



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Benchmarking Mobile Frameworks for Computer Vision Applications

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING
INGEGNERIA INFORMATICA

Author: **Carlo Barbagiovanni Minciullo**

Student ID: 927629

Advisor: Prof. Piero Fraternali

Co-advisors: Federico Milani

Academic Year: 2021-22

Abstract

Nowadays, fields such as *Deep Learning*, *Computer Vision* and *Augmented Reality* are growing significantly, especially on mobile devices. It is therefore necessary for software to run properly on devices with limited resources without ruining the user experience. *PeakLens* is an Augmented Reality application that can recognize mountain peaks by framing a panorama with a mobile phone camera. To make the best use of Deep Learning and Computer Vision technologies on devices with limited hardware, it is necessary to optimize the Convolutional Networks. The algorithm has to be in fact very efficient and accurate without slowing down the device too much. This work performs various comparisons on several devices comparing their performance in terms of initialization time, image pre-processing, latency, throughput, memory and battery consumption. The frameworks tested are PolimiDL, TensorFlow Lite and PyTorch Mobile. PolimiDL is the Deep Learning framework on which PeakLens is based, TensorFlow Lite is the solution proposed by Google and PyTorch Mobile is the framework developed mainly by Meta AI(Facebook). The performance of the three frameworks will be compared to understand the behaviors and strengths of each.

Keywords: Optimization, Mobile Application, Convolutional Neural Networks, Inference, PyTorch Mobile, TensorFlow Lite, PolimiDL, Deep Learning, Android, Benchmarking, Devices.

Abstract in lingua italiana

Al giorno d'oggi campi come *Deep Learning*, *Computer Vision* e *Realtà aumentata* stanno notevolmente crescendo, soprattutto sui dispositivi mobili. È dunque necessario che il software venga eseguito correttamente su dispositivi con limitate risorse senza rovinare l'esperienza dell'utente. *PeakLens* è un'applicazione di Realtà Aumentata che inquadrando un panorama con la fotocamera del cellulare è in grado di riconoscere i picchi delle montagne. Per utilizzare al meglio le tecnologie di Deep Learning e Computer Vision su dispositivi dall'hardware limitato è necessario ottimizzare le Reti Convolutionali. L'algoritmo deve infatti essere molto efficiente ed accurato senza rallentare troppo il dispositivo. Questo lavoro effettua vari confronti su più dispositivi confrontandone le prestazioni in termini di tempo di inizializzazione, pre-elaborazione dell'immagine, latenza, throughput, consumo di memoria e batteria. I framework testati sono PolimiDL, TensorFlow Lite e PyTorch Mobile. PolimiDL è il framework di Deep Learning su cui PeakLens è basata, TensorFlow Lite è la soluzione proposta da Google e PyTorch Mobile è il framework sviluppato principalmente da Meta AI(Facebook). Le prestazioni dei tre framework verranno confrontate per capire i comportamenti e i punti di forza di ognuno.

Parole chiave: Ottimizzazione, Applicazione mobile, Reti neurali convoluzionali, Inferenza, PyTorch Mobile, TensorFlow Lite, PolimiDL, Apprendimento profondo, Android, Misurazione prestazioni, Dispositivi.

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
1 Introduction	1
2 Related Work	3
2.1 Mobile Inference	3
2.1.1 Hardware	4
2.1.2 Existing Framework	5
2.1.3 Optimization Techniques	8
2.1.4 Evaluation	9
3 Methods	13
3.1 Mobile inference frameworks	13
3.1.1 PolimiDL	14
3.1.2 TensorFlow Lite	15
3.1.3 PyTorch Mobile	15
3.2 Deep Learning model	16
3.2.1 Peaklens Original	16
3.2.2 Peaklens Optimized	17
3.2.3 Mobilenet v1	18
3.3 Model conversion	19
4 Evaluation	21
4.1 Environment	21
4.1.1 Devices	21
4.1.2 Application	23

4.1.3	Execution	24
4.2	Frameworks comparison	25
4.2.1	Metrics	25
4.2.2	Image pre-processing	26
4.2.3	Initialization	28
4.2.4	Inference	29
4.2.5	Consumption	30
5	Conclusions and Future work	43
	Bibliography	45
	List of Figures	49
	List of Tables	51
	Acknowledgements	55

1 | Introduction

The recent success of Deep Learning (DL) has boosted its operation in numerous areas, with remarkable results that are impacting people's lives. Typical implementations of DL models concentrate on maximizing accuracy for a given task, and infrastructures to achieve such an ideal have become significantly deeper and more complex over time. DL is widely used on smartphones, generating a need for optimizations at the architecture and framework levels to obtain good performance for real-time use. At first, powerful workstations with GPUs were fundamental for DL success, making their computationally expensive training possible. For this reason, to solve this problem, we have decided to use resources of embedded systems, such as smartphones, tablets, wearable devices, drones, and FPGAs, as these are improving rapidly; however, they are still not entirely suitable for the deployment of complex models. The use of a remote cloud to run the models is countered by privacy, cost, availability, and latency problems. These limitations drive us to use compact architectures for accelerating execution by optimizing computation, storage, memory occupation, and energy consumption. The development of DL on edge is helpful in various areas such as robotics, autonomous vehicles, Augmented Reality, health monitoring, and digital assistance, which rely on mobile devices with real-time constraints.



Figure 1.1: A picture taken using PeakLens.

An example of an app is PeakLens, an Augmented Reality mobile application that uses Artificial Intelligence to recognize mountain peaks. It does so we have improved the execution times of existing ready-to-use models are accelerated directly on the device without needing external hardware or specific components. Thanks to PolimiDL, a framework for accelerated DL inference on mobile and embedded systems, the execution times and efficiency of existing ready-to-use models have improved without compromising accuracy. This thesis analyzes three frameworks for mobile inference studying their opportunities on devices with different power and hardware.

The advantages of the thesis can be summarized as follows:

- We updated a benchmarking application by adding the PyTorch Mobile framework;
- We evaluated the performance with new metrics;
- We estimated and compared the performance of different deep learning frameworks on the execution of several models.

The thesis is organized as follows: Chapter 2 discusses the related work; Chapter 3 introduces mobile deep learning frameworks and models; Chapter 4 describes the evaluation of the frameworks and finally, Chapter 5 concludes and gives an outlook on the future work.

2 | Related Work

2.1. Mobile Inference

Deep Learning inference is the process of using a trained Neural Network model to make predictions over data inputs that have not been used in the model training phase. Neural networks are the key to recognizing and classifying data, for example labeling emails as spam or how applications create a music playlist based on our tastes. These operations require a significant amount of computing power and memory and are increasingly present in mobile devices such as smartphones, tablets, IoT devices and embedded systems. Previously, the architectures were client-server architectures and were built to allow a connection between the end device and the service provider. The server was always the entity that gave the incoming data, processed it and returned new data, such as predictions. The pros are the possibility of being able to separate the development of the mobile application from the management of Deep Learning and the distribution to different and more performing devices such as servers. This type of solution has its constraints and disadvantages: latency, in addition to all the service times, the communication time between client and server must also be considered; the continuous availability of an Internet connection; and privacy, as the data travels from the device through the web.

For this reason, the idea of developing and centralizing the exchange of data and processing it locally on mobile devices has gained ground in recent years. This solves some of the problems as data processing no longer takes place on remote servers, however, one must consider the restriction due to the limited capacity of mobile devices. In particular, machine learning models must be pre-trained, to avoid unnecessary calculations by the devices, with limited size, as they must be loaded into the RAM and processed by the processors; furthermore, the models must analyze a large amount of data in the shortest possible time, without overheating the devices. Therefore, the speed of calculation and the size of the model are the main issues to be managed.

To overcome these factors, mobile inference frameworks have implemented some solutions and exploited hardware acceleration or optimization techniques. Some of these frame-

works have been developed by large communities, such as Google's TensorFlow Lite from 2017 which tries to address the complexity of Deep Learning with ad-hoc solutions for model implementation in mobile, IoT and embedded devices, providing different levels of support for mobile inference and its possible optimizations.

2.1.1. Hardware

In the context of mobile devices, even though in recent years they either mount increasingly powerful CPUs or rely on additional hardware to support the workload for computationally intensive operations and to achieve better data parallelism, do not have the computing power of a computer. This is why hardware is used as efficiently as possible, or when possible, hardware acceleration is used to improve processing speed.

Starting from Digital Signal Processors (DSPs), the feature extraction started applying deep learning algorithms to signals present in sensory inputs, for example from images, objects motion and audio contexts like speeches and music. Then the graphics processing units (GPUs) is a dedicated processor that generally performs floating point operations (FLOPs) in computer graphics scenarios; since FLOPs are also involved in Deep Learning applications, the use of GPUs was also extended in this field, taking advantage of their increased number of cores, to allow for better parallel processing and to better perform matrix calculations, which are mainly present in CNNs. Neural Processing Units (NPUs) are a class of microprocessors that have been designed to accelerate neural networks and other artificial intelligence applications. Unlike GPUs, NPUs have a dedicated design for machine learning processing, which enables them to recognize computational patterns in deep neural networks more easily.

In addition, newer mobile devices make use of modern mounted chips (SoCs) that enable specialized hardware for vector and matrix operations, but access to them depends on software development kits (SDKs). It depends on the software development kits (SDKs) of each vendor, which are generally not scalable and reusable. For this reason, Google introduced Android Neural Networks API [10] (NNAPI), a set of libraries, designed to provide a basic level of functionality for machine learning frameworks, which seeks to standardize access to the SoC and provide runtime acceleration of Deep Learning for devices running Android version 8.1 (or higher). The same idea was developed by Apple, with its Metal API [6], available as of iOS 8: the integration of this library maximizes the parallel processing power of GPUs to realize machine learning challenges, and they have the advantage of targeting a limited and relatively homogeneous set of devices, for which they have full control over the production, which simplifies integration and support.

The deep learning solutions not only limit their applications to heavy computing machines but there is also growing interest in deploying DNNs in edge devices with limited hardware resources and energy. The hardware solutions for DNNs development and deployment range from general purpose architectures (CPUs and GPUs) to spatial architectures (FPGA and ASCI) [26]. The multiply-and-accumulate (MAC) operation is the fundamental component in both the fully connected and convolution layers, which can be parallelized easily to achieve a high inference speed. The hardware accelerator can either be a conventional hardware optimization with enhanced compute parallelism or modern accelerators that combine both hardware and software design capabilities. Recent advancements in developing efficient DNNs using software solutions provide promising performance with reduced memory and computing operations. DNN acceleration that uses hardware-software co-design is the current trend in developing efficient DNN applications.

One of the recent works on hardware acceleration [30] proposed a deep learning algorithm for image classification, implementing it on an FPGA hardware architecture using Simulink and an HDL encoder, emphasizing that optimizing power consumption is the main concern for mobile systems.

2.1.2. Existing Framework

Frameworks for the execution of DL models on mobile and embedded systems pursue optimized deployment on devices with limited resources, by managing memory allocation efficiently and exploiting the available hardware resources at best. The growing global interest in the challenge of Deep Learning and the possibility of using neural networks on mobile devices have led both IT industries and academic groups to develop software frameworks for running Deep Learning models. Starting with a pre-trained model, with weights from the training phase, and using their inference engine to execute the model, these frameworks make predictions based on the input data provided, for example recognizing whether there are dogs or cats in a photo and even people's faces.

Table 2.1 shows the most popular inference frameworks released in recent years.

Release year	Name	Android	iOS
2021	ODIN [11]	Yes	No
2020	DEEPLEARNING 4J [3]	Yes	Yes
2020	PHONE BIT	Yes	No
2019	PYTORCH MOBILE [16]	Yes	Yes
2019	POLIMIDL [14]	Yes	No
2018	MACE [5]	Yes	Yes
2018	HiAI [4]	Yes	Yes
2018	CAFFE2 [1]	Yes	Yes
2017	MXNET [8]	Yes	No
2017	NNAPI [10]	Yes	No
2017	TFLITE [21]	Yes	Yes
2017	SNPE [17]	Yes	No
2017	NCNN [9]	Yes	Yes
2017	ONNX [12]	Yes	No
2017	COREML [2]	No	Yes

Table 2.1: Some of the existing CNN frameworks.

The frameworks that we have used in our application are now better introduced.

