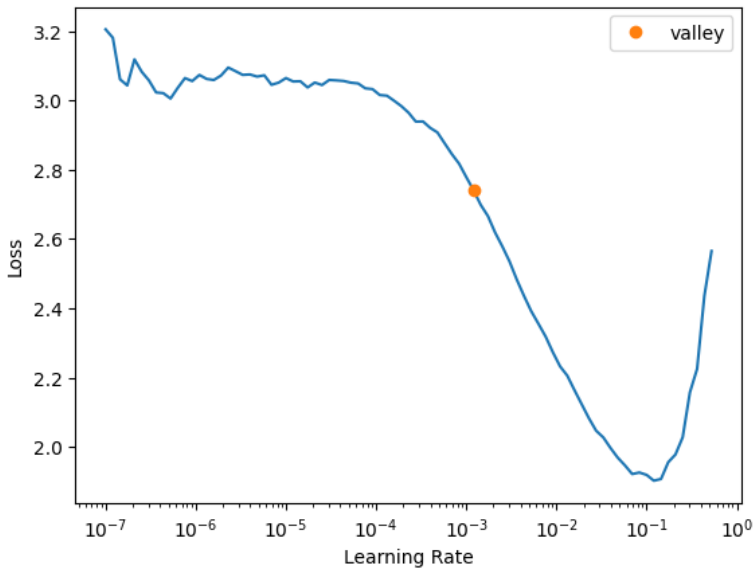

Figure 29 Learning rate curve and selected valley point for ResNet-18 training

The initial eight-epoch fit shows steady improvement: accuracy rises from 0.852 to 0.927, while validation loss declines from 0.457 to 0.215 on this training process. The curve suggests efficient head adaptation followed by meaningful gains post-unfreezing, without obvious overfitting at this stage.