PolimiDL

PolimiDL [14] is a framework for the execution of small Deep Neural Networks. It is designed with Fully Convolution Neural Networks in mind, but it can also execute networks with Fully Connected layers. PolimiDL is mainly targeted to mobile devices and embedded systems, where resources (especially RAM and CPU) are limited, it is developed by a group of researchers at the Department of Electronics, Informatics and Bioengineering (DEIB) of the Politecnico di Milano. It is entirely written in C++17 and exploits all the features of the language that can improve performance or usability. It is used in PeakLens, an Android application that uses Deep Learning to recognize mountains. PolimiDL is not a training framework. The focus of the project is executed, enabling use-cases and optimization which are not possible (or at least not easily implementable) in all-in-one solutions. PolimiDL assumes that the model has already been trained in the framework of your choice and later converted into a compatible format. The trained models can be converted using PolimiDL Converter [15], a tool that extracts weights from models and binds them in the network definition, written in PolimiDL. It applies some optimizations as early as possible to span the model life-cycle.

TensorFlow Lite

TensorFlow Lite [21] is a mobile library for deploying models on mobile, microcontrollers and other edge devices, it is widely used in Computer Vision, audio and text recognition tasks. This framework is an open-source end-to-end platform for device-based deep learning inference, released by Google in 2017. It supports several languages, including Java, Swift, C++ and Python, to ensure cross-platform support as well. Tensorflow picks a new model or retrains an existing one, then converts a TensorFlow model into a compressed flat buffer with the TensorFlow Lite Converter [20], takes the compressed ".tflite" file and loads it into a mobile or embedded device and finally quantizes by converting 32-bit floats to more efficient 8-bit integers or run on GPU. This framework also allows model optimizations, such as quantization to reduce model size and latency, with minimal or no accuracy loss.

PyTorch Mobile

PyTorch Mobile [16] is a deep learning framework to build neural networks. It is based on Torch, a scientific computing framework with wide support for machine learning algorithms. PyTorch replaces the engine of Torch with a Python-based, GPU-accelerated dynamic translator. It is open-source software released under the Apache 2.0 license. PyTorch is mainly used for deep learning models, including sequence models, reinforcement learning models, and relational models. It is written in Python. In 2020, PyTorch Mobile announced a new prototype feature supporting Android's Neural Networks API (NNAPI) to expand hardware capabilities to execute models quickly and efficiently. The initial release included support for popular linear convolutional and multilayer perceptron models. PyTorch Mobile provides an end-to-end workflow that simplifies the research to the production environment for mobile devices and privacy-preserving features via federated learning techniques. It currently supports deploying pre-trained models for inference on both Android and iOS platforms.

ONNX

ONNX [12] (Open Neural Network Exchange) is an open format built to represent machine learning models. ONNX defines a common set of operators, the building blocks of machine learning and deep learning models, and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers. It was released by the PyTorch team at Facebook in 2017, it is an open ecosystem that empowers AI developers to choose the right tools as their project evolves. ONNX provides an open-source format for AI models, both deep learning and traditional ML. It defines an extensible computation graph model, as well as definitions of built-in operators and

standard data types. ONNX defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. Each computation dataflow graph is structured as a list of nodes that form an acyclic graph. Nodes have one or more inputs and one or more outputs. Each node is a call to an operator. The graph also has metadata to help document its purpose, author, etc. Operators are implemented externally to the graph, but the set of built-in operators are portable across frameworks. Every framework supporting ONNX will provide implementations of these operators on the applicable data types. ONNX is widely supported and can be found in many frameworks, tools, and hardware.

In this work, ONNX has been used for the conversion of TFLite models to the PyTorch Mobile format.

2.1.3. Optimization Techniques

Computer vision applications require high real-time performance as many samples require inference processing every second, consuming a lot of energy and throughput. This could critically increase device performance in real-world scenarios, which is why efforts are made to accelerate inference processes through both hardware and software.

Concerning **Hardware acceleration**, additional hardware is required to help the CPU during inference procedures; this is not an automatic optimization, in fact, any Deep Learning framework should provide primitives or interfaces to increase performance. On the other hand, as far as **Software acceleration** is concerned, there are various techniques to analyze the pattern graph, without modifying it, and convert it into instructions suitable for the hardware. Some low-level libraries are especially suited to GPUs, intercepting CNN models and converting them into ad-hoc instructions for graphics cards so that operations such as convolutions, normalization and pooling take advantage of GPU parallelism. Other techniques can improve memory reuse and prevent cache misses by generating code optimized for the target hardware after modifying the model graph. Neural networks may contain redundancies so it is necessary to handle repetitions to speed up the inference process.

Algorithmic acceleration may have been applied to models that have undergone a modification or architectural change. It is possible to prune less important weights or filters from a model with a small reduction in accuracy. The more weights are pruned, the more connections between neurons are removed. Quantization replaces floating-point weights with low-precision, more compact representations. Through the use of this technique, networks will be smaller and thus calculations faster, but this leads to a loss of informa-

tion and distortion of the network architecture. Deep learning frameworks offer a number of optimizations and allow developers to enable and customize them.

Year	Name	Metrics
2018	AI Benchmark: Running Deep Neural Networks on Android Smartphones [28]	Latency
2019	Accelerating Deep Learning inference on mobile systems	Latency Threads
2019	AI Benchmark: All About Deep Learning on Smartphones in 2019 [29]	Latency
2019	AIoT Bench: Towards Comprehensive Benchmarking Mobile and Embedded Device Intelligence	-
2019	Benchmarking and Optimizations for Mobile Computer Vision Applications	Image pre-processing Latency Throughput
2019	EmBench: Quantifying Performance Variations of Deep Neural Networks across Modern Commodity Devices	Latency Throughput Batch-size Accuracy TimePerLayer
2020	Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices [31]	Latency Accuracy
2020	Deep Learning on Mobile and Embedded Devices: State-of-the-art, Challenges, and Future Directions	Latency Accuracy EnergyConsumption
2021	Efficient Execution of Deep Neural Networks on Mobile Devices with NPU [33]	Latency Accuracy

Table 2.2: Past frameworks comparison and metrics used.

2.1.4. Evaluation

During inference procedures, a huge variety of possible factors must be analyzed such as different layers, parameters, dimensions and several combinations of models and frameworks. A benchmark can simply be composed of the combination of a model of interest and a framework, and for each of these possible combinations, certain metrics can be calculated to measure the quality of the output from the inference and certain quantities

to analyze its performance.

Table 2.2 represents the main metrics used in past years in the analysis of the most recent benchmarks, it can be seen that two metrics have been used most frequently: accuracy and latency.

Accuracy can be defined as the ratio of the number of correct predictions to the total number of predictions; it is a classification metric used to evaluate classification models. It is combined with other metrics, such as statistical metrics, precision, recall and mean absolute value. If the objective of benchmarking is to measure the effectiveness of classification models, these metrics can be very useful.

Latency, on the other hand, is the time taken to perform inference, generally expressed in milliseconds, and since a single measurement is not significant enough to get correct data, tests are repeated on a significant number of samples. It is very important to choose the number of inferences to execute for each test according to the objectives of the benchmark. AI Benchmarks [28] [29] performed a huge sample collection over thousands of devices, where each device runs a dozen tests, with a single inference of various Computer Vision tasks; then the average latency is calculated among the same model devices, for each test. Other researchers compute the same just described metrics in an image classification scenario, where models recognize up to 1000 classes, but they measured 5000 samples: e.g. in [31], 5 images per class have been picked, meanwhile in [33] simply picking 5000 images from the original model dataset.

The last work done [34] delved into the ecosystem of modern DL libraries and provided quantitative results on their performance. a comprehensive benchmark including 6 representative DL libraries and 15 diverse DL models was created. The tests were run on 10 mobile devices to get a comprehensive view of the current ecosystem of DL libraries for mobile devices. For example, we find that the best-performing DL lib is highly fragmented among different models and hardware, and the gap between these DL libs can be quite large. The impact of DL libraries can outweigh algorithm or hardware optimizations, e.g., model quantization and GPU/DSP-based heterogeneous processing.

The tests carried out by Darian Frajberg, Christian Marone et al., researchers at the Politecnico di Milano, are based on 50 samples collected for each test and focus on measuring the pre-processing time [32] of the images and varying the latency by modifying the number of threads [25] involved in the execution.

Image pre-processing time is the time required to convert the input data into a format compatible with the model. For image classification tasks, the raw input data is repre-

sented by an image, whose size or representation generally does not match the input data format expected by the model. The image may need to be resized or represented with different data structures, compatible with the model. Latency may be greatly modified by the **number of threads** involved, in fact during the inference process the workload may be split or may compile the model layers according to a thread pool, depending on the impact of different frameworks on the CPUs cores.

The initialization time is another key metric to understand how a framework would behave before the execution of inference and whether preliminary operations are performed, in fact, some frameworks can apply optimizations in the initial steps, for instance by profiling the model or preallocating memory for model layers.

The memory consumption is calculated by averaging the amount of RAM used during the iterations, the consumption remains constant by changing the number of threads for the execution of the inference, it varies a lot by changing the model and especially the framework. It is generally expressed in MB, and it is a very important metric as it gives us useful information about the behavior of the framework.

Battery consumption measures the difference in capacity between start-up and the end of the inference execution, generally expressed in mAh. How memory consumption varies mainly by changing models and frameworks. The Analysis is very important for developers to understand how the framework would behave before the execution of the inference. For this reason, choosing the framework for a given model is crucial to understand if a neural network has to be reconstructed more than once, or how the framework will allocate memory.

In Chapter 4, the analysis will focus on the fundamental steps of the inference procedure: the pre-processing of the input image, the initialization of the neural network (specifying the number of threads to be used) and the execution of inference on a model. Model-dependent operations, such as initialization and inference execution, achieve different measures of time by changing the number of threads; non-model-dependent operations, such as input pre-processing, which simply transform an image into another format, memory and battery consumption, do not involve a specific number of threads.

This work will analyze some deep learning frameworks, with or without initialization optimization, varying the number of threads and showing the differences in latency when different levels of multi-thread execution are configured. In addition, when significant, different image pre-processing techniques will be adopted to see if the total time to perform an inference can be reduced. Therefore, for more than 50 samples, the metrics used will be:

1. **Initialization time**, varying the number of threads;
2. **Average Image pre-processing time**, varying methods to do that;
3. **Average Latency**, varying the number of threads, is useful to have direct feedback on the time required by one inference on average;
4. **Latency Standard Deviation**, varying the number of threads and checking the stability of latency measurements;
5. **Average throughput**, to represent how many inferences can be run in a second;
6. **Average total latency time**, given by the sum of average pre-processing time and the average latency;
7. **Average Memory Consumption**, represents the average of memory used during the iterations, on MB;
8. **Battery Consumption**, represents the consumption of battery capacity on mAh.

3 | Methods

3.1. Mobile inference frameworks

PeakLens uses a convolutional neural network to perform the image/frame skyline extraction task. It uses the PolimiDL framework, developed by a group of researchers from the Department of Electronics, Informatics and Bioengineering (DEIB) at the Politecnico di Milano and released in 2019 to perform inference on mobile devices. Advances in edge Deep Learning have led to the improvement of many frameworks (Section 2.1.2). In this Chapter, we study the performance of several frameworks that could be used to improve user experience on augmented reality mobile applications, such as PeakLens.



Figure 3.1: PeakLens image/frame skyline extraction task.

The CNN frameworks chosen are:

- **PolimiDL;**
- **TensorFlow Lite;**
- **PyTorch Mobile.**

PolimiDL was chosen because it is the framework created for PeakLens, TensorFlow Lite

because it is considered to be one of the most widely used and high-performance frameworks today, and PyTorch Mobile because it is considered an excellent framework for memory management.

3.1.1. PolimiDL

Frameworks for executing DL models on mobile and embedded systems pursue an optimized implementation on devices with limited resources, efficiently managing memory allocation and making the best use of available hardware resources. PolimiDL has been implemented to optimize the DL acceleration on mobile devices and embedded systems. Training is performed off-board, with mainstream tools such as TensorFlow or Caffe2, and the resulting models are converted into the mobile framework format for distribution. Open Neural Network Exchange Format (ONNX) proposes the standardization of model definition, to simplify the porting of models trained with different tools.

PolimiDL aims to speed up the execution time of ready-to-use models by applying multiple optimizations that increase the efficiency of operations without changing the model output. Its implementation is highly generic, with no hardware or platform-specific components; this provides an increase in performance on heterogeneous devices and simplifies maintenance, eliminating the need to address different platforms with different tools. It can be compiled for all major platforms, requiring only a very simple interface layer to interact with platform-specific code. PolimiDL takes advantage of multi-threaded execution.

The trained model is converted into a format compatible with PolimiDL while applying generation-time optimizations. Next, the model is compiled for the target architecture and compile-time optimizations are applied. Once the model has been deployed on the target device, an initialization phase applies initialization-time optimizations to determine the best memory layout for the specific CNN model. The first time a model is deployed, the initialization phase may include model profiling, which enables compile-time optimizations to determine the best scheduling approach. Finally, the model is ready to process inputs by applying run-time optimizations, which involve dynamic workload scheduling to speed up inference.