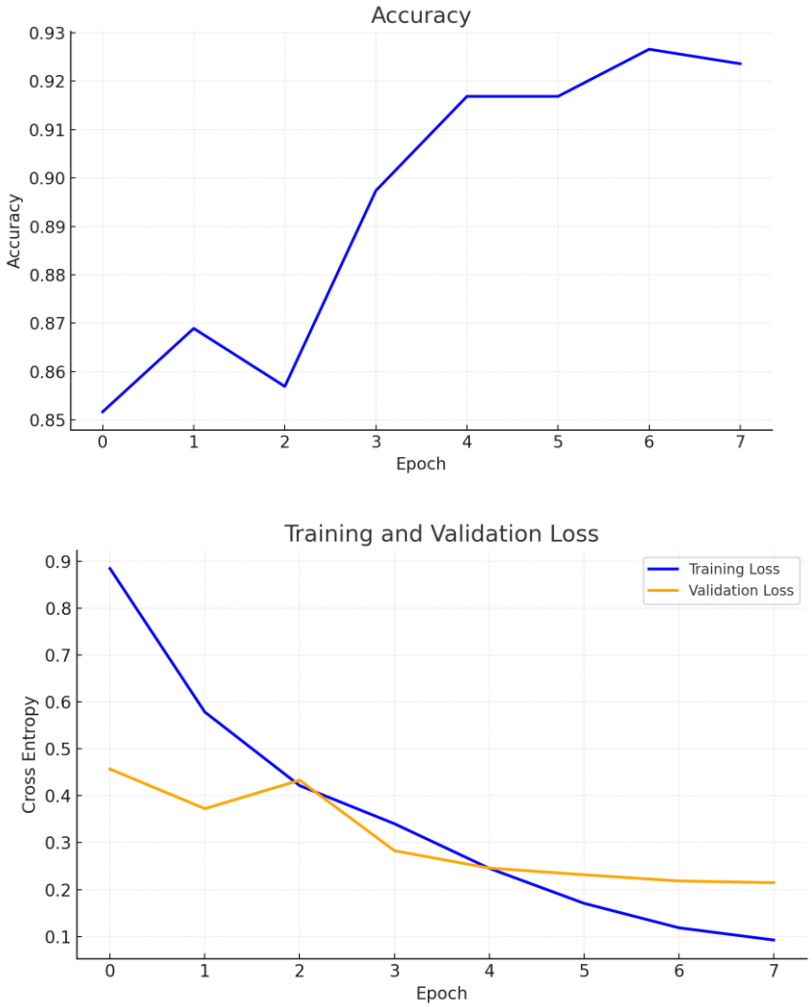

Figure 30 ResNet-18 accuracy & training & validation loss

A snapshot of most confused pairs highlights domain-specific overlaps: melanoma with actinic keratoses (35 and 29 cases, respectively), dermatofibroma with melanoma (13), and chickenpox with measles (6). The first two reflect well-documented diagnostic challenges in dermoscopy, where pigmented actinic keratoses and melanomas can share flat, pigmented facial patterns and non-specific structures that blur visual boundaries, as documented in [60].

Overall, the first stage meets expectations for a compact, edge-oriented classifier: fast convergence, high baseline accuracy, and interpretable errors that align with clinical nuance rather than pipeline faults. The stage-1 model is saved as:

```
learner.export('biological-stage-1').
```

The second stage fine-tuning (12 epochs, `fit_one_cycle` with LR slice 1e-5 to 1e-4) behaves as expected for a well-initialized residual backbone. One-cycle scheduling stabilizes around a shallow validation-loss basin (≈0.18–0.22) while accuracy trends upward from 0.924 to a peak of 0.946 (epoch 10, 18th in Figure 6.8), with small, non-monotonic fluctuations typical of cyclical LR training. Training loss continues to fall (0.087 down to 0.006), but without a concurrent rise in validation loss; this suggests useful representation sharpening rather than harmful overfitting at this horizon. The trajectory is consistent with the published advantages of the 1-cycle policy for rapid convergence and generalization when fine-tuning DNNs.