The three buffers used by the layers (input, output and temporary) are merged into one. During initialization, each layer organizes itself according to the size of input/output and the temporary data it requires. Memory requirements are often layer-specific but sometimes depend on the hardware capabilities of the device, such as the number of threads or input size in the case of Fully Convolutional Network models. The single

buffer serves to hold the data of the already demanding layer. PolimiDL implements a dynamic scheduler that allows the workload to be divided into tasks executed by different threads [25].

This framework requires matrix objects as input. For this purpose OpenCV has been used, which is a highly optimized Computer Vision library [13].

3.1.2. TensorFlow Lite

TensorFlow Lite [24] is the framework created by Google in 2017 to develop and run Deep Learning-based applications for mobile devices, derived from the TensorFlow project [19]. Moreover, it not only manages to support iOS and Android devices, thanks to C++ APIs and a specific Java wrapper for Android systems, but it also supports microcontrollers and IoT devices.

TensorFlow Lite guarantees low power consumption and high performance, incorporating hardware acceleration techniques (on CPUs, GPUs and the latest NPUs) and model optimization, even supporting multi-threaded execution, which ideally enables faster operator executions. To perform inference on TensorFlow Lite, the framework's official documentation recommends using a Buffer as the input data type for better performance. Image pre-processing tests were performed using the following conversions: via TensorImage, via OpenCV, and in standard mode. In the end, the ByteBuffer object was chosen.

Two versions will be tested for this framework:

1. **TensorFlow Lite 2.6.0:** Released on Aug 11th 2021;
2. **TensorFlow Lite 2.6.0 + XNN PACK:** the same version of the previous item, but with an additional library [22], released on July 24th 2020 by Google, that provides highly-optimized implementations of floating-point neural network operators and improves inference performances of a neural network on CPUs. Specifically, all operators have been optimized for ARM NEON cores and an optimization called operator fusion allows TensorFlow Lite to optimize the whole computational graph and to optimize it rather than executing operators one-by-one [23].

3.1.3. PyTorch Mobile

PyTorch is an open-source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, developed mainly by Meta AI(Facebook). Although the Python interface is more polished and

is the main focus of development, PyTorch also has a C++ interface.

There is a growing need to run ML models on edge devices in order to reduce latency, preserve privacy and enable new interactive use cases, PyTorch Mobile enables a seamless transition from training a model to deploying it while remaining entirely within the PyTorch ecosystem. It provides an end-to-end workflow that simplifies the research and production environment for mobile devices, it supports multi-threaded execution. It also paves the way for privacy-preserving features through federated learning techniques. PyTorch defines a class called Tensor to store and operate on homogeneous multidimensional rectangular arrays of numbers.

PyTorch uses a method called automatic differentiation. A recorder records the operations performed and then plays them backward to calculate gradients. This method is particularly effective when building neural networks to save time because parameter differentiation is directly calculated at the next step. PyTorch has various optimization algorithms used for constructing neural networks. Most of the commonly used methods are already supported. PyTorch also contains many other useful sub-modules, such as utilities for loading data and distributed training functions.

3.2. Deep Learning model

The deep learning models used are PeakLens Original, PeakLens Optimized, and Mobilenet v1 1.0.224 (Table 3.1). The PeakLens ones perform the semantic segmentation task that allows the PeakLens application to extract the skyline from the input images. Instead, the Mobilenet v1 model executes the object classification task and it has been used to compare frameworks' performances introducing an external model not specifically deployed for PeakLens' specific task.

Model	Task	Parameters	Input Size
PeakLens Original	Semantic Segmentation	429K	320x240x3
PeakLens Optimized	Semantic Segmentation	21K	320x240x3
MobileNet	Object Classification	4.24M	224x224x3

Table 3.1: CNN models architectures.

3.2.1. Peaklens Original

The PeakLens Original is a Fully Convolutional Neural Network model used in PeakLens. PeakLens Original is inspired by LeNet, an old model that recognized handwritten digits,

trained with image patches for binary classification that acquired 28x28 greyscale images. PeakLens Original retains the same idea of binary classification but accepts 29x29 RGB colored images. For each pixel, PeakLens Original calculates the probability of it being part of the skyline or not. The model is structured as a fully convolutional neural network, in fact, the LeNet Fully Connected layers have been replaced with Standard Convolution layers. It is possible to input images of entire mountains and receive as output a probability map describing the probability of each pixel being part of the skyline.

Table 3.2 presents the detailed architecture of the PeakLens Original model while [32] presents the training procedure.

Layer Type	Input Shape	Filter Shape	Stride
Conv	29 x 29 x 3	6 x 6 x 3 x 20	1
Pool (max)	24x24x20	2x2	2
Conv	12x12x20	5x5x20x50	1
Pool (max)	8 x 8 x 50	2 x 2	2
Conv	4 x 4 x 50	4 x 4 x 50 x 500	1
ReLU	1 x 1 x 500	-	1
Conv	1 x 1 x 500	1 x 1 x 500 x 2	1

Table 3.2: PeakLens Original Architecture.

3.2.2. Peaklens Optimized

PeakLens Optimized is the improved version of the PeakLens Original model, it is inspired by the MobileNet model and introduces vertically separable convolution layers replacing the standard convolution layers. Compared to the previous model, several optimizations have been implemented. All possible bottlenecks in the network, identified both in the layers and in the filters, have been replaced or modified. A new main layer, the new Depthwise Separable Convolution, has been added to PeakLens Original. To make this change, PolimiDL had to be modified, because previously it did not support this layer, and ad-hoc support for both Depthwise and Pointwise Convolution has been added. This resulted in better performance with this model, allowing a decrease in parameters and computational cost.

Table 3.3 presents the detailed architecture of the PeakLens Optimized model from [32].

Layer Type	Input Shape	Filter Shape	Stride
Conv	29 x 29 x 3	3 x 3 x 3 x 32	1
ReLU	29 x 29 x 3	-	1
DepthwiseSeparableConv	27x27x32	3x3x32x32	2
DepthwiseSeparableConv	13x13x32	3x3x32x64	1
DepthwiseSeparableConv	11x11x64	3x3x64x64	2
DepthwiseSeparableConv	5 x 5 x 64	3 x 3 x 64 x 128	1
Conv	3 x 3 x 128	3 x 3 x 128 x 2	1

Table 3.3: PeakLens Optimized Architecture.

3.2.3. Mobilenet v1

MobileNet is a class of convolutional neural network models [27]. Its main strengths are its efficiency and architecture, in fact, it is very well suited to perform inference on mobile devices and for Computer Vision fields such as image classification. It has introduced a novelty in architectural layers using vertically separable convolution chains with serial normalization and ReLU. Several versions of the MobileNet models were trained on the ImageNet dataset, on which they performed very well compared to other ImageNet classification models [28]. The version chosen for testing was the largest one: MobileNet v1 1.0 224; for simplicity and readability, we will refer to it as MobileNet.

Table 3.4 presents the detailed architecture of the MobileNet model from [7].

Layer Type	Input Shape	Filter Shape	Stride
Conv	224 x 224 x 3	3 x 3 x 3 x 32	2
Depthwise Conv	112 x 112 x 32	3 x 3 x 1 x 32	1
Conv	112 x 112 x 32	1 x 1 x 32 x 64	1
Depthwise Conv	112 x 112 x 64	3 x 3 x 1 x 64	2
Conv	56 x 56 x 64	1 x 1 x 64 x 128	1
Depthwise Conv	56x56x128	3 x 3 x 1 x 128	1
Conv	56x56x128	1x1x128x128	1
Depthwise Conv	56x56x128	3 x 3 x 1 x 128	2
Conv	56x56x128	1x1x128x256	1
Depthwise Conv	28x28x256	3 x 3 x 1 x 256	1
Conv	28x28x256	1x1x256x256	1
Depthwise Conv	28x28x256	3 x 3 x 1 x 256	2
Conv	14x14x256	1x1x256x512	1
5 x {Depthwise Conv + Conv}	14 x 14 x 512	3 x 3 x 1 x 512	1
	14 x 14 x 512	1 x 1 x 512 x 512	
Depthwise Conv	14x14x512	3 x 3 x 1 x 512	2
Conv	7 x 7 x 512	1 x 1 x 512 x 1024	1
Depthwise Conv	7 x 7 x 1024	3 x 3 x 1 x 1024	2
Conv	7 x 7 x 1024	1 x 1 x 1024 x 1024	1
Pool (avg)	7 x 7 x 1024	7 x 7	1
FC	1 x 1 x 1024	1024 x 1000	

Table 3.4: MobileNet Architecture.

3.3. Model conversion

PyTorch Mobile uses a different format for models. In order to use the same models that are already drawn, it is necessary to convert them, one of the most effective ways is through ONNX, which as mentioned in the Chapter 2.1.2 is compatible with many frameworks.

The conversion implies multiple steps. The first step was carried out thanks to *tf2onnx* which is a library provided by ONNX and supports also TFLite models, the output is a ".onnx" file containing the weights of the layers. Then, the model architectures was defined following the PyTorch standard and the weights extracted from the ONNX format file were used to initialize the PyTorch models. Finally, these models were quantized,

optimized for mobile and saved in the PyTorch Mobile format (".ptl"). The last step was executed using specific functions provided by the PyTorch library.

4 | Evaluation

4.1. Environment

The evaluation process defines considerations based on the comparison of certain performance metrics, which examine time and consumption during benchmarking. To obtain device-specific results, benchmarking tests were performed on target devices. The environment in which the tests were performed is an important factor in ensuring the reliability of the data. Since benchmark tests are performed in multi-threads, users will have to follow certain requirements otherwise measurements may be compromised. For example, a too-low battery level or battery on charge could compromise battery consumption, or background processes could alter memory space allocation and consequently increase the time.

For this reason, certain settings have been requested from users:

- The battery level of the device must always be above 80% and not be on charge;
- The device must always be with the display on, neither on standby nor with the screen locked;
- No applications must be run in the background during the benchmark tests.

4.1.1. Devices

The target devices that were used to perform the benchmarking tests have various technical specifications that have been reported in Table 4.1. The device models utilized cover a period of 9 years, from 2014 to 2022. The oldest Android version is Android 5 (Asus ZenFone 2), while the newest is Android version 12. RAM memory equipment is between 2 and 8 GB. We tested one Dual-Core, two Quad-Core, one Hexa-Core and eight Octa-Core devices. We can point out that from 2017 onwards the market has been dominated by Octa-Core processors, and we can also see that the most widely used architecture in mobile devices and embedded systems is ARM. The processors mounted on our devices are mainly Kryo microarchitectures, based on ARM but developed independently

by Qualcomm, and a large number of Cortex microarchitectures. Most devices divide the processes into 8 threads, only the four oldest devices divide the workload into 4 and 6 threads.

Table 4.1 presents the devices used for evaluation, which are ordered by year, the oldest is from 2014 up to 2022.

Device name	Year	RAM	Android	Threads	Devices specifications
Motorola Nexus 6	2014	3	7	4	4x 2.7 GHz Krait 450
LG Nexus 5X	2015	2	8	6	4x 1.4 GHz Cortex-A53 2x 1.8 GHz Cortex-A57
Asus ZenFone 2	2015	2	5	4	2x 1.6 GHz Intel Atom Z2560
LG G5	2016	4	8	4	2x 2.15 GHz Kryo 2x 1.6 GHz Kryo
ONEPLUS 5t	2017	6	10	8	4x 2.45 GHz Kryo 4x 1.9 GHz Kryo
Samsung S9	2018	4	10	8	4x 2.8 GHz M3 Mongoose 4x 1.7 GHz Cortex-A55
Redmi Note 8T	2019	4	11	8	4x 2.0 GHz Kyro 260 4x 1.8 GHz Kyro 260
Mi 9 Lite	2019	6	10	8	2x 2.2 GHz Kryo 360 6x 1.7 GHz Kryo 360
Mi 10 Lite	2020	6	11	8	1x 2.4 GHz Kryo 475 Prime 1x 2.2 GHz Kryo 475 Gold 6x 1.8 GHz Kryo 475 Silver
Samsung S10 Lite	2020	8	12	8	1x 2.84 GHz Kryo 485 3x 2.42 GHz Kryo 485 4x 1.8 GHz Kryo 485
Xiaomi 11t	2021	8	12	8	1x 3.0 GHz Cortex-A78 3x 2.6 GHz Cortex-A78 4x 2.0 GHz Cortex-A55
Samsung S22	2022	8	12	8	1x 2.99 GHz Cortex X2 3x 2.49 GHz Cortex A710 4x 1.78GHz Cortex A510

Table 4.1: Devices used in the benchmark.