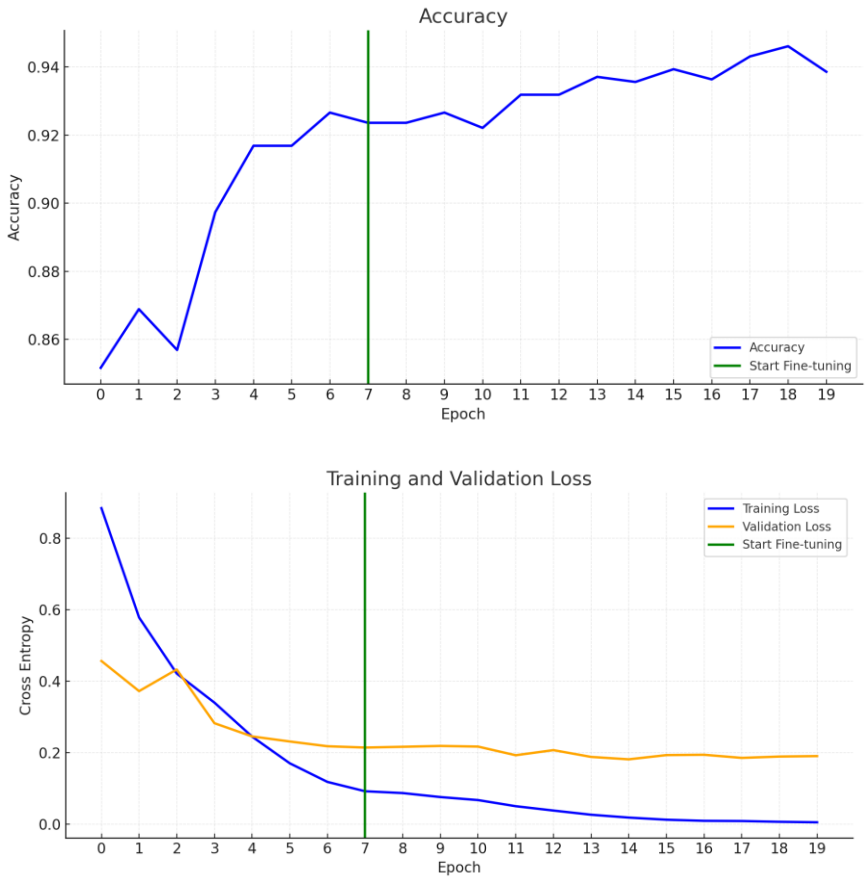

Figure 31 ResNet-18 accuracy & training & validation loss

For real-time use, the achieved accuracy appears sufficient to justify deployment on the selected edge stack, provided the end-to-end path (capture to preprocessing and inference) maintains interactive throughput. Final confirmation should come from direct measurement on the target pipeline.

6.3 Code & performance analysis

This subchapter applies the benchmark discipline to the live application in two steps. First, the pipeline is exercised on the MSI host to validate end-to-end correctness and establish a reference for accuracy and throughput under controlled conditions. Second, the same code path is deployed on the Raspberry Pi Zero 2 W with Intel Neural Compute Stick 2 to verify that accuracy and interactive performance hold on the selected edge target.

6.3.1 MSI Modern 14 with NCS SwiftCam 300

The Python script developed for this test follows with the one developed in subchapter 6.1. There is no model conversion: the script loads the exported checkpoint `biological-stage-2-pytorch` and runs it directly in PyTorch, locking evaluation under `torch.no_grad()` to avoid gradient bookkeeping during inference. Per frame, the script captures from the SwiftCam 300 via OpenCV's VideoCapture, explicitly requests 320x240 (width/height properties), converts BGR to RGB for the model, tensors the image, and obtains the top-1 class with a softmax-based confidence. The inference loop also measures per-image latency and reports it (ms) alongside the predicted label and confidence, providing immediate feedback on both correctness and interactivity.

During execution time, predictions remain stable when lighting is even, focus is adequate, and framing fills the field of view; confidence frequently exceeds 90% under these conditions. These observations are expected: reducing to 320x240 matches the dataset's prepared resolution and limits preprocessing, while the per-frame reporting keeps the assessment focused on user-visible confidence and instantaneous latency, which are the only runtime metrics of interest on this test.

This first point of contact serves as a baseline: it confirms the end-to-end path (capture, preprocessing, inference and display) and provides a reference for confidence before moving to the selected and intended hardware/software deployment in the next subsection.

6.3.2 Raspberry Pi Zero 2 W & Intel Neural Compute Stick 2 with NCS SwiftCam 300

For the edge deployment, the same trained checkpoint is exported once to ONNX at the target input shape (NCHW 1x3x240x320). The ONNX graph is then loaded by OpenVINO Runtime and compiled for the MYRIAD target, offloading inference to the Intel Neural Compute Stick 2's Movidius Myriad-X VPU.

The inference driver mirrors the MSI test but swaps the backend. It initializes OpenVINO's Core, reads the ONNX model, compiles it for "MYRIAD", and opens the SwiftCam via OpenCV. Capture is requested at 320×240 to match the dataset's prepared resolution and to keep transfer and preprocessing lightweight on the Pi; OpenCV properties confirm the effective

width and height at runtime. Each frame is converted BGR to RGB, converted to a CHW tensor, and submitted to the compiled model. Per-frame softmax, top-1 label, confidence, and latency in ms are printed to the console.

Empirically, the system behaves as expected. On the RPZ2W paired with the NCS2, predictions on the same ad-hoc images used on MSI remain stable and accurate, with confidence frequently above 90% under good lighting and framing; perceived responsiveness is comparable to the host-only run, reflecting effective VPU offload.

6.4 Results

It is indisputable that the objective of the chapter has been realized: namely, the conversion of the benchmarked edge stack into a functional biomedical application.

The small DL model was trained and fine-tuned on the curated skin lesions dataset, exported, and executed in two environments: (i) the MSI host, used as a baseline to verify end-to-end correctness and establish expected behavior; and (ii) the Raspberry Pi Zero 2 W paired with the Intel Neural Compute Stick 2, used as the target edge platform. In both settings, the live loop reports only the two metrics relevant for interactive use: the top-1 predicted class with its confidence and the per-frame latency in milliseconds. Observed behavior is consistent across hosts: on ad-hoc images not contained in the training set but depicting one of the six conditions, the system typically maintains a confident top-1 prediction (≥70% confidence) over sustained viewing (≈10 seconds) when framing and illumination are adequate. These findings are compatible with prior evidence that edge accelerators like the Myriad-X VPU can sustain practical, low-power inference for computer-vision tasks.

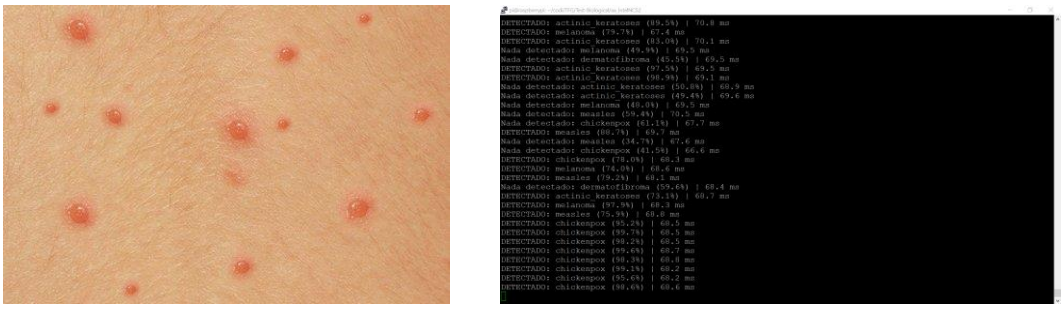

Figure 32 Terminal output for chickenpox

While encouraging, these results represent an application-level verification rather than a clinical validation. Performance on "in-the-wild" images supports the feasibility of real-time classification on constrained hardware, but broader generalization across other devices and clinical settings requires prospective testing, as emphasized in recent reviews of AI in dermatology, coming from [61]. The accompanying terminal capture (Figure 6.9) illustrates the live output format; predicted lesion, confidence, and latency per frame, which shows a practice for reporting actionable, user-visible signals in a point-of-care prototype.

CHAPTER 7
Conclusions

This thesis established to identify, under controlled and reproducible conditions, the edge platform and model that best balanced accuracy, latency, and cost. Additionally, an objective to demonstrate that the winning choice could underpin a practical biomedical application was also set. A standardized benchmarking methodology was designed and executed across heterogeneous devices. The resulting evidence was synthesized into a transparent ranking that foregrounds performance–price trade-offs. Building on that selection, the work then translated the top hardware–software pair into a live, six-class skin-lesion classifier, validating end-to-end feasibility on constrained hardware while preserving portability and methodological rigor.

To further demonstrate the complete exit of the thesis, the objectives set on subchapter 1.2 are discussed:

First, a comparison of devices under equal conditions was proposed, and for that, the thesis defined and enforced a standard protocol: 3,000 single-image, sequential inferences per run; identical or equivalent preprocessing per model; per-image timing; and fixed, documented conversion paths. Chapter 4 validated the pipeline and exposed critical pitfalls (e.g., color space and scaling mismatches), which were corrected further in the development. Chapter 5 then executed fine-tuned MobileNetV2, ResNet-18, and YOLOv5s across devices. Metrics were normalized and aggregated with a justified weighted-sum scheme and a modest resize bonus, yielding a defensible, like-for-like comparison.

Secondly, a real-world biomedical application was also set as an objective. Chapter 6 operationalized the winner by building a webcam-based classifier that proposes the most probable condition among six skin diseases. Dataset selection and curation were documented. Results commented in the subchapter 6.4 demonstrate the viability of the test for real-world scenarios, although improvements are to be made still, the experiment sets a baseline for the future work and possible deployment in healthcare scenarios.