4.1.2. Application

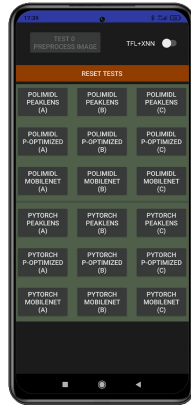


Figure 4.1: Benchmark application tests interface.



Figure 4.2: Tests interface running PyTorch Mobile test.



Figure 4.3: Tests interface running test with XNN PACK enabled.



Figure 4.4: Benchmark Completed and ready for send result via e-mail.

In order to carry out the benchmarking tests and to have all the data concerning the execution of all the frameworks to be tested on the different devices, the existing Android application used in the work [25] was updated. The user interface is very intuitive and includes an introductory tutorial for users. The main interface of the application presents all the cases to be tested (Figure 4.1). Each button will start a different test (Figure 4.2),

and as soon as it is pressed the other buttons will be disabled until the test is completed. When the Benchmark is complete (Figure 4.4), the results can be sent via e-mail as CSV values and a txt file will be saved in the device memory. When the application marks the test as completed, it will restart silently to avoid reusing any cached data or unreleased libraries. This procedure was very important to ensure data reliability. The button in the top right-hand corner will start the image pre-processing tests. At the top left, we can enable XNN PACK for TensorFlow (Figure 4.3). If the tests should return unexpected values, due to Android background processes, it is necessary to repeat the tests by pressing the 'Reset Test' button, which will re-enable all buttons again.

4.1.3. Execution

Each benchmark test performed is defined by:

- the framework used;
- the CNN model exploited to make the inference;
- the configuration of the threads.

Before testing, it is good to check if the three requirements described in Chapter 4.1 are being met (battery > 80% and not charging, display always On, no background processes). The multi-thread configuration is tested to analyze multi-thread execution.

The three thread configurations tested are:

- **Min(4, Threads):** equal to 4 threads running. Mainly on newer devices where, as mentioned in Chapter 4.1.1, the number of Cores is usually greater than 4. (In the tests we also analyzed the behavior of a Dual-Core device which might have been the only one to achieve a lower value, but in our case, it also divides the work into 4 threads);
- **Max(1, Threads-1):** all available threads but one. This exact number of threads should ideally achieve good results since one thread is always left free to perform Android background operations;
- **Threads:** all threads, for maximum parallelism. It is the maximum number of usable threads, which depends on the device.

The total number of tests per device is 4 (framework) x 3 (models) x 3 (thread configuration), that is 36 tests to measure configuration and inference times, plus 5 additional tests to evaluate image pre-processing performance, so 41 tests in total.

4.2. Frameworks comparison

The performance analysis is divided into several steps. First, the pre-processing time to prepare the input is analyzed. For each framework, the best solution will be chosen. Then, **initialization times**, **inference metrics** (latency + standard deviation and throughput) and **consumption** (memory and battery) will be evaluated. Frameworks running the same CNN model individually will be compared. The results will then be aggregated highlighting which framework performs better, to understand how a CNN model may change its behavior depending on the framework running it.

4.2.1. Metrics

The main goal of the current evaluation is to compare CNN frameworks' performances running different CNN models on a set of various devices. To better understand and classify devices' behaviors running the evaluation tests we decided to use the following metrics:

- **Pre-processing time** is the time required to convert the input into the specific format that will depend on the framework that can only handle an input if it is presented in a specific way that depends on it. 50 pre-process executions have been averaged to define the pre-process value. It might have an impact on the overall performance of the framework. It is measured in milliseconds;
- **Initialization time** is the average time required by the CNN framework to build the neural network following the CNN model characteristics. Thanks to the CNN model, the framework prepares a precise number of layers, as well as the type of layers, and it is suitable to analyze a fixed number of parameters. It is measured in milliseconds;
- **Latency** is the metric that returns direct feedback on the average time required by each device to run a single inference. One warm-up-inference run is executed before the measurements. 50 consecutive inference iterations are executed and averaged to define the latency value. It is measured in milliseconds;
- **Throughput** exploits latency results looking at it from a different perspective since it measures the average number of inferences per second;
- **Memory Consumption** is the average RAM consumption during the 51 iterations performed. It depends on the model received as input and the framework that will execute it. Each framework manages differently the way it preallocates memory. It

is measured in MegaByte;

- **Battery Consumption** is the consumption of the device's battery capacity in mAH. Besides depending on the frameworks, the values will depend a lot on battery life, older batteries will have much higher consumption than newer batteries.

4.2.2. Image pre-processing

When the model has been deployed for the CNN framework, it will be ready to process the input and make inferences about it. Each framework has different characteristics as explained in Chapter 3.1. In our case PolimiDL needs an OpenCV matrix, TensorFlow Lite requires a ByteBuffer as input and PyTorch Mobile uses a Tensor class. OpenCV is an optimized library for processing and handling matrices. Three alternatives were tested for the construction and optimization of the ByteBuffer input for the TensorFlow Lite framework. To do so, have been tested on target devices and have been analyzed the following alternatives:

- **TensorFlow Lite via TensorImage (0.2.0) (TFLTI)**: the input is prepared to take advantage of a TensorFlow Lite Android support library [18] that has a suite of basic image manipulation methods. To convert the image into the tensor format required by the TensorFlow Lite Interpreter, the TensorImage is created to be used as input. Specifically, the TensorImage class presents a simple constructor that directly receives the Bitmap input. Then, once the TensorImage has been created, it is sufficient to pass it to the ImageProcessor class. This class is responsible for image manipulation to perform resize, rotation, normalization, and quantization operations if necessary. At this point, the input image is ready to be scanned as a ByteBuffer Object;
- **TensorFlow Lite via OpenCV (TFLOCV)**: the input is prepared using OpenCV library [13] methods to manipulate the image Bitmap to obtain a matrix Object. Once the matrix Object has been obtained, it is inserted in the ByteBuffer;
- **TensorFlow Lite via Standard Manual ByteBuffer (TFLMB)**: the bitmap input is manually scanned through nested cycles, in order to analyze each bitmap pixel. Then, pixels' values are inserted one by one in the ByteBuffer;
- **PolimiDL via OpenCV (PDLOCV)**: the input uses OpenCV in the same way as explained above;
- **PyTorch Mobile via TensorImageUtils (PTMTIU)**: the input required is a class called Tensor to store and operate on homogeneous multidimensional rectangular

arrays of numbers.

<i>Device</i>	TFLTI	TFLOCV	TFLMB	PDLOCV	PTMTIU
Asus ZenFone 2	8,58 ± 1,90	10,0 ± 2,59	53,0 ± 1,70	10,0 ± 0,55	94,5 ± 2,88
LG G5	2,65 ± 1,27	1,79 ± 0,60	6,46 ± 1,70	1,16 ± 0,11	8,22 ± 1,82
Mi 10 Lite	1,70 ± 0,62	1,61 ± 0,45	3,93 ± 0,82	1,15 ± 0,11	5,73 ± 0,96
Mi 9 Lite	3,12 ± 1,61	3,18 ± 1,02	7,32 ± 1,03	1,96 ± 0,15	11,1 ± 2,24
Nexus 5X	7,05 ± 2,57	5,20 ± 1,50	28,6 ± 3,39	4,54 ± 0,38	42,0 ± 3,52
Nexus 6	4,83 ± 3,30	5,15 ± 2,98	12,4 ± 3,45	3,32 ± 1,40	18,6 ± 7,23
ONEPLUS 5t	2,28 ± 0,94	1,98 ± 0,31	6,98 ± 1,30	1,74 ± 0,16	10,3 ± 1,70
Redmi Note 8T	2,63 ± 1,45	2,57 ± 0,95	8,08 ± 0,97	2,11 ± 0,12	11,9 ± 1,44
Samsung S10 Lite	1,44 ± 0,77	1,30 ± 0,37	3,45 ± 1,85	0,92 ± 0,08	5,02 ± 0,87
Samsung S22	2,13 ± 0,79	1,87 ± 0,61	4,17 ± 1,77	1,22 ± 0,18	5,70 ± 0,76
Samsung S9	2,95 ± 1,76	3,04 ± 1,44	6,65 ± 1,86	2,21 ± 0,69	9,67 ± 2,29
Xiaomi 11t	1,24 ± 0,62	1,29 ± 0,42	3,07 ± 1,43	0,99 ± 0,28	3,96 ± 0,77

Table 4.2: Image pre-processing times with standard deviation - Bold values indicate the pre-processing strategy that requires the lowest time on devices. Values are expressed in milliseconds [ms].

Table 4.2 presents the image pre-processing times for the TensorFlow Lite (3 alternatives - TFLTI, TFLOCV, TLFMB), PolimiDL (1 solution - PDLOCV) and PyTorch Mobile (1 solution - PTMTIU) frameworks. In order to obtain more reliable results, the values are calculated by performing 50 image pre-processing procedures on each device and averaging them. The best image pre-processing is indicated in bold for each device. It is evident that considering only this metric, the best framework to choose is PolimiDL since, by directly exploiting the OpenCV library to prepare the input, it obtains the best pre-processing time in all cases except one (Asus ZenFone 2) where it still comes very close to the best value. The second framework with the best image pre-processing performance is TensorFlow Lite and again the best strategy for processing its input is to take advantage of the OpenCV library, as it is necessary to convert the processed OpenCV matrix to the Byte-Buffer format, the pre-processing times are similar to those of PolimiDL, but slightly increased to perform this additional conversion. The strategy with TensorImage is also a good alternative to OpenCV, as the pre-processing times are still quite acceptable and in rare cases even better than OpenCV. The alternative Manual ByteBuffer is to be discarded, as the pre-processing times increase considerably compared to the better strategy. The framework with the worst performance for image pre-processing is PyTorch Mobile. It uses TensorImageUtils and converts the image data into a Tensor.

4.2.3. Initialization

In this Section, for each CNN model, the initialization times of the proposed frameworks are first compared and then some framework-specific behavior is discussed to emphasize how each framework affects the execution of CNN models.

In Table 4.3 we can see that PolimiDL has a much higher initialization time than other frameworks. The reason for it is closely related to the memory optimization operation that PolimiDL performs at this stage. This optimization is called model profiling and allows PolimiDL to optimize the memory computation for each layer (explained in more detail in Section 3.2.1). TensorFlow Lite is the quickest to perform the initialization operations, followed by TensorFlow Lite + XNN and PyTorch Mobile, which having acceptable times and are significantly faster than PolimiDL, but are greater than TensorFlow Lite. PeakLens Optimized has the best initialization times for all frameworks except for PolimiDL, which is more performance-oriented with PeakLens Original, followed by MobileNet. PyTorch Mobile has times in line with TensorFlow Lite for PeakLens Optimized, it increases for the Original version, and with MobileNet it has significantly higher times. TensorFlow Lite and TensorFlow Lite + XNN prefer PeakLens Optimized, followed by the Original version and lastly MobileNet. The three models analysis is explained better in Tables 4.8, 4.9, and 4.10.

<i>Model</i>	<i>Threads</i>	PolimiDL	PyTorch	TensorFlow	TF+XNN
PeakLens Original	Min(4,T)	7318,56	74,23	14,15	20,30
	Max(1,T-1)	7738,97	30,78	5,14	16,31
	T	8066,87	33,17	5,75	22,09
PeakLens Optimized	Min(4,T)	14308,90	17,21	7,36	12,28
	Max(1,T-1)	16534,42	14,30	5,02	12,79
	T	17274,91	13,63	4,56	14,61
MobileNet	Min(4,T)	11837,84	288,26	20,72	75,19
	Max(1,T-1)	13017,88	179,67	5,57	71,27
	T	14074,95	191,30	5,54	72,36

Table 4.3: Frameworks comparison of **initialization time** varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in milliseconds [ms].

4.2.4. Inference

Once an input has been passed to the CNN frameworks, to evaluate frameworks' performances running the described models, the metric used is the latency. The following tables give an overview of latency computations, for each model and framework.

From Table 4.4, it is evident that the best framework for processing inferences is TensorFlow Lite + XNN. Also from Table 4.5, we can see that TensorFlow Lite + XNN processes many more inferences per second than the other frameworks. The fastest model for all frameworks is PeakLens Optimized, the latency for this model is significantly lower than for the other frameworks, it also has the lowest standard deviation and is therefore the most stable. Concerning PeakLens Optimized compared to PolimiDL (as we can see from Table 4.13), we find a performance gain on the part of both TensorFlow Lite + XNN and PyTorch, while TensorFlow Lite performs worse. PeakLens Original, on the other hand, has fairly high latency compared to PyTorch Mobile, while TensorFlow Lite also worsens, albeit only slightly (when working with a few threads, it has an even better latency but with a significantly higher standard deviation). TensorFlow Lite + XNN is the only one to perform better (in this case with the Max(1, T-1) configuration it has a worse latency but with a lower standard deviation, Table 4.13). MobileNet performs very well with TensorFlow Lite + XNN, with TensorFlow Lite it has better times than PolimiDL and worsens its performance with PyTorch Mobile (Table 4.15).