Finally, it was set to show whether edge devices surpassed conventional systems in AI-based tasks. The benchmarking from chapter 5 and the whole chapter 6 showed where edge devices excel and where they do not, with accuracy–latency–price trade-offs made explicit. Dedicated accelerators can supply near-desktop responsiveness at a fraction of the cost and power, while general-purpose CPUs still provide robustness and simplicity. The study thus characterizes scenarios where edge inference is most compelling: low-power settings, bandwidth-limited environments, and privacy-sensitive applications that benefit from on-device processing.

The path to these conclusions was not linear. Early coordination at a distance, the absence of provided edge devices at thesis start phase, and the move between countries forced disciplined planning and curated advancements control.

Toolchain instability was a major source of friction: operating systems and Python versions occasionally clashed with framework and driver support, requiring several clean installations and tighter pinning of dependencies. Conversion idiosyncrasies (e.g., NHWC/NCHW adjustments, opset constraints, legacy MYRIAD support) demanded careful diagnostics. Some errors could have been prevented with deeper up-front compatibility checks; this insight shaped later choices, such as isolating per-device export paths and freezing preprocessing conventions.

The most valuable technical outcome is transferable skill: the ability to fine-tune and deploy modern DNNs across frameworks and devices, to measure them fairly, and to reason about their trade-offs under real constraints.

The most valuable personal outcome is confidence. Cultivated over a long arc spanning Spain and Latvia, the confidence in approaching ambiguous engineering tasks, structuring them into testable steps, and carrying them through to a functioning system is achieved greatly.

On a final note, the thesis demonstrates both analytical capability and practical synthesis. It integrates learning across machine learning, edge devices, and software engineering, and it produces a working artifact that is faithful to the evidence assembled in earlier chapters.

7.1 Legacy

The project leaves a twofold legacy: a reproducible edge-based benchmark for DL models and a working biomedical application.

The benchmark contributes a standardized harness, including uniform preprocessing, per-image timing, and documented conversion paths (TensorFlow & Pytorch to TFLite, ONNX Runtime), plus a transparent MCDA ranking tool to synthesize accuracy, latency, and cost. These assets enable like-for-like evaluation across heterogeneous devices and can be reused to assess new hardware or models.

Complementing the benchmark, the winning platform is transformed into a real-life automatic skin disease classification tool, demonstrating practical feasibility on constrained hardware and offering a portable blueprint for point-of-care image classification.

All code, scripts, training notebooks, and fine-tuned model artifacts are left published in a public GitHub repository, with clear documentation, versioned exports, and environment files. This open package allows full replication, facilitates extension to additional datasets or classes, and supports comparative research in edge inference and innovative skin disease detection.

7.2 Relationship of the work carried out with the studies completed

The Informatics Engineering degree at the Universitat Politècnica de València provides a comprehensive, theory-and-practice-oriented curriculum that equips students to execute projects of the scope and rigor of this thesis. The work presented here draws directly on core competencies developed across mandatory and specialization courses, integrating them into a coherent engineering outcome.

The programming and software backbone of the project, including clean drivers, reproducible pipelines, and disciplined versioning, rests on Introduction to Computer Science and Programming, Programming, Data Structures and Algorithms, and Software Engineering. These courses established sound coding practices, algorithmic reasoning, testing habits, and the ability to structure maintainable, extensible codebases. This capabilities were essential for the multi-device inference testing and the biomedical application.

Model adaptation and familiarity with modern deep learning pipelines trace to Intelligent Systems, which first introduced machine-learning concepts and an evaluation methodology.

The image processing and performance-aware aspects of the work benefited from Parallel Computing, for understanding compute/throughput trade-offs and hardware utilization. Then, Interactive and Immersive Multimedia Systems was useful for practical handling of imaging, capture, and real-time constraints.

Edge deployment required fluency beyond model training. Internet of Things, Computer Networks, Operating Systems Fundamentals, and Design of Operating Systems underpinned remote access to the edge devices, shell scripting, process supervision, and the reliable configuration of heterogeneous devices. These courses directly supported SSH-based workflows, headless operation, and the careful alignment of libraries, drivers, and runtimes across platforms.

Finally, the comparative analysis and benchmarking mindset for defining fair protocols, measuring latency and accuracy consistently were strongly reinforced by Advanced Architectures, whose laboratory sessions applied some measurement disciplines that could be adapted and used in the present thesis.

The thesis is a concrete synthesis of many degree's courses learning outcomes. Among them, Internet of Things stands out as the closest analogue to the end-to-end engineering performed. Appropriately, it is led by the thesis tutor, Pietro, and provided the basics for taking models out of the training environments and into dependable, real-world systems.

7.3 Future work

To strengthen scientific rigor, future iterations should embed systematic error analysis around a per-class confusion matrix. Beyond reporting top-1 accuracy, inspecting false positives/negatives can reveal stable confusion modes (e.g., melanoma vs actinic keratoses) and guide targeted remedies: class-specific augmentation, hardness-aware sampling, additional data collection for underrepresented patterns, threshold tuning, or post-hoc calibration.

The benchmark should also expand to more devices and architectures to increase external validity. Adding diverse accelerators (e.g., newer VPUs, Edge USB sticks, Jetson Orin-class GPUs, and modern CPUs with different architectures and instruction sets) would stress the portability of conversion paths and surface price-performance frontiers under broader constraints.

For the biomedical application, the next phase is an engineering push that transforms the already promising prototype into a dependable, clinic-ready tool. Concretely, the codebase should be hardened into a modular capture-preprocess-infer-render pipeline with version-pinned dependencies and automated checks for image I/O. In parallel, probability calibration should align scores with empirical correctness so that a given confidence threshold has predictable meaning at the point of use. Together, these steps tighten technical validity and make confidence displays actionable for clinicians.

In tandem, the user interface should evolve from a developer console to a task-fit surface: concise top-1 class, calibrated confidence, unobtrusive quality warnings (e.g., blur/poor exposure), and optional top-2 to support differential reasoning. All of this should be presented with language that communicates uncertainty rather than overclaiming. Human-factors principles emphasize matching real use environments and providing clear feedback on system state; adhering to them reduces use errors and builds trust.

Bibliography

[1] Grand View Research, "Edge AI Market Size, Share & Trends Analysis Report By Component (Hardware, Software, Services), By End-use Industry (Consumer Electronics, Smart Cities, Automotive), By Region, And Segment Forecasts, 2025 - 2030".

[2] T. Pangarkar, "Edge Computing Demand for The Proliferation of IoT Devices," 2023.

[3] Xubin Wang, and Weijia Jia, "Optimizing Edge AI: A Comprehensive Survey on Data, Model, and System Strategies," 2025.

[4] Amit Kharat, Vinay Duddalwar, Krishna Saoji, Ashrika Gaikwad, Viraj Kulkarni, Gunjan Naik, Rohit Lokwani, Swaraj Kasliwal, Sudeep Kondal, Tanveer Gupte, and Aniruddha Pant, "Role of Edge Device and Cloud Machine Learning in Point-of-Care Solutions Using Imaging Diagnostics for Population Screening," 2020.

[5] A. Elhanashi, "Advancements in TinyML: Applications, Limitations, and Impact on IoT Devices," 2024.

[6] S. Heydari, "Tiny Machine Learning and On-Device Inference: A Survey of Applications, Challenges, and Future Directions," 2025.

[7] Y. Abadade, "Empowering Healthcare: TinyML for Precise Lung Disease Classification," 2024.

[8] X. Kunran, "An Ultra-Low Power TinyML System for Real-Time Visual Processing at Edge," 2022.

[9] R. David, "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems," 2021.

[10] R. Fernandez, "A brief history of edge computing," 2022.

[11] I. Aktaş, "Cloud and edge computing for IoT: a short history," 2018.

[12] ThinkRobotics, "Jetson Nano vs Raspberry Pi 5 for AI: The Ultimate Performance and Value Comparison," 2025.

[13] Wikipedia, "Nvidia Jetson," 2019.