<i>Model</i>	<i>Threads</i>	PolimiDL	PyTorch	TensorFlow	TF+XNN
PeakLens	Min(4,T)	290,79 ± 10,23	1264,98 ± 46,36	288,76 ± 27,06	284,47 ± 10,29
	Max(1,T-1)	301,31 ± 20,02	1267,09 ± 50,78	335,77 ± 37,83	309,17 ± 11,49
	T	301,05 ± 19,31	1205,33 ± 29,58	311,96 ± 31,05	288,43 ± 13,75
PeakLens Opt	Min(4,T)	64,83 ± 4,53	48,50 ± 9,74	96,28 ± 7,84	38,98 ± 5,77
	Max(1,T-1)	65,62 ± 6,01	53,70 ± 6,70	90,47 ± 8,38	47,07 ± 6,75
	T	60,60 ± 3,80	58,26 ± 10,07	93,72 ± 11,86	47,58 ± 8,39
MobileNet	Min(4,T)	144,96 ± 12,01	148,12 ± 15,17	110,02 ± 14,12	80,06 ± 9,52
	Max(1,T-1)	154,99 ± 10,93	251,23 ± 26,35	137,89 ± 11,46	93,25 ± 8,56
	T	160,09 ± 10,68	311,39 ± 50,67	139,98 ± 15,21	95,98 ± 12,83

Table 4.4: Frameworks comparison of **latency** varying CNN models with devices aggregation and standard standard deviation values - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in milliseconds [ms].

<i>Model</i>	<i>Threads</i>	PolimiDL	PyTorch	TensorFlow	TF+XNN
PeakLens Original	Min(4,T)	8,48	2,79	8,15	12,33
	Max(1,T-1)	8,53	3,01	8,23	12,86
	T	8,46	2,94	8,21	12,53
PeakLens Optimized	Min(4,T)	31,91	52,58	29,96	67,77
	Max(1,T-1)	32,52	44,42	22,69	56,10
	T	31,68	37,76	20,33	52,93
MobileNet	Min(4,T)	13,68	21,79	34,91	39,20
	Max(1,T-1)	12,66	11,94	22,55	30,37
	T	12,39	8,61	19,45	28,47

Table 4.5: Frameworks comparison of **throughput** varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are the average number of inferences per second.

4.2.5. Consumption

This Section compares the memory and battery consumption caused by the execution of inferences.

We can see that PolimiDL is the framework that consumes the least RAM memory by the analysis of Table 4.6. For PeakLens Original, after PolimiDL, TensorFlow Lite + XNN is the one that consumes the least, followed by PyTorch Mobile, TensorFlow Lite on the other hand consumes much more memory. For PeakLens Optimized, PyTorch Mobile consumes little and is very similar to TensorFlow Lite + XNN, TensorFlow Lite is also the worst. As far as MobileNet is concerned, TensorFlow Lite consumes the least, followed by TensorFlow Lite + XNN and PyTorch Mobile. The most comprehensive analyses are performed in Tables 4.20, 4.21 and 4.22, where we can see that the impact on memory remains constant by changing the number of threads running, but only changes according to the framework and model used.

<i>Model</i>	<i>Threads</i>	PolimiDL	PyTorch	TensorFlow	TF+XNN
PeakLens Original	Min(4,T)	30,95	74,77	119,85	62,61
	Max(1,T-1)	31,22	74,28	119,15	61,41
	T	31,20	74,55	119,01	60,64
PeakLens Optimized	Min(4,T)	33,91	43,04	67,63	43,56
	Max(1,T-1)	33,78	43,10	67,35	42,47
	T	33,73	42,93	67,42	42,51
MobileNet	Min(4,T)	21,06	52,50	32,07	47,00
	Max(1,T-1)	21,23	52,29	31,96	47,36
	T	21,10	52,29	32,20	47,16

Table 4.6: Frameworks comparison of **memory consumption** varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in megabyte [MB].

From Table 4.6 we can see that PeakLens Optimized on PyTorch Mobile has the best power consumption, while PeakLens Original and MobileNet consume less on TensorFlow Lite and TensorFlow Lite + XNN. However, to better analyze the battery consumption data, we should look at Tables 4.23, 4.24 and 4.25, where we can see that consumption varies mainly according to the device model being tested. Newer models will have very low consumption, older models will have higher consumption. Except in rare cases, we can see that PolimiDL is the framework that consumes the most battery power with all three models.

<i>Model</i>	<i>Threads</i>	PolimiDL	PyTorch	TensorFlow	TF+XNN
PeakLens Original	Min(4,T)	7,55	8,64	2,79	2,39
	Max(1,T-1)	4,14	8,00	3,11	2,75
	T	4,26	8,24	2,97	3,02
PeakLens Optimized	Min(4,T)	3,74	1,02	1,17	1,51
	Max(1,T-1)	3,32	0,89	1,23	1,50
	T	3,43	0,88	1,67	1,34
MobileNet	Min(4,T)	3,39	2,10	1,43	2,13
	Max(1,T-1)	2,95	2,86	1,77	1,60
	T	6,82	3,19	1,81	1,79

Table 4.7: Frameworks comparison of **battery consumption** varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in milliampere hour [mAh].

PeakLens Original	PolimiDL		T	PyTorch		T	TensorFlow2.6.0		TensorFlow2.6.0 + XNN			
	Min(4,T)	Max(1,T-1)		Min(4,T)	Max(1,T-1)		Min(4,T)	Max(1,T-1)	Min(4,T)	Max(1,T-1)	T	
Asus ZenFone 2	33639,23	34448,39	33871,68	93,77	69,14	68,92	9,98	10,01	10,03	38,82	37,32	36,86
LG G5	6600,57	6580,68	6622,32	23,96	25,42	24,40	5,68	5,64	6,00	19,68	19,23	58,98
Mi 10 Lite	2812,60	2865,70	2932,39	35,92	18,74	19,51	17,56	2,61	2,49	32,87	8,77	9,30
Mi 9 Lite	5762,79	6003,96	6166,38	68,08	28,42	31,60	21,66	16,22	7,24	11,02	16,20	14,85
Nexus 5X	13539,71	13315,19	17816,88	83,27	40,34	67,47	9,14	7,39	6,82	23,70	26,28	45,44
Nexus 6	4419,80	5961,79	7129,10	6064,07	42,73	49,69	55,55	1,13	7,24	15,23	9,85	15,91
ONEPLUS 5t	419,80	4269,23	4355,48	325,51	22,03	21,71	15,63	0,55	0,44	25,76	13,22	12,37
Redmi Note 5T	6796,32	7636,06	7713,76	84,46	30,99	33,20	55,33	6,22	15,56	30,57	19,04	21,52
Samsung S10 Lite	1798,25	2245,11	2304,81	25,97	17,13	17,87	2,23	1,92	1,90	10,06	9,61	9,89
Samsung S22	1010,58	1547,99	1704,60	32,85	12,30	15,63	12,12	1,77	2,11	8,82	9,08	9,83
Samsung S9	3654,84	4754,39	4988,43	49,59	42,20	29,73	10,20	5,81	5,97	18,08	19,25	25,22
Xiaomi 11t	1826,28	2071,90	2261,67	24,60	13,01	12,49	13,01	2,45	3,20	8,95	7,88	4,89
Average	7318,56	7738,97	8066,87	74,23	30,78	33,17	14,15	5,14	5,75	20,30	16,31	22,09

Table 4.8: Frameworks initialization time with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

PeakLens Optimized	PolimiDL		T	PyTorch		T	TensorFlow2.6.0		TensorFlow2.6.0 + XNN			
	Min(4,T)	Max(1,T-1)		Min(4,T)	Max(1,T-1)		Min(4,T)	Max(1,T-1)	Min(4,T)	Max(1,T-1)	T	
Asus ZenFone 2	32305,06	32410,28	32087,82	26,42	11,94	11,94	6,44	5,56	10,02	10,06	16,45	15,32
LG G5	24233,55	20680,79	24581,90	15,37	17,11	13,81	6,44	5,66	5,85	36,31	12,69	31,99
Mi 10 Lite	6172,36	7479,38	7917,36	7,12	7,74	7,28	1,99	2,07	2,13	9,01	6,30	6,13
Mi 9 Lite	13544,12	18626,21	21476,63	13,38	14,13	15,91	5,37	5,70	4,80	7,29	8,75	10,44
Nexus 5X	27633,34	24586,93	21818,29	42,24	21,31	22,50	16,03	7,57	7,26	28,81	25,74	38,91
Nexus 6	13882,20	17026,13	16922,35	31,50	25,65	24,08	6,67	10,23	6,59	10,22	13,39	10,07
ONEPLUS 5t	9405,54	12587,86	13334,42	10,77	7,67	9,12	6,26	0,47	0,39	6,62	7,26	7,25
Redmi Note 8T	15353,26	21893,97	23488,49	15,37	15,23	15,38	23,91	5,21	5,10	10,96	17,72	14,77
Samsung S10 Lite	4769,29	8053,42	8588,97	9,84	6,76	5,85	4,59	2,20	2,32	5,66	7,47	7,76
Samsung S22	2990,04	4823,90	5123,86	9,86	8,50	5,14	2,01	2,01	1,89	5,85	7,71	6,68
Samsung S9	16148,67	22530,73	24180,46	18,46	18,83	14,33	1,68	5,70	5,79	8,26	28,11	19,13
Xiaomi 11t	5269,42	7713,45	7808,38	6,23	5,99	4,80	1,72	3,44	2,51	1,91	3,02	5,81
Average	14308,90	16534,42	17274,91	17,21	14,30	13,63	7,36	5,02	4,56	5,75	12,28	14,61

Table 4.9: Frameworks initialization time with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

<i>MobileNet</i>		PolimiDI			PyTorch			TensorFlow2,6,0			TensorFlow2,6,0 + XNN		
Device	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	
Asus ZenFone 2	38541,59	38758,54	37703,78	808,21	475,75	484,96	12,40	12,60	13,93	227,95	226,44	228,49	
LG G5	12845,87	12926,44	16960,99	171,47	123,88	117,55	8,64	5,83	6,27	59,96	51,78	50,25	
Mi 10 Lite	5651,63	6566,54	6708,06	156,38	89,01	88,55	2,38	4,55	2,35	91,95	26,95	28,15	
Mi 9 Lite	10330,59	12300,60	13448,26	228,84	153,67	153,14	5,84	5,23	6,57	41,30	44,71	49,09	
Nexus 5X	22381,16	22659,17	30259,39	536,71	278,06	287,99	61,50	7,15	7,15	144,65	161,16	149,48	
Nexus 6	14076,33	15873,51	14425,57	374,62	314,71	449,59	10,62	6,87	7,00	102,07	102,55	114,31	
ONEPLUS 5t	6952,30	7959,18	8597,81	176,71	127,06	127,68	1,06	5,72	1,40	43,88	49,34	50,40	
Redmi Note 8T	12633,46	15721,37	15917,34	365,58	148,18	147,64	111,62	5,82	5,62	55,54	58,94	60,69	
Samsung S10 Lite	3863,88	4724,53	5016,43	168,53	75,49	74,81	13,39	2,15	2,27	24,82	26,29	27,35	
Samsung S22	2518,87	3190,12	3478,46	107,00	117,69	118,17	3,16	2,88	3,42	37,66	32,09	25,57	
Samsung S9	8522,92	10967,22	11667,90	281,12	176,67	173,92	13,68	6,02	6,07	40,79	50,67	63,26	
Xiaomi 11t	3735,48	4567,37	4715,44	83,93	75,88	71,55	4,36	1,41	4,40	31,67	24,25	21,21	
Average	11837,84	13017,88	14074,95	288,26	179,67	191,30	20,72	5,37	5,54	75,19	71,27	72,36	

Table 4.10: Frameworks initialization time with Mobilenet model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