[14] B. Varghese, "Revisiting the Arguments for Edge Computing Research," 2021.

[15] Seeed Studio, "The NVIDIA Jetson Family: For AI at the Edge and Autonomous Machines," 2025.

[16] Asma Baobaid, and Mahmoud Meribout, "Edge-GPU Based Face Tracking for Face Detection and Recognition Acceleration," 2025.

[17] Karumbunathan, and Leela S., "NVIDIA Jetson AGX Orin Series: A Giant Leap Forward for Robotics and Edge AI Applications," 2022.

[18] Renesas, "RZ/V2H Datasheet," 2025.

[19] Andrew G. Howard, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," 2017.

[20] Mark Sandler, and Andrew Howard, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018.

[21] D. Qin, "MobileNetV4: Universal Models for the Mobile Ecosystem," 2024.

[22] X. Zhang, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," 2018.

[23] J. Lin, "MCUNet: Tiny Deep Learning on IoT Devices," 2020.

[24] Sachin Mehta, and Mohammad Rastegari, "MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer," 2021.

[25] Tess Watt, Christos Chrysoulas, and Peter J Barclay, "Moving Healthcare AI-Support Systems for Visually Detectable Diseases onto Constrained Devices," 2024.

[26] Shinde, and Rupali Kiran, "Squeeze-MNet: Precise Skin Cancer Detection Model for Low Computing IoT Devices Using Transfer Learning," 2022.

[27] Romina Aalishah, Mozhgan Navardi, and Tinoosh Mohsenin, "MedMambaLite: Hardware-Aware Mamba for Medical Image Classification," 2025.

[28] S. Somvanshi, "From Tiny Machine Learning to Tiny Deep Learning: A Survey," 2025.

[29] H. Han, "TinyML: A Systematic Review and Synthesis of Existing Research," 2022.

[30] Soroush Heydari, and Qusay H. Mahmoud, "Tiny Machine Learning and On-Device Inference: A Survey of Applications, Challenges, and Future Directions," 2025.

[31] Y. Abadade, "A Comprehensive Survey on TinyML," 2023.

[32] TDK SensEI, "11 Myths About Machine Learning at the Edge".

[33] Jude Haris, Perry Gibson, José Cano, Nicolas Bohm Agostini, and David Kaeli, "SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference," 2021.

[34] PyTorch, "PyTorch Edge: Build innovative and privacy-aware AI experiences for edge devices".

[35] Microsoft, "Faster inference for PyTorch models with OpenVINO Integration with Torch-ORT," 2022.

[36] Tianqi Chen, and Thierry Moreau, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," 2018.

[37] A. G. Howard, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," 2017.

[38] Sandler, Mark, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018.

[39] K. He, "Deep Residual Learning for Image Recognition," 2016.

[40] PyTorch, "resnet18 Documentation".

[41] F. Ramzan, "A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State fMRI and Residual Neural Networks," 2019.

[42] Roboflow, "YOLOv5 Classification," 2022.

[43] E. Haque, "Rice leaf disease classification and detection using YOLOv5," 2022.

[44] Canicius Mwitta, Glen C. Rains, and Eric Prostko, "Evaluation of Inference Performance of Deep Learning Models for Real-Time Weed Detection in an Embedded Computer," 2024.

[45] Areeba Abid, Priyanshu Sinha, and Aishwarya Harpale, "Optimizing Medical Image Classification Models," 2021.

[46] Stanford Vision Lab, Stanford University, Princeton University, "ImageNet Download Site," 2012-2025.

[47] Aidar Amangeldi, Angsar Taigonyrov, Muhammad Huzaid Jawad, and Chinedu Emmanuel Mbonu, "CNN and ViT Efficiency Study on Tiny ImageNet and DermaMNIST Datasets," 2025.

[48] TensorFlow, "Transfer learning and fine-tuning," 2024.

[49] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling and C.-J. Wu, "MLPerf Inference Benchmark," 2024.

[50] S. Kornblith, "Do Better ImageNet Models Transfer Better?," 2019.