PeakLens Original Device	PolimIDL		PyTorch		TensorFlow2.6.0		TensorFlow2.6.0 + XNN					
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T			
Asus ZenFone 2	1536,01	1657,38	1537,09	9474,97	10319,21	9449,08	1696,13	2026,44	1788,51	2252,12	2522,86	2275,94
LG G5	215,48	260,24	225,44	756,40	823,27	796,84	163,27	174,51	168,27	163,73	194,39	198,35
Mi 10 Lite	108,67	100,23	99,35	268,09	235,15	245,38	110,74	94,15	92,86	62,97	51,33	55,16
Mi 9 Lite	213,88	160,42	164,60	507,32	375,77	365,98	182,10	160,41	156,77	118,55	86,74	86,88
Nexus 5X	505,03	581,05	696,12	1681,25	1114,00	1343,17	374,97	374,52	406,74	255,89	309,01	309,01
Nexus 6	270,68	270,62	313,67	703,13	805,47	722,50	333,02	586,14	521,11	160,05	198,32	156,75
ONEPLUS 5t	173,77	151,46	138,64	540,74	334,47	335,31	136,48	126,95	126,95	104,06	71,32	71,07
Redmi Note 5T	214,06	176,29	177,29	456,06	406,90	423,34	172,15	194,05	177,64	113,40	112,98	110,94
Samsung S22	61,68	58,89	58,46	206,26	184,47	187,69	65,20	64,52	64,83	44,44	41,57	43,32
Samsung S10 Lite	49,24	58,21	57,62	176,10	190,45	188,09	52,78	52,74	56,99	48,54	50,27	49,67
Samsung S9	87,03	85,99	82,71	210,31	209,72	221,05	105,43	118,43	118,95	51,13	55,64	59,63
Xiaomi 11t	54,00	54,89	61,56	199,18	206,17	225,52	71,04	66,89	63,94	38,77	40,96	44,42
Average	290,79	301,31	301,05	1264,98	1267,09	1205,33	288,76	335,77	311,96	284,47	309,17	288,43

Table 4.11: Frameworks latency with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

PeakLens Original Device	PolimIDL		PyTorch		TensorFlow2.6.0		TensorFlow2.6.0 + XNN					
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T			
Asus ZenFone 2	35,34	46,04	36,77	146,66	143,56	121,70	189,18	193,07	208,99	39,09	69,92	74,67
LG G5	9,01	68,23	25,33	56,59	74,76	59,95	10,18	15,76	25,07	25,07	15,27	22,16
Mi 10 Lite	7,41	5,12	4,21	7,18	14,94	11,01	5,68	6,27	7,16	3,57	1,72	2,88
Mi 9 Lite	11,44	5,69	20,26	14,03	9,27	11,37	5,41	4,58	13,76	1,76	3,84	5,19
Nexus 5X	41,12	12,12	205,50	67,22	241,16	16,34	14,45	38,31	8,42	16,78	5,76	13,68
Nexus 6	25,22	33,20	93,93	33,20	20,68	40,78	65,45	145,19	80,89	19,60	9,73	5,49
ONEPLUS 5t	3,02	5,81	15,19	13,85	13,62	16,26	2,26	4,76	5,23	3,72	2,45	6,65
Redmi Note 8T	4,09	8,30	8,12	3,26	31,47	21,56	5,99	28,16	14,73	1,39	9,76	10,06
Samsung S10 Lite	4,76	2,88	1,48	11,63	7,52	7,80	3,71	3,69	3,09	2,50	2,16	2,72
Samsung S22	6,92	7,90	5,03	4,85	18,62	10,55	2,30	4,78	5,12	1,83	5,04	4,28
Samsung S9	6,38	13,52	3,18	9,50	18,74	11,92	5,38	11,62	7,29	6,05	10,44	12,19
Xiaomi 11t	1,86	2,44	6,15	15,08	25,76	25,76	4,99	3,30	2,21	2,09	1,81	5,04
Average	10,23	20,02	19,31	46,36	50,78	29,58	27,06	37,83	31,05	10,29	11,49	13,75

Table 4.12: Frameworks standard deviation of latency measurements with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

Device	PeakLens Optimized			PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	289,25	321,57	287,97	309,01	308,55	310,17	676,51	537,60	548,54	250,00	211,15	205,89	252,86	20,46	20,46
LG G5	42,17	41,02	41,40	30,65	23,92	25,09	40,03	45,85	40,43	11,87	10,28	10,48	10,48	10,48	10,48
Mi 10 Lite	38,37	38,84	39,70	16,46	15,95	18,15	36,84	35,20	41,86	18,55	18,55	22,47	22,47	22,47	22,47
Mi 9 Lite	47,47	43,42	43,53	33,95	23,88	25,60	46,48	45,12	43,31	20,45	18,55	22,47	22,47	22,47	22,47
Nexus 5X	142,12	120,76	100,84	62,46	103,07	110,35	104,10	119,40	120,71	68,52	116,73	126,17	126,17	126,17	126,17
Nexus 6	66,89	70,76	61,94	39,66	47,37	47,56	92,12	76,59	64,43	25,14	29,24	26,73	26,73	26,73	26,73
ONEPLUS 5t	32,77	28,03	28,12	19,40	21,60	41,40	31,58	38,26	47,46	15,10	17,35	17,95	17,95	17,95	17,95
Redmi Note 8T	43,76	46,99	43,23	22,39	23,99	31,80	45,53	54,41	59,49	20,13	23,81	23,98	23,98	23,98	23,98
Samsung S10 Lite	12,72	14,07	14,43	9,06	12,21	13,30	16,24	25,36	27,83	7,40	10,35	11,02	11,02	11,02	11,02
Samsung S22	22,90	16,41	19,29	16,68	24,88	28,29	32,43	36,15	32,43	8,64	17,45	19,16	19,16	19,16	19,16
Samsung S9	25,66	31,51	31,90	14,51	29,83	36,62	31,90	51,96	58,82	10,50	17,45	19,16	19,16	19,16	19,16
Xiaomi 11t	13,88	14,12	14,88	7,80	9,17	10,83	16,64	23,47	35,59	7,31	7,31	7,99	7,99	7,99	7,99
Average	64,83	65,62	60,60	48,50	53,70	58,26	96,28	90,47	93,72	38,98	47,07	47,58	47,58	47,58	47,58

Table 4.13: Frameworks latency with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

Device	PeakLens Optimized			PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	13,45	23,49	12,14	68,91	19,92	39,30	47,85	21,64	71,27	35,32	20,68	26,70	26,70	20,68	26,70
LG G5	2,90	2,86	5,36	11,13	5,36	7,56	4,23	10,57	4,88	5,06	2,15	5,26	5,26	2,15	5,26
Mi 10 Lite	3,62	3,51	3,74	1,50	3,83	3,94	1,60	3,70	7,90	1,69	1,11	1,88	1,88	1,11	1,88
Mi 9 Lite	3,22	2,17	2,42	9,98	2,98	5,45	3,71	8,48	5,45	1,40	2,90	6,20	6,20	1,40	6,20
Nexus 5X	2,79	20,30	3,85	8,37	4,63	5,07	8,73	7,01	6,13	9,89	4,03	4,59	4,59	4,03	4,59
Nexus 6	8,28	8,15	6,65	9,17	5,24	21,90	8,69	17,11	13,78	4,48	4,43	13,81	13,81	4,43	13,81
ONEPLUS 5t	1,16	1,80	1,16	0,81	5,73	4,76	3,46	5,12	7,33	1,66	5,99	6,77	6,77	1,66	6,77
Redmi Note 8T	6,45	3,43	2,89	1,75	3,99	9,95	2,90	4,42	5,68	1,30	6,68	5,52	5,52	1,30	5,52
Samsung S10 Lite	0,59	0,63	0,69	1,05	2,97	4,06	0,87	3,52	5,18	2,89	3,89	3,89	3,89	2,89	3,89
Samsung S22	4,75	2,04	3,05	1,61	4,25	6,64	2,91	6,86	3,27	1,47	2,38	2,82	2,82	1,47	2,82
Samsung S9	5,69	3,13	2,43	1,75	20,22	10,37	7,26	10,07	5,50	0,72	25,35	21,49	21,49	0,72	21,49
Xiaomi 11t	1,42	1,12	1,18	0,84	1,25	1,81	1,91	2,08	6,00	3,84	1,46	1,70	1,70	3,84	1,70
Average	4,53	6,01	3,80	9,74	6,70	10,07	7,84	8,38	11,86	5,77	6,75	8,39	8,39	5,77	8,39

Table 4.14: Frameworks standard deviation of latency measurements with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

MobileNet Device	PolmiDL		T	PyTorch		TensorFlow2.6.0		TensorFlow2.6.0 + XNN				
	Min(4,T)	Max(1,T-1)		Min(4,T)	Max(1,T-1)	Min(4,T)	Max(1,T-1)	Min(4,T)	Max(1,T-1)	T		
Asus ZenFone 2	630,28	648,96	630,56	1012,50	893,37	1023,35	836,61	1045,64	997,45	586,48	609,42	595,56
LG G5	97,48	94,88	99,12	90,08	82,23	105,08	47,87	54,76	46,80	47,74	39,44	40,22
Mi 10 Lite	86,09	95,91	96,01	49,27	64,67	85,48	34,59	31,68	42,99	21,79	23,62	25,93
Mi 9 Lite	130,38	130,50	134,24	69,70	72,41	104,87	47,83	40,62	50,46	36,00	33,46	35,62
Nexus 5X	245,24	253,84	325,54	202,14	1197,47	1284,67	129,20	176,55	183,65	93,98	187,55	199,52
Nexus 6	202,02	250,94	236,97	129,74	159,86	158,95	82,17	69,62	63,17	51,33	55,60	49,90
ONEPLUS 5t	73,12	74,42	71,78	62,54	52,60	81,15	35,76	35,36	51,22	28,15	24,70	26,67
Redmi Note 5T	110,06	111,89	114,12	63,71	87,52	118,10	43,60	47,57	54,99	35,76	34,43	44,74
Samsung S10 Lite	30,37	32,11	30,97	22,08	41,09	52,97	12,75	22,06	24,48	11,68	16,04	16,59
Samsung S22	47,47	49,49	57,06	18,64	189,93	270,62	10,89	51,98	60,18	18,61	40,11	43,88
Samsung S9	50,67	80,89	88,65	32,38	135,51	397,81	22,83	59,78	78,61	17,14	37,53	57,28
Xiaomi 11t	36,34	36,02	35,69	24,62	38,15	53,58	16,11	22,90	25,78	12,07	17,10	15,87
Average	144,96	134,99	160,09	148,12	281,23	311,39	110,02	137,89	139,98	80,06	93,25	95,98

Table 4.15: Frameworks latency with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

MobileNet Device	PolmiDL		T	PyTorch		TensorFlow2.6.0		TensorFlow2.6.0 + XNN				
	Min(4,T)	Max(1,T-1)		Min(4,T)	Max(1,T-1)	Min(4,T)	Max(1,T-1)	Min(4,T)	Max(1,T-1)	T		
Asus ZenFone 2	27,50	20,63	30,86	110,40	96,66	129,99	98,05	42,38	56,30	46,91	36,48	57,10
LG G5	10,41	5,84	7,09	17,86	24,04	27,65	5,53	10,77	3,98	30,25	2,25	4,71
Mi 10 Lite	7,50	12,29	9,28	5,86	8,52	17,57	2,47	4,72	7,85	3,70	2,73	3,00
Mi 9 Lite	10,67	7,13	7,04	2,51	10,96	47,17	1,65	3,84	11,41	1,55	6,67	10,32
Nexus 5X	6,84	13,19	10,82	19,74	25,65	18,84	18,33	7,75	11,70	7,75	8,12	6,72
Nexus 6	42,04	33,00	29,57	10,58	18,09	88,31	29,75	8,87	9,28	3,34	8,10	8,10
ONEPLUS 5t	3,91	4,49	4,53	2,38	5,33	36,70	1,67	2,69	30,79	1,21	3,84	3,33
Redmi Note 8T	9,86	9,26	4,12	2,03	21,52	34,58	4,02	8,23	8,74	1,33	2,59	13,98
Samsung S10 Lite	1,91	3,02	1,10	1,61	15,08	21,95	0,80	4,98	6,52	1,07	3,64	3,35
Samsung S22	7,27	10,56	6,81	3,25	46,39	31,14	1,50	21,69	20,61	4,35	7,27	11,60
Samsung S9	12,49	8,94	14,24	3,60	38,99	133,32	3,31	17,79	11,19	1,33	23,00	29,26
Xiaomi 11t	3,68	2,86	2,69	2,27	4,91	20,81	2,39	3,78	4,14	1,28	2,77	2,48
Average	12,01	10,93	10,68	15,17	26,95	50,67	14,12	11,46	15,21	9,52	8,56	12,83

Table 4.16: Frameworks standard deviation of latency measurements with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].

PeakLens Original Device	PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	0,65	0,60	0,65	0,11	0,10	0,11	0,59	0,49	0,56	0,44	0,40	0,44
LG G5	4,64	3,84	4,44	1,32	1,21	1,32	6,12	5,73	5,94	6,11	5,14	5,04
Mi 10 Lite	9,20	9,98	10,07	3,73	4,25	4,08	9,03	10,62	10,77	15,88	19,48	18,13
Mi 9 Lite	4,68	6,23	6,08	1,97	2,66	2,73	5,49	6,65	6,38	8,44	11,53	11,51
Nexus 5X	1,98	1,72	1,44	0,59	0,90	0,74	2,67	2,67	2,46	3,91	3,52	3,24
ONEPLUS 5t	3,69	3,70	3,19	1,42	1,24	1,38	3,00	3,00	1,92	6,25	1,71	5,04
Redmi Note 8T	5,75	6,60	7,21	1,85	2,99	2,98	7,23	7,91	7,88	9,61	14,02	14,07
Samsung S10 Lite	4,67	5,67	5,64	2,19	2,46	2,36	5,81	5,15	5,63	8,82	8,85	9,01
Samsung S22	16,21	16,98	17,10	4,85	5,42	5,33	15,34	15,50	15,42	24,05	22,50	23,08
Samsung S9	20,31	17,18	17,36	5,68	5,25	5,32	18,95	18,96	17,55	20,60	19,89	20,13
Xiaomi 11t	11,49	11,63	12,09	4,75	4,77	4,52	9,48	8,44	8,41	19,56	17,97	16,77
Average	8,52	8,48	8,46	2,79	3,01	2,94	8,15	8,23	8,21	12,33	12,86	12,53

Table 4.17: Frameworks throughput with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best throughput. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are the average number of inferences per second.

PeakLens Optimized Device	PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	3,46	3,11	3,47	3,24	3,24	3,22	1,48	1,86	1,82	4,00	3,76	3,95
LG G5	23,72	24,38	24,15	32,63	41,80	39,86	24,98	21,81	24,74	47,29	48,35	48,87
Mi 10 Lite	26,06	25,75	25,19	60,77	62,70	55,08	27,15	28,41	23,89	84,26	97,31	95,40
Mi 9 Lite	21,07	23,03	22,97	29,46	41,88	39,06	21,51	22,17	23,09	48,89	53,91	44,50
Nexus 5X	7,04	8,28	9,92	9,70	9,06	9,06	9,61	8,38	8,28	14,59	8,57	7,93
ONEPLUS 5t	14,95	14,13	16,15	25,22	21,11	21,03	10,86	13,06	15,52	39,78	34,20	37,41
Redmi Note 8T	22,85	35,56	35,56	51,55	44,65	41,68	31,67	26,14	21,07	66,21	57,63	55,72
Samsung S10 Lite	78,62	71,06	69,29	110,43	81,91	75,17	61,58	39,42	135,20	96,60	96,60	90,79
Samsung S22	43,66	60,94	51,83	59,93	40,19	35,35	57,35	30,84	27,66	115,68	57,32	52,19
Samsung S9	38,98	31,74	31,34	68,93	33,52	27,31	31,35	19,25	17,00	95,25	36,75	31,61
Xiaomi 11t	72,05	70,81	67,19	128,19	108,99	92,33	60,09	42,60	28,10	136,77	112,47	125,11
Average	31,91	32,52	31,68	52,58	44,42	37,76	29,96	22,69	20,33	67,77	56,10	52,93

Table 4.18: Frameworks throughput with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best throughput. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are the average number of inferences per second.

MobileNet Device	PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	1,59	1,54	1,59	0,99	1,12	0,98	1,20	0,96	1,00	1,71	1,64	1,68
LG G5	10,26	10,54	10,09	11,10	12,16	9,52	20,89	18,26	21,37	20,95	25,36	24,86
Mi 10 Lite	11,62	10,43	10,42	20,30	15,46	11,70	28,91	31,56	23,26	45,88	42,34	38,56
Mi 9 Lite	7,67	7,66	7,45	14,35	13,81	9,54	20,91	24,62	19,82	27,78	29,89	28,08
Nexus 5X	4,08	3,94	3,07	4,95	0,84	0,78	7,74	5,66	5,45	10,64	5,33	5,01
Nexus 6	4,95	3,99	4,22	7,71	6,26	6,29	12,17	15,21	15,83	19,48	17,99	20,04
ONEPLUS 5t	13,68	13,44	13,93	15,99	19,01	12,32	27,96	28,28	19,52	35,52	40,49	37,50
Redmi Note 8T	9,09	8,94	8,76	15,70	11,43	8,47	22,93	21,02	18,18	27,97	29,04	22,35
Samsung S10 Lite	32,93	31,14	32,29	45,29	24,34	18,88	78,43	45,34	40,85	85,60	62,36	60,27
Samsung S22	21,07	20,21	17,52	53,64	5,27	3,70	91,86	19,24	16,62	53,74	24,93	22,79
Samsung S9	19,74	12,36	11,28	30,89	7,38	2,51	43,80	16,73	12,72	58,34	26,64	17,46
Xiaomi 11e	27,52	27,77	28,02	40,63	26,21	18,66	62,07	43,67	38,79	82,85	58,47	63,00
Average	13,68	12,66	12,39	21,79	11,94	8,61	34,91	22,55	19,45	39,20	30,37	28,47

Table 4.19: Frameworks throughput with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best throughput. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are the average number of inferences per second.

<i>PeakLens Original</i>		PolimiDL		PyTorch		TensorFlow2.6.0		TensorFlow2.6.0 + XNN	
Device	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	21,02	20,78	20,94	43,35	43,20	44,69	24,69	24,49	24,67
LG G5	43,63	43,61	43,63	90,45	91,16	90,80	138,39	138,45	138,45
Mi 10 Lite	36,49	36,84	37,12	83,12	83,37	83,33	131,27	132,33	131,29
Mi 9 Lite	33,35	33,43	33,43	79,75	79,78	79,76	129,25	128,22	127,90
Nexus 5X	26,55	26,53	26,76	74,33	72,80	74,18	121,24	122,10	122,10
Nexus 6	17,14	17,14	17,14	37,88	37,63	37,90	112,92	112,92	111,96
ONEPLUS 5t	30,29	30,29	29,31	80,75	77,00	76,98	123,73	125,39	125,27
Redmi Note 8T	32,14	32,24	32,41	79,02	79,00	79,04	129,14	127,06	127,10
Samsung S10 Lite	30,37	30,39	30,37	76,78	77,18	77,06	125,16	125,16	125,16
Samsung S22	27,24	27,90	28,16	81,92	81,33	81,10	133,33	129,24	129,51
Samsung S9	31,41	31,41	31,39	78,53	77,80	78,76	127,22	125,18	126,24
Xiaomi 11t	44,04	43,75	43,75	91,39	91,45	90,98	141,84	139,24	138,43
Average	30,95	31,22	31,20	74,77	74,28	74,55	119,85	119,15	119,01

Table 4.20: Frameworks **memory consumption** with **PeakLens Original** model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in megabyte [MB].

<i>PeakLens Optimized</i>		PolimiDL		PyTorch		TensorFlow2.6.0		TensorFlow2.6.0 + XNN	
Device	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	23,41	23,53	23,49	24,29	24,39	24,49	25,45	25,43	25,45
LG G5	45,63	45,53	45,63	56,14	56,69	56,18	82,02	82,04	54,88
Mi 10 Lite	38,49	38,67	38,61	48,76	48,82	48,76	75,47	75,73	75,55
Mi 9 Lite	35,27	35,43	35,43	45,73	45,71	45,71	71,43	71,53	71,82
Nexus 5X	28,65	28,55	28,75	39,59	39,59	39,59	65,71	65,71	65,71
Nexus 6	19,14	19,14	19,14	20,20	20,25	20,12	55,65	55,78	55,80
ONEPLUS 5t	32,47	31,35	31,35	42,47	42,73	41,59	68,76	69,00	70,43
Redmi Note 8T	34,27	34,27	34,31	44,69	44,69	44,69	70,78	70,80	70,80
Samsung S10 Lite	32,37	32,37	32,39	42,96	42,67	42,67	68,76	68,76	68,49
Samsung S22	36,55	36,73	36,67	50,39	50,18	49,98	71,88	70,86	70,86
Samsung S9	34,96	34,20	33,39	44,41	44,57	44,41	72,98	69,80	69,80
Xiaomi 11t	45,65	45,65	45,65	56,88	56,98	57,00	82,61	82,75	82,25
Average	33,91	33,78	33,73	43,04	43,10	42,93	67,63	67,35	67,42

Table 4.21: Frameworks **memory consumption** with **PeakLens Optimized** model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in megabyte [MB].

MobileNet Device	PolimDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	16,61	16,65	16,75	44,59	44,53	44,06	34,18	34,31	34,57	43,80	43,86	43,86
LG G5	29,76	29,67	29,86	60,12	60,12	60,12	37,90	37,59	37,73	53,98	53,98	53,98
Mi 10 Lite	27,41	27,41	26,39	57,06	57,06	57,06	35,53	35,53	35,53	50,92	50,94	50,94
Mi 9 Lite	23,33	23,65	23,75	53,76	54,00	54,00	31,47	31,47	31,49	46,96	46,96	47,67
Nexus 5X	18,22	18,24	18,18	47,88	47,88	47,88	24,98	26,08	26,24	41,90	41,90	41,98
Nexus 6	12,12	12,14	12,14	36,76	35,73	35,67	20,39	20,55	20,57	35,65	35,67	35,65
ONEPLUS 5t	21,29	22,18	22,20	51,57	51,41	51,35	32,47	29,43	32,18	45,31	45,55	45,24
Redmi Note 8T	22,33	22,33	22,33	52,98	52,98	52,98	30,45	30,45	30,45	46,84	46,69	46,53
Samsung S10 Lite	21,31	22,12	22,29	51,96	51,96	51,96	29,43	29,43	29,43	45,31	45,84	45,84
Samsung S22	0,45	0,47	0,47	53,02	51,59	52,12	33,49	33,49	33,49	47,22	46,84	47,16
Samsung S9	25,37	25,37	24,35	55,02	55,02	55,02	32,57	32,47	32,47	47,92	47,94	48,00
Xiaomi 11e	34,55	34,55	34,55	65,22	65,22	65,22	42,00	42,67	42,27	58,08	62,18	59,10
Average	21,06	21,23	21,10	52,50	52,29	52,29	32,07	31,96	32,20	47,00	47,36	47,16

Table 4.22: Frameworks memory consumption with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in megabyte [MB].

PeakLens Original	PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	5,03	8,03	9,03	9,12	11,12	10,12	5,03	6,03	12,03	2,02	17,03	15,03
LG G5	7,93	8,26	7,08	16,58	17,08	16,63	5,44	4,90	4,92	4,50	4,45	3,87
Mi 10 Lite	0,04	0,07	0,04	0,13	0,04	0,07	0,06	0,11	0,03	0,09	0,21	0,18
Mi 9 Lite	4,38	4,04	4,40	9,66	8,01	7,58	2,87	2,70	3,17	2,30	1,70	2,23
Nexus 5X	14,19	5,66	11,67	18,67	22,33	18,71	4,18	6,02	0,02	5,35	6,06	0,02
Nexus 6	11,02	12,36	12,13	25,64	24,60	25,52	12,29	11,97	12,04	5,53	6,07	6,33
ONEPLUS 5t	5,40	5,14	5,25	12,97	7,55	7,36	3,35	3,31	3,20	2,65	2,17	2,38
Redmi Note 8T	37,48	0,06	0,10	0,16	0,11	0,03	0,06	0,10	0,06	0,09	0,11	0,02
Samsung S10 Lite	0,04	0,02	0,05	4,37	0,06	0,12	0,03	0,06	0,09	0,12	0,12	0,05
Samsung S22	0,06	0,09	0,11	0,18	0,07	6,74	0,09	0,06	0,07	0,18	0,03	0,16
Samsung S9	2,97	5,93	0,21	0,09	0,05	0,03	0,05	0,02	0,05	5,93	0,02	5,93
Xiaomi 11t	2,01	0,01	1,00	6,10	5,01	6,02	0,01	2,00	0,01	0,03	1,00	0,04
Average	7,55	4,14	4,26	8,64	8,00	8,24	2,79	3,11	2,97	2,39	2,75	3,02

Table 4.23: Frameworks battery consumption with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in milliampere hour [mAh].