[51] Roboflow Blog, "How to Train MobileNetV2 On a Custom Dataset," 2021.

[52] Roboflow, "ResNet-32," 2017.

[53] Roboflow, "YOLOv5 Classification," 2022.

[54] Evangelos Triantaphyllou, and Alfonso Sánchez, "A SENSITIVITY ANALYSIS APPROACH FOR SOME DETERMINISTIC MULTI-CRITERIA DECISION MAKING METHODS," 1997.

[55] Irik Z. Mukhametzyanov, "Normalization of Multidimensional Data for Multi-Criteria Decision Making Problems," 2023.

[56] Richard O'Shea, Peter Deeney, Evangelos Triantaphyllou, Luis Diaz-Balteiro, and S. Armagan Tarim, "Weight stability intervals for multi-criteria decision analysis using the weighted sum model," 2025.

[57] Jérémy Morlier, Mathieu Léonardon, and Vincent Gripon, "Input Resolution Downsizing as a Compression," 2025.

[58] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler, "The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions," 2018.

[59] Shams Nafisa Ali, Md. Tazuddin Ahmed, Tasnim Jahan, Joydip Paul, S. M. Sakeef Sani, Nawsabah Noor, Anzirun Nahar Asma, and Taufiq Hasan, "A Web-based Mpox Skin Lesion Detection System Using State-of-the-art Deep Learning Models Considering Racial Diversity," 2023.

[60] Philipp Weber, Philipp Tschandl, Christoph Sinz, and Harald Kittler, "Dermatoscopy of Neoplastic Skin Lesions: Recent Advances, Updates, and Revisions," 2018.

[61] Maria Paz Salinas, Javiera Sepúlveda, Leonel Hidalgo, Dominga Peirano, Macarena Morel, Pablo Uribe, Veronica Rotemberg, Juan Briones, Domingo Mery, and Cristian Navarrete-Dechent, "A systematic review and meta-analysis of artificial intelligence versus clinicians for skin cancer diagnosis," 2024.

[62] Stanley Glenn Enriquez Brucal, "Development of Tomato Leaf Disease Detection using Single Shot Detector (SSD) Mobilenet V2," 2023.

APPENDIX A
Sustainable Development Goals

This appendix explains the degree of relationship of the thesis with the Sustainable Development Goals (SDGs).

Sustainable Development Goal	High	Medium	Low	Not related
1. No poverty			x	
1. Zero hunger				x
2. Good Health and Well-Being	x			
3. Quality Education				x
4. Gender Equality				x
5. Clean Water and Sanitation				x
6. Affordable and Clean Energy				x
7. Decent Work and Economic Growth				x
8. Industry, Innovation and Infrastructure	x			
9. Reduced Inequalities		x		
10. Sustainable Cities and Communities			x	
11. Responsible Consumption and Production			x	
12. Climate Action				x
13. Life Below Water				x
14. Life on Land				x
15. Peace, Justice and Strong Institutions				x
16. Partnerships				x

Table 34 Relationship between the thesis and the SDGs

The thesis is deeply aligned with the following Sustainable Development Goals (SDGs): Industry, Innovation and Infrastructure (SDG 8): By leveraging artificial intelligence to enhance diagnostic tools, the thesis fosters cutting-edge innovation and contributes to resilient, technology driven infrastructure that improves quality of life.

Reduced Inequalities (SDG 9): The developed solution is intentionally affordable, ensuring accessibility in low-resource and developing settings, thereby promoting greater equity in healthcare access across socio-economic divides.

Good Health and Well-Being (SDG 2): The core aim of the thesis is to deliver a balanced, cost-effective AI-powered tool to detect skin problems early, improving preventive care and contributing to universal health and well-being.

In essence, the thesis achieves a triad of sustainable objectives: enabling innovation in infrastructure through AI; tackling inequality by ensuring affordability; and advancing health outcomes by providing accessible, high-value diagnostic capabilities. This synergy underscores its relevance and potential impact across societal, economic and health domains.