PeakLens Optimized	PolimiDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	11,01	8,01	5,01	7,00	6,00	5,00	7,01	8,09	9,02	12,00	8,00	11,01
LG G5	8,69	7,51	8,16	0,95	1,00	1,91	1,31	1,45	1,34	0,54	1,54	1,40
Mi 10 Lite	0,02	0,04	0,03	0,18	0,03	0,09	0,21	0,03	0,18	0,21	0,02	0,11
Mi 9 Lite	3,39	4,05	4,43	0,97	0,38	0,39	0,72	0,84	0,89	0,42	0,40	0,43
Nexus 5X	3,50	2,21	4,03	0,03	0,02	0,09	0,12	0,03	0,03	4,75	3,03	0,02
Nexus 6	6,99	6,72	5,98	1,93	1,79	1,88	3,48	2,91	2,43	1,22	1,32	1,20
ONEPLUS 5t	3,12	4,13	4,28	0,37	1,27	0,86	1,05	1,07	1,15	0,47	0,55	0,55
Redmi Note 8T	0,02	0,11	0,07	0,06	0,02	0,06	0,04	0,02	0,03	0,05	0,07	0,11
Samsung S10 Lite	0,09	0,04	0,13	0,10	0,07	0,09	0,02	0,07	0,02	0,03	0,05	0,12
Samsung S22	0,06	0,03	0,13	0,13	0,06	0,03	0,02	0,16	0,13	0,11	0,02	0,07
Samsung S9	5,93	5,93	5,93	0,06	0,03	0,09	0,03	0,13	0,07	0,04	5,93	0,05
Xiaomi 11t	2,01	1,10	3,00	0,40	0,02	0,03	0,01	0,01	0,01	0,04	0,03	1,03
Average	3,74	3,32	3,43	1,02	0,89	0,88	1,17	1,23	1,67	1,51	1,50	1,34

Table 4.24: Frameworks battery consumption with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in milliampere hour [mAh].

MobileNet Device	PolimIDL			PyTorch			TensorFlow2.6.0			TensorFlow2.6.0 + XNN		
	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T	Min(4,T)	Max(1,T-1)	T
Asus ZenFone 2	4.02	3.01	2.01	9.02	10.01	11.01	11.01	9.07	15.01	19.01	14.01	16.01
LG G5	6.08	5.24	5.14	2.78	2.06	2.57	1.80	1.67	1.66	2.16	1.62	1.33
Mi 10 Lite	0.13	0.02	0.21	0.01	0.07	0.13	0.06	0.11	0.07	0.06	0.03	0.02
Mi 9 Lite	4.11	4.40	4.70	1.39	1.31	1.88	0.88	0.43	0.89	0.91	0.83	0.84
Nexus 5X	5.18	4.65	9.72	5.33	13.65	10.04	0.07	0.02	0.09	0.04	0.02	0.06
Nexus 6	9.30	9.40	9.22	4.49	4.37	5.26	2.40	2.10	2.11	2.28	1.72	1.95
ONEPLUS 5t	3.72	4.34	4.47	1.75	1.36	1.88	0.66	1.59	1.63	0.77	0.80	0.78
Redmi Note 8T	0.04	0.16	37.48	0.13	0.02	0.13	0.11	0.09	0.05	0.07	0.04	0.05
Samsung S10 Lite	0.09	0.13	0.04	0.05	0.12	0.05	0.04	0.06	0.02	0.05	0.02	0.06
Samsung S22	0.02	0.11	6.74	0.11	0.09	0.18	0.09	0.16	0.09	0.06	0.02	0.18
Samsung S9	5.93	2.97	0.10	0.13	0.21	2.97	0.06	5.93	0.03	0.10	0.07	0.13
Xiaomi 11e	2.02	1.00	2.02	0.04	1.06	2.13	0.02	0.02	0.02	0.02	0.02	0.03
Average	3.39	2.95	6.82	2.10	2.86	3.19	1.43	1.77	1.81	2.13	1.60	1.79

Table 4.25: Frameworks battery consumption with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in milliampere hour [mAh].

5 | Conclusions and Future work

In this thesis, we analyzed the behavior of various frameworks to perform inferences for three different models, PeakLens Original, PeakLens Optimized and MobileNet. We added the PyTorch Mobile framework to a benchmarking app since PyTorch is used for many deep learning projects today, and its popularity is increasing among AI researchers. It is very flexible, has good debugging capabilities and short training duration.

PolimiDL, TensorFlow Lite, TensorFlow Lite + XNN and PyTorch Mobile were compared. The analysis began by looking at image pre-processing times and choosing the best solution for each framework. Then, we analyzed performance through initialization time and inference times, finally, we evaluated memory and battery consumption. The analysis was first evaluated for each device, thus evaluating hardware-related differences, highlighting the best performance for each framework by changing the number of threads running. Then we grouped all the values by averaging them and studied the behaviors related to the individual framework.

We can conclude that the OpenCV library is the best solution for pre-processing images while input initialization is faster with PolimiDL and the best inference times were obtained by enabling the XNN PACK on TensorFlow Lite. However, the XNN PACK increases the initialization time compared to TensorFlow Lite and memory consumption on these frameworks is never high.

Future work may focus on testing the performance of the following frameworks on other devices than Android, like iOS on Apple or HarmonyOS, Huawei's new operating system. It may also be useful to test new frameworks that may be more performing given the increasing use of Deep Learning.

Bibliography

- [1] Caffe2 Framework. <https://github.com/facebookarchive/caffe2>. Accessed: 2018-09-27.
- [2] CoreML Framework. <https://developer.apple.com/documentation/coreml>. Accessed: 2022.
- [3] Eclipse DeepLearning4J. <https://deeplearning4j.konduit.ai>. Accessed: 2022-08-11.
- [4] HiAi Framework. <https://github.com/huaweicodelabs/HiAI-Foundation>. Accessed: 2022-07-06.
- [5] Mace Framework. <https://github.com/XiaoMi/mace>. Accessed: 2022-05-30.
- [6] Metal accelerating graphics and much more. <https://developer.apple.com/documentation/metal>. Accessed: 2022.
- [7] MobileNet. <https://github.com/tensorflow/tfjs-models/tree/master/mobilenet>. Accessed: 2022-01-26.
- [8] MXNET. <https://github.com/apache/incubator-mxnet>. Accessed: 2022-08-29.
- [9] NCNN Framework. github.com/Tencent/ncnn. Accessed: 2022-08-26.
- [10] NNAPI Framework. <https://developer.android.com/ndk/guides/neuralnetworks>. Accessed: 2022-08-22.
- [11] Odin Framework. <https://github.com/wpbrasil/odin.git>. Accessed: 2020-09-04.
- [12] ONNX. <https://onnx.ai>. Accessed: 2019.
- [13] OpenCV library. <https://opencv.org>. Accessed: 2022.
- [14] PolimiDL. <https://github.com/darianfrajberg/polimidl>, . Accessed: 2020-09-20.
- [15] PolimiDLConverter. https://github.com/darianfrajberg/polimidl_converter, . Accessed: 2019-04-12.

- [16] PyTorchMobile Framework. <https://pytorch.org/mobile/home/>. Accessed: 2022.
- [17] SNPECam Framework. <https://github.com/fdchiu/SNPECam>. Accessed: 2019-01-23.
- [18] Process input and output data with the TensorFlow Lite Support Library. https://www.tensorflow.org/lite/inference_with_metadata/lite_support. Accessed: 2022-05-26.
- [19] TensorFlow. <https://www.tensorflow.org>. Accessed: 2022-01-21.
- [20] Tensorflow lite converter. <https://www.tensorflow.org/lite/models/convert>. Accessed: 2022-08-23.
- [21] TensorFlow Lite. <https://www.tensorflow.org/lite>. Accessed: 2022-01-21.
- [22] Tensorflow lite xnnpack delegate. <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/delegates/xnnpack/README.md>. Accessed: 2021-11-26.
- [23] Accelerating tensorflow lite with xnnpack integration. <https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html>. Accessed: 2021-11-26.
- [24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [25] D. Frajberg, C. Bernaschina, C. Marone, and P. Fraternali. Accelerating deep learning inference on mobile systems. pages 118–134, 2019.
- [26] D. Ghimire, D. Kil, and S.-h. Kim. A survey on efficient convolutional neural networks and hardware acceleration. *Electronics*, 11(6):945, 2022.
- [27] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [28] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool.

- Ai benchmark: Running deep neural networks on android smartphones. pages 0–0, 2018.
- [29] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool. Ai benchmark: All about deep learning on smartphones. pages 3617–3635, 2019.
- [30] L. Kechiche. Hardware acceleration for deep learning of image classification. In *2021 International Conference of Women in Data Science at Taif University (WiDSTaif)*, pages 1–5. IEEE, 2021.
- [31] C. Luo, X. He, J. Zhan, L. Wang, W. Gao, and J. Dai. Comparison and benchmarking of ai models and frameworks on mobile devices. *arXiv preprint arXiv:2005.05085*, 2020.
- [32] C. Marrone. *Benchmarking and optimizations for mobile computer vision applications. Master’s thesis*. PhD thesis, Politecnico di Milano, Milano, apr, 2019.
- [33] T. Tan and G. Cao. Efficient execution of deep neural networks on mobile devices with npu. pages 283–298, 2021.
- [34] Q. Zhang, X. Li, X. Che, X. Ma, A. Zhou, M. Xu, S. Wang, Y. Ma, and X. Liu. A comprehensive benchmark of deep learning libraries on mobile devices. In *Proceedings of the ACM Web Conference 2022*, pages 3298–3307, 2022.

List of Figures

1.1	A picture taken using PeakLens.	2
3.1	PeakLens image/frame skyline extraction task.	13
4.1	Benchmark application tests interface.	23
4.2	Tests interface running PyTorch Mobile test.	23
4.3	Tests interface running test with XNN PACK enabled.	23
4.4	Benchmark Completed and ready for send result via e-mail.	23

List of Tables

2.1	Some of the existing CNN frameworks.	6
2.2	Past frameworks comparison and metrics used.	9
3.1	CNN models architectures.	16
3.2	PeakLens Original Architecture.	17
3.3	PeakLens Optimized Architecture.	18
3.4	MobileNet Architecture.	19
4.1	Devices used in the benchmark.	22
4.2	Image pre-processing times with standard deviation - Bold values indicate the pre-processing strategy that requires the lowest time on devices. Values are expressed in milliseconds [ms].	27
4.3	Frameworks comparison of initialization time varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in milliseconds [ms].	28
4.4	Frameworks comparison of latency varying CNN models with devices aggregation and standard standard deviation values - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in milliseconds [ms].	29
4.5	Frameworks comparison of throughput varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are the average number of inferences per second.	30
4.6	Frameworks comparison of memory consumption varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in megabyte [MB].	31

4.7	Frameworks comparison of battery consumption varying CNN models with devices aggregation - Bold values indicate, for each CNN model, the best framework in a specific thread configuration. Values are expressed in milliamperere hour [mAh].	31
4.8	Frameworks initialization time with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].	32
4.9	Frameworks initialization time with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].	32
4.10	Frameworks initialization time with Mobilenet model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].	33
4.11	Frameworks latency with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].	34
4.12	Frameworks standard deviation of latency measurements with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].	34
4.13	Frameworks latency with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms].	35

4.14 Frameworks **standard deviation** of latency measurements with **PeakLens Optimized** model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms]. 35

4.15 Frameworks **latency** with **MobileNet** model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms]. 36

4.16 Frameworks **standard deviation** of latency measurements with **MobileNet** model - Bold values indicate the thread configuration, on each framework, that returns the best latency. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are expressed in milliseconds [ms]. 36

4.17 Frameworks **throughput** with **PeakLens Original** model - Bold values indicate the thread configuration, on each framework, that returns the best throughput. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are the average number of inferences per second. 37

4.18 Frameworks **throughput** with **PeakLens Optimized** model - Bold values indicate the thread configuration, on each framework, that returns the best throughput. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are the average number of inferences per second. 37

4.19 Frameworks **throughput** with **MobileNet** model - Bold values indicate the thread configuration, on each framework, that returns the best throughput. Green values indicate the best device latency, comparing all the latencies measured on different frameworks. Values are the average number of inferences per second. 38

4.20 Frameworks **memory consumption** with **PeakLens Original** model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in megabyte [MB]. 39

4.21	Frameworks memory consumption with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in megabyte [MB].	39
4.22	Frameworks memory consumption with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in megabyte [MB].	40
4.23	Frameworks battery consumption with PeakLens Original model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in milliampere hour [mAh].	41
4.24	Frameworks battery consumption with PeakLens Optimized model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in milliampere hour [mAh].	41
4.25	Frameworks battery consumption with MobileNet model - Bold values indicate the thread configuration, on each framework, that returns the best consumption. Green values indicate the best device consumption, comparing all the consumptions measured on different frameworks. Values are expressed in milliampere hour [mAh].	42

Acknowledgements

First of all, I would like to thank Professor Piero Fraternali for the opportunity and the advice provided throughout this thesis.

A special thanks to Federico Milani for his patience, experience and aid to make this thesis more incisive.

I would also like to thank my family for the unfailing support and the countless opportunities provided throughout these years.

Finally, I would like to express my gratitude to all my friends that have helped me during this unforgettable journey.

This milestone would have not been reachable without all of you.